Scalable Applications on Heterogeneous System Architectures: A Systematic Performance Analysis Framework

Dissertation

zur Erlangung des akademischen Grades Doktoringenieur (Dr.-Ing.)

vorgelegt an der
Technischen Universität Dresden
Fakultät Informatik

eingereicht von

Robert Dietrich
geboren am 10. August 1983 in Dresden

Gutachter:
Prof. Dr. rer. nat. Wolfgang E. Nagel
Technische Universität Dresden,
Prof. Dr. rer. nat. Matthias S. Müller
Rheinisch-Westfälische Technische Hochschule Aachen

Tag der Einreichung: 6. Mai 2019
Tag der Verteidigung: 9. Oktober 2019

Abstract

The efficient parallel execution of scientific applications is a key challenge in high-performance computing (HPC). With growing parallelism and heterogeneity of compute resources as well as increasingly complex software, performance analysis has become an indispensable tool in the development and optimization of parallel programs. It is a recurring task as HPC systems and their software stack are regularly replaced and applications have to be ported to the new execution environment.

This thesis presents a framework for systematic performance analysis of scalable, heterogeneous applications. Based on event traces, it automatically detects the critical path and inefficiencies that result in waiting or idle time, e.g. due to load imbalances between parallel execution streams. The building blocks of the analysis are patterns of inefficient execution in the parallelization at process and thread level and in computation offloading. The latter is, compared to the other two, a relatively new programming model in HPC. As a prerequisite for the analysis of heterogeneous programs, this thesis specifies inefficiency patterns for computation offloading. Furthermore, an essential contribution was made to the development of tool interfaces for OpenACC and OpenMP, which enable a portable data acquisition and a subsequent analysis for programs with offload directives. The specified runtime events also enable the tracking of dependencies between tasks and thus between program regions on the host and offloaded tasks. At present, these interfaces are already part of the latest OpenACC and OpenMP API specification.

The aforementioned work, existing preliminary work, and established analysis methods are combined into a generic analysis process, which can be applied across programming models. Based on the detection of wait or idle states, which can propagate over several levels of parallelism, the analysis identifies wasted computing resources and their root cause as well as the critical-path share for each program region. Thus, it determines the influence of program regions on the load balancing between execution streams and the program runtime. The analysis results include a summary of the detected inefficiency patterns and a program trace, enhanced with information about wait states, their cause, and the critical path. In addition, a ranking highlights program regions that are relevant for program optimization. The ranking criteria is the amount of waiting time a program region caused on the critical path.

The thesis concludes with a description of the performance analysis framework, its implementation, and application. The scalability is demonstrated using High-Performance Linpack (HPL), while the analysis results are validated with synthetic programs. A scientific application that uses MPI, OpenMP, and CUDA simultaneously is investigated in order to show the applicability of the analysis.

Contents

1 Introduction

Most parallel programs offer optimization potential, which, if left unused, wastes an unpredictable amount of computer resources. Especially in high-performance computing (HPC), where scientific simulations can execute on thousands of interconnected compute nodes concurrently, a small inefficiency can have a large impact at scale. In addition, the hierarchical parallelism of computing systems with heterogeneous hardware increases the complexity of application development and thus the challenge of efficient parallelization and load balancing. In November 2018, seven systems in the top 10 of the TOP500 [MSD+] fastest supercomputers in the world were heterogeneous systems with accelerators or coprocessors.

Performance analysis is an indispensable tool to detect execution inefficiencies and optimize applications for such complex systems. As waiting or idle time can propagate over multiple levels of parallelism, e.g. from a delayed task on an accelerator over host threads to another compute node, the actual cause of an inefficiency might be difficult to find. Hence, there is a need for a holistic analysis that covers all parallelization layers in a program.

Due to the regular replacement of HPC systems and changes in the software environment, performance optimization is also a recurring task. It can be significantly shortened by prioritizing inefficiencies and program regions appropriately and thus preventing unnecessary local optimizations that have a negligible influence on the overall runtime. Such a systematic approach requires a sophisticated analysis, which nevertheless can be executed in a timely manner, even for large-scale applications with hybrid parallelization.

In HPC, MPI+X has been established to develop scalable heterogeneous programs [GS13]. While the Message Passing Interface (MPI) is the defacto standard for communication between nodes, prominent X-models for node-local parallelization are OpenMP, OpenACC, OpenCL, and CUDA [MLP+17]. Due to the variety and the combined use of multiple X-models, performance analysis becomes more complex.

Main Contribution

This thesis proposes a framework for systematic performance analysis of scalable, heterogeneous applications, which covers process- and thread-level parallelism as well as computation offloading. It addresses two essential aspects that have so far been neglected: potential inefficiencies with computation offloading and generic analyses across programming models. Furthermore, established analyses are combined in such a way that inefficiencies and program regions can be prioritized to enable a more focused optimization process.

In HPC, computation offloading is a relatively new programming concept, compared to process- and thread-level parallelization. On the basis of common offload application programming interfaces (APIs), the underlying programming model is examined and generic inefficiency patterns are specified. In the context of this thesis, a significant contribution was also made to the specification of the tool interfaces for OpenACC and OpenMP, which build the basis for corresponding performance data collection and subsequent analysis.

Using this groundwork and existing preliminary work on performance analysis of MPI and OpenMP applications, a generic analysis is presented. Based on wait and idle states, it determines wasted computing resources and their cause as well as the influence of program regions on the load balancing and the overall program runtime. The analysis results are presented in form of a program region profile, a summary of all inefficiencies, and timelines. In the core analyses, the implementation is independent of APIs and can detect further inefficiencies and wait states by adding new analysis rules. It is applied to synthetic and real-world programs to validate the applicability, correctness, and scalability.

Structure of this Thesis

After the brief introduction to the topic of this thesis, the first part of Chapter 2 describes heterogeneous systems as a target platform for the developed performance analysis. The second part describes the most relevant programming models and APIs that are used on such systems, including a short introduction to computation offloading as an additional parallelization layer next to process- and thread-level parallelization. The following Chapter 3 discusses the state-of-the-art in performance monitoring and analysis as well as related software tools. After discussing previous work in this area of research, the following chapters present the contribution of this thesis. The first part of Chapter 4 specifies potential inefficiencies in the execution of programs with computation offloading. To enable a reasonable analysis of programs with OpenACC and OpenMP target directives, respective tool interfaces are presented in the second part of the chapter. Based on this preliminary work, Chapter 5 presents the proposed generic analysis for scalable heterogeneous programs and their requirements on the performance data. To validate the analysis, its implementation in a software tool as well as its application to synthetic programs and a scientific use case are described in Chapter 6. Moreover, the scalability is demonstrated. Chapter 7 concludes this thesis with a summary of the scientific contribution and an outlook on future work.

Formatting and Terminology

To ease reading, different font styles are used. *Italics* highlights the introduction of new terms or terms that are taken from the original sources or referenced figures. `Typewriter font` is used for function names and code segments.

The term *accelerator* refers to compute devices, which are attached to a host processor to speed up parts of the computational workload. As the implied acceleration is not given, other names are also used synonymously. For example, Intel calls their Xeon Phi extension boards *coprocessors*, whereas the programming APIs CUDA and OpenCL use the term *device*. The OpenACC API specification describes directives for accelerators and OpenMP provides device constructs for *target devices*. For abstraction, the terms *offloading device*, *compute device*, or only *device* are used.

In the context of this thesis, the terms *heterogeneous computing*, *heterogeneous program*, *heterogeneous hardware*, and *heterogeneous system* refer to the coexistence or combined use of central processing units (CPUs) and additional compute devices.

Computation offloading describes the programming model used by programming APIs such as CUDA, OpenCL, OpenACC, and OpenMP target. In HPC, this refers to the offloading of computational workload from a host CPU to other compute devices.

2 Heterogeneous High Performance Computing

Today's computing systems are usually composed of heterogeneous computing resources, often with additional many-core processors such as graphics processing units (GPUs) or Intel Many Integrated Core (MIC) coprocessors. Referring to the 52nd TOP500 list [MSD+] of the fastest supercomputers, a significant number of systems (137 out of 500, 27,4%, 7 in the top 10) utilize accelerator or coprocessor technology. There are several reasons for the success of many-core devices in general-purpose computing. One reason is the efficient execution (also in terms of energy) of massively data-parallel workloads. Another reason is the availability of GPUs in almost every computer, which provides virtually everyone with access to heterogeneous hardware.

The use of heterogeneous hardware introduces additional challenges for software development. Usually, it is necessary to use an additional programming model or API. Another layer of parallel compute resources also increases the complexity of load balancing. Furthermore, applications on heterogeneous systems should benefit from advantages of different hardware architectures and compensate their individual drawbacks, which requires knowledge of the functioning of each architecture. As a rule of thumb, serial and hardly parallel parts of an application should be executed on a multi-core processor with high single-core performance, whereas massively parallel parts are appropriate for offloading to many-core processing units such as GPUs or Intel MIC coprocessors.

The first part of this chapter, section 2.1, gives an overview on current heterogeneous HPC system architectures. Section 2.2 provides more details on the currently widely used many-core device types GPUs and Intel MIC. As basis for the contributions of this work, Section 2.3 introduces the offloading concept and the currently most relevant programming APIs for the development of scalable applications on heterogeneous system architectures.

2.1 System Architectures

Today's large-scale systems are composed of multiple compute nodes, which are connected through a network. A compute node is typically equipped with one or two multi-core CPUs, which share local memory in the form of dynamic random access memory (DRAM). The major number of large-scale systems and the fastest supercomputers in the world are distributed-memory systems, so called computing clusters, where nodes cannot address the memory of another node. Fast network interconnects are either based on InfiniBand or they are proprietary such as Intel's Omni-Path, NVIDIA's NVLink, and Cray's Aries interconnect. The most common network topologies for large-scale systems are tree or torus layouts [BDG+16].

There are also shared-memory systems based on non-uniform memory access (NUMA), where all CPUs share a large amount of (virtual) memory. Large systems of this kind are distributed shared memory systems, where the memory is distributed across nodes, but exposed as a shared resource. This allows programs to use a large amount of (shared) memory. However, the locality of the memory is hidden to the program. To achieve decent performance, remote memory accesses are tried to reduce to a minimum, as they introduce an extra penalty. Shared-memory systems are not used at very large scale, as ensuring coherence and memory consistency introduces additional costs [SSR95] and it is not required in most scalable applications, which use message passing instead.

In general, heterogeneity of computing resources implies multiple different compute units or processors, often also with different instruction set architectures (ISAs). Mostly, it means that a combination of CPU and many-core processor is used. Figure 2.1 shows the conceptional composition of a heterogeneous

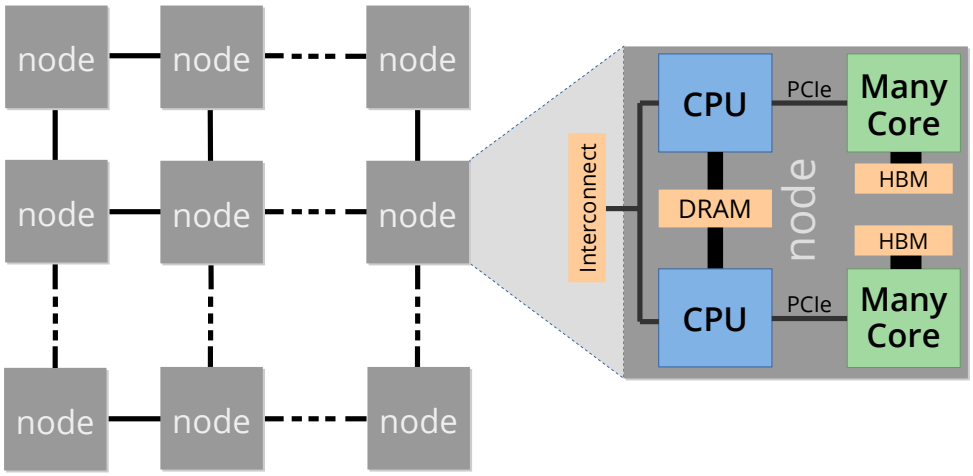


Figure 2.1: Heterogeneous HPC clusters use multiple types of processors, e.g. one or two CPUs and additional many-core devices per compute node. Different topologies are used to connect compute nodes (illustration shows a 2D mesh). The interconnects are either proprietary or they are based on InfiniBand or Gigabit Ethernet.

computing cluster, where each node is equipped with two CPUs and two GPUs. Current offloading devices for HPC are equipped with high bandwidth memory (HBM), which is physically separated from the node's main memory. Hence, a heterogeneous compute node is already a distributed memory system.

A major bottleneck of heterogeneous HPC compute nodes has been the limited bandwidth of the Peripheral Component Interconnect Express (PCIe) bus with about $32\,\mathrm{GByte/sec}$ for PCIe 3.0. It is used to communicate and exchange data between the CPU and the offloading device. GPUs from NVIDIA and AMD as well as Intel's Knights Corner (second generation of Intel's MIC) are connected via PCIe on systems with AMD or Intel platforms. Recent proprietary interconnects, such as NVIDIA NVlink, reduce the this communication bottleneck by providing a higher bandwidth. Nevertheless, it is still required to move data to and from offloading devices, which is a challenge for the development of efficient heterogeneous applications on most HPC systems, where offloading devices have their own memory.

2.2 Many-Core Processors

Two many-core hardware architecture types have established in HPC in the past years: GPUs and Intel's MIC architecture. To distinguish the use of GPUs for graphics processing from its use for general purpose computations, the term general-purpose computing on graphics processing units (GPGPU) has been introduced. Figure 2.2 illustrates basic similarities and differences between the architecture of CPUs, GPUs and Intel MIC. Table 2.1 provides a quantitative comparison of respective high-end products launched in 2016 of these three processor architectures. The theoretical peak performance of the Intel CPU has been determined as described in [Dol16].

Starting with the launch of the Tesla product series, NVIDIA released the first GPU for HPC in 2007, which was deployed in the 29th most powerful supercomputer in TOP500 list [MSD$^+$] in November 2008. The first Intel MIC product was released in 2012 under the branding of the processor product family "Intel Xeon Phi". In November 2012 the first system with Intel MIC technology (Stampede) entered the TOP500 list [MSD$^+$] directly in the top 10 at rank 7. The Green500 list [cFC07], a ranking of the most energy-efficient supercomputers in the world, is dominated by GPU-based systems.

Juckeland also describes in [Juc12] other accelerator technologies and special purpose solutions that did not gain enough market share and therefore were discontinued. Hence, this work will briefly explain only the concepts of GPUs, Intel MIC, and the combination of CPU and GPU as a hybrid processor. Other relevant many-core processors are the Matrix-2000, which is used as an accelerator in Tianhe-2A (rank 4

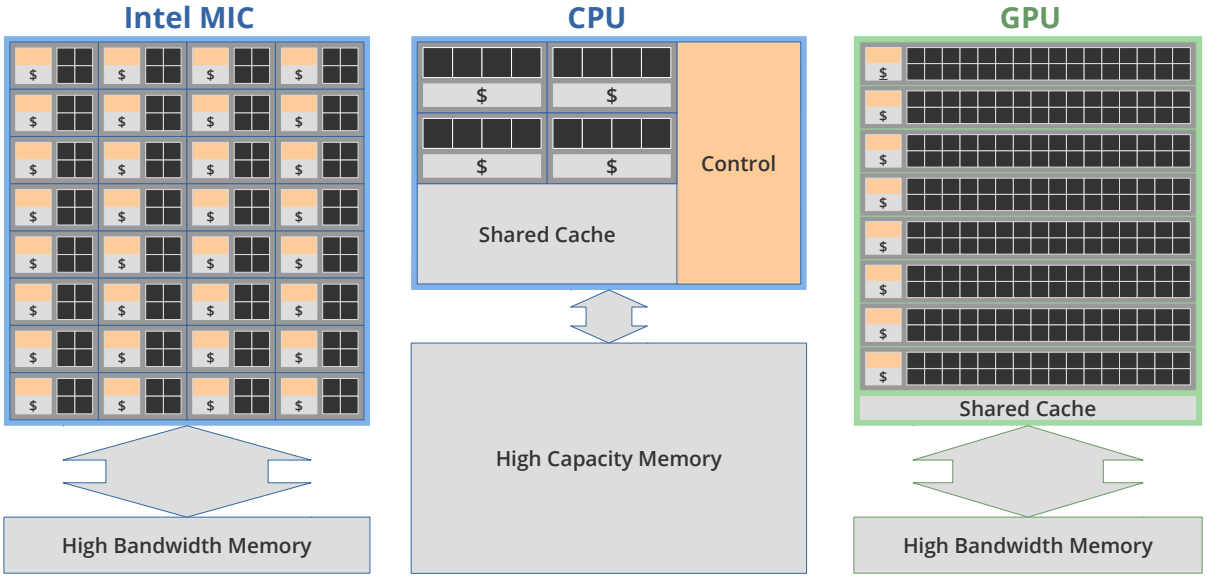


Figure 2.2: Heterogeneous compute nodes use multiple processor architectures. The dedicated memory of offloading devices is optimized for high bandwidth, whereas the usually larger CPU main memory is optimized for short latencies. Typical CPUs have more control logic than many-core processors. The latter use more chip area for arithmetic logic units (ALUs) and their core designs are less complex and their caches are smaller.

Feature	Xeon Phi 7290	CPU E5-2699v4	GPU Tesla P100
Number of Cores	72	22	3584 (FP32)
Max. Clock Frequency [GHz]	1.7	3.6	1.48
FP64 Peak Performance [GFLOPS]	3456	986	5304
Max. Memory Bandwidth [GB/s]	~400	76.8	720
Last-Level Cache Size [MB]	36 (L2)	55 (L3)	4 (L2)

Table 2.1: Performance indicators for an Intel Xeon Phi 7290 [Int16b], an Intel E5-2699v4 CPU [Int16c], and an NVIDIA Tesla P100 GPU [NVI16]

in TOP500 list [MSD+] from November 2018), and the PEZY-SC2, which is used as an accelerator in the most energy-efficient system according to Green500 in November 2018 [FS18]. Both are conceptional similar to the Intel MIC architecture.

2.2.1 Graphic Processing Units

Although GPUs are used for accelerated graphics visualization in the segment of desktop computers for a long time, their final breakthrough in HPC happened with the Titan system at Oak Ridge National Laboratory in 2012, the first GPU-based supercomputer with a peak performance over 10 petaFLOPS (PFLOPS). According to the 45th TOP500 [MSD+] list from June 2015, Titan was with 17.59 PFLOPS on the Linpack benchmark the second fastest supercomputer in the world.

Compared to the microarchitecture of CPUs and Intel MIC, GPUs have much simpler cores with less control logic and smaller caches. Hence, a larger percentage of the die (silicon) can be used for ALUs. GPU cores are grouped into so called streaming multiprocessors (NVIDIA) or compute units (AMD). They share resources such as a common cache and schedulers for GPU thread groups. Recent GPU architectures provide special purpose units, e.g. tensor cores in NVIDIA Volta GPUs [NVI17] for deep learning. Modern GPUs have a last-level cache (LLC), which is compared to the LLC in CPUs (most

often the L3 cache) much smaller. Similar to the L2 cache in CPUs, GPUs provide scratchpad memory that is shared across all ALUs in a compute unit (AMD) or streaming multiprocessor (NVIDIA). Furthermore, the scratchpad memory is managed by software and therewith a key property for the tuning of GPU kernels.

2.2.2 Many Integrated Core Architecture

The Intel MIC architecture uses the x86 ISA, which enables the use of programming APIs such as MPI, OpenMP and Pthreads. The first commercial MIC product is the *Knights Corner* (KNC) coprocessor board, which is connect via the PCIe bus with the host CPU. KNC boards can either operate in offloading mode as coprocessor (similar to GPUs), in native mode as independent processor that runs a whole application, or in a symmetric manner with the host. The latter mode allows some processes of a program to execute on the CPU and others on the Xeon Phi coprocessor. Nevertheless, it is necessary to use vectorization, multithreading, and data locality to achieve decent performance [JR13]. The fastest KNC coprocessors, e.g. the Xeon Phi 7120D, provide up to 61 cores with a total theoretical peak performance of 1.2 teraFLOPS (TFLOPS) (double precision). The enhanced P54C cores with in-order execution are capable of four-way simultaneous multi-threading (SMT) and provide a 512-bit-wide vector unit.

The successor of KNC is *Knights Landing* (KNL). Primarily designed as standalone processor, KNL is based on the Airmont microarchitecture with out-of-order execution, which improves the single-core performance over KNC. The up to 72 cores support AVX-512 instructions and provide a theoretical peak performance of about 3.5 TFLOPS (double precision). The offloading concept (see Section 2.3.1) is supported by offloading computations from a normal Intel Xeon CPU server to an Intel Xeon Phi server with Intel's proprietary OmniPath interconnect.

2.2.3 Hybrid Processor Architectures

Hybrid processors combine two types of processors on a single die, often with different instruction set architectures (ISAs). In the context of general purpose computing, a hybrid processors integrates a CPU and another processor. A common combination are CPU and GPU. AMD APUs, NVIDIA Tegra, Intel Sandy Bridge Core processors and their successors are commercially available hybrid processors, which address the desktop and the mobile personal computer market. The IBM Cell processor is another prominent hybrid processor architecture, where the Power Processor Element (PPE) controls the Synergistic Processing Units (SPUs).

An important characteristic of recent hybrid processors is the shared memory, which supersedes data transfers over external buses, such as PCIe. Since the Intel Sandy Bridge microarchitecture, all Intel Core processors also have a cache that is shared between the CPU cores and the on-die GPU.

The Heterogeneous System Architecture (HSA) is a cross-vendor standard, which covers the specification of a system architecture, a programmer's reference manual, and a runtime programmer's reference. It "is designed to efficiently support a wide assortment of data-parallel and task-parallel programming models."[Fou18] Furthermore, it allows the efficient integration of multiple processors with potentially different ISAs, e.g. a CPU and a GPU. An important requirement for HSA-conformance is a shared virtual memory for all processing units, which eases the programming of heterogeneous processor, e.g. due to enabling so called zero-copy operations [SAF12]. Nevertheless, data have to be copied between processing units if they do not physically share a memory. CUDA 6 also introduced unified virtual memory.

Other combination candidates for CPUs in hybrid processors are field programmable gate arrays (FPGAs) and potentially also Intel MIC. Intel Arria FPGAs combine ARM CPU cores as fixed logic and programmable logic that characterizes typical FPGAs. Due to the reconfigurability of FPGAs, the hardware can adapt to the needs of applications and not vice versa. In the context of embedded systems, hybrid processor architectures are called multiprocessor system-on-chips (MPSoCs).

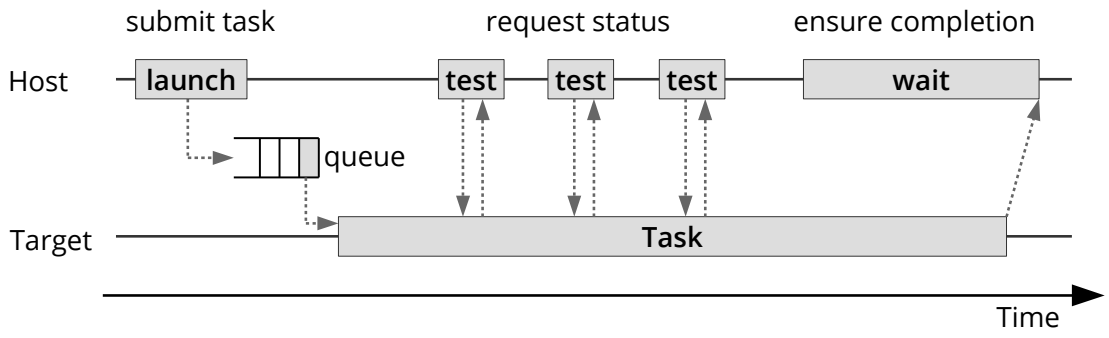


Figure 2.3: Asynchronous computation offloading

2.3 Programming Models

To effectively utilize the hardware resources of large heterogeneous computing clusters, several parallelization models and programming APIs have to be used in combination. There are three parallelization levels that need to be covered in HPC: inter-node parallelization, shared-memory parallelization and computation offloading. With regard to the programming APIs, MPI is the de facto standard for inter-process and therewith also inter-node communication, whereas OpenMP has become a widely-adopted programming API for shared-memory parallelization based on threads. In addition, a number of APIs for computation offloading have evolved to cope with the trend towards complex heterogeneous system architectures.

Section 2.3.1 introduces the concept of computation offloading. The following sections describe programming models and APIs that are used to develop heterogeneous HPC applications. If it is not further specified, this thesis refers to MPI 3.1, CUDA 9.0, OpenCL 2.2, OpenACC 2.5, and OpenMP 4.5. Common offloading APIs basically differ in the abstraction level for the programmer, which influences the performance and the portability between different compute devices.

2.3.1 Computation Offloading

Computation offloading is the process of transferring the execution of computational tasks to a target processor, device, or system. It describes a host-centric programming model, where the host triggers the offloading of tasks, may request their execution status, or wait for offloaded tasks. Figure 2.3 illustrates the concept in a timeline visualization. Offloaded tasks are typically submitted to a target queue before they start executing. This allows the target device to autonomously schedule tasks on the assigned resources when they are available.

Offloading can be performed either synchronous or asynchronous with the host execution. In the first case, the host triggers the offloading task and directly waits until it is completed. Asynchronous offloading gives the control back to the host, after the task has been submitted to its target. The host can continue execution and start waiting for the target if it cannot proceed otherwise. Synchronization or wait operations are a potential waste of computing resources and hence an inefficiency, which should be considered in application development and performance analysis.

A reasonable computational task generates output data and most often also consumes input data. Hence, computation offloading requires data movement between the host and the target resource, if both do not share the same memory. In the discussed offloading models, the host also initiates the allocation and deallocation of memory on the target device as well as the data transfer between host and device. In HPC, workloads are offloaded to specialized compute devices mostly via the PCIe bus or vendor specific interconnects such as NVIDIA's NVLink within a compute node. With the Xeon Phi processor x200 (KNL architecture) Intel introduced *offload over fabric* [Int16a], which enables offloading to other compute nodes.

For mobile devices there are obvious reasons for offloading of computations to remote devices [KLLB13], e.g. the mobile device cannot execute a task due to resource limitations (not enough main memory or insufficient compute resources), reducing the load to save energy on the mobile device, or improve performance. The latter is the main reason for computation offloading in HPC. However, energy efficiency is currently a key factor to build exascale systems [BP13]. GPUs and Intel MIC processors can process massively data-parallel workloads more efficiently than CPUs, which enables faster execution and energy savings.

Computation offloading has some similarities with multiple program, multiple data (MPMD) parallelization and task-based programming approaches. In the MPMD model, different programs can be executed simultaneously. Each program uses different data. This is similar for computation offloading, where the host and all its targets can execute a different program and process different data. However, the host controls the programs of the offloading targets.

Computation offloading uses the term *task* as a unit of work, which is executed on a target resource. Similar to other task-based models, it allows dependencies between tasks. Other inherent computation offloading dependencies include the task trigger on the host to start before the associated offloaded task and the host wait operation for a task to end after the associated task.

2.3.2 Low-Level Offloading APIs

The offloading of general purpose computations to GPUs, hardware that was initially optimized for graphics, became popular with the introduction of programmable shaders and respective APIs such as Directx and OpenGL. Programming APIs such as CUDA and OpenCL hide most of the underlying graphical concepts in favor of general-purpose-computing capabilities. The proliferation of NVIDIA GPUs in desktop and HPC systems has established NVIDIA's proprietary programming model CUDA for GPGPU. According to the TOP500 list [MSD+] from November 2018, rank one and two as well as three other supercomputers in the top ten are equipped with NVIDIA GPUs. OpenCL is a vendor-independent open standard which can be applied on a large variety of target devices, which provides a portability advantage over CUDA.

CUDA

In November 2006 NVIDIA introduced the Compute Unified Device Architecture (CUDA), the first programming model for GPUs that did not require in-depth knowledge of graphics programming. It extends the C programming language with new constructs to allow the execution of subroutines, so called kernels, on the GPU. The programming of such device kernels is described in detail in [NVI18a].

Kernels are passed to the GPU scheduler and executed as asynchronous tasks sequentially within a device execution stream (see Figure 2.3). Since CUDA 9 and the NVIDIA Pascal architecture, a single kernel instance can be executed on multiple GPUs of the same type (see cooperative groups in [NVI18a]). Data transfers are handled similar to kernels as tasks, which are submitted to a device stream. Multiple device streams can be associated with a single GPU to enable the concurrent processing of tasks on a device. CUDA events are submitted to a device stream to describe inter-stream dependencies, take timestamps on the device, or serve as synchronization point for the host.

CUDA also describes a general-purpose parallel computing platform [NVI18a], where different hardware revisions or GPU generations provide different compute capabilities. A higher capability means more functionality with 1.0 being the lowest.

OpenCL

The Open Computing Language (OpenCL) [Khr19] is an open standard that abstracts from hardware-specific APIs such as CUDA and enables portable access to compute devices from numerous different

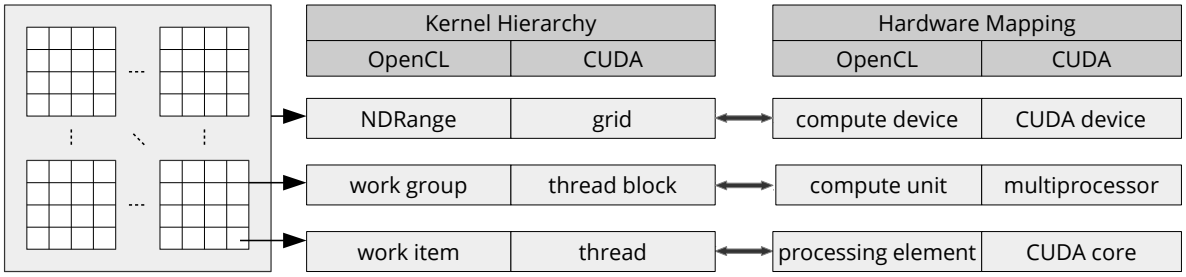


Figure 2.4: Hierarchical decomposition and hardware mapping of device kernels with OpenCL and CUDA terminology

vendors. Therefore, it can be used for programming of heterogeneous computing systems that are composed of diverse compute resources. It is also intended to run on supercomputers and personal computers as well as on embedded systems and mobile devices.

OpenCL consists of an API for coordinating parallel compute resources, compute kernels and buffer objects as well as a kernel language that utilizes a subset of ISO C99 with extensions for parallelism to program the compute devices. The programming model uses the concept of command queues to enable concurrent processing of tasks on a single device. A device task, such as a compute kernel and a data transfer, can be associated with an OpenCL event, which can be used to describe task dependencies (also between different command queues), gather timing information of tasks, query the task's execution status, and wait for the completion of a task.

CUDA and OpenCL Device Kernels

The programming of parallel device kernels in CUDA and OpenCL follows a similar principle. Many threads or workers concurrently execute the kernel code, which is a sequential description of instructions. As each of them has a unique identifier, they span an index space and can operate on different data. Kernels are hierarchically decomposed to map to CUDA's and OpenCL's hardware model.

In CUDA terminology, a kernel spans a grid of thread blocks, whereas OpenCL defines a kernel as an n-dimensional range (NDRange) of work groups. The smallest structural unit is a CUDA thread or an OpenCL work item. Figure 2.4 illustrates the hierarchical decomposition of the computation index space of device kernels for two dimensions. It also shows the mapping of the kernel hierarchy to the hardware model, which is similar, except for the terminology.

A CUDA device is composed of multiple simultaneous multiprocessors (SMs), which themselves group a set of CUDA cores. The cores in an SM share a scratchpad memory for fast data exchange. The slower global device memory has to used for data exchange between SMs. OpenCL devices are decomposed into compute units, which contain a set of processing elements and a shared scratchpad memory.

CUDA's hierarchical execution model further specifies warps as a group of 32 CUDA threads. All threads in a warp execute the same instruction simultaneously, which is similar to vector processing. Different warps can execute different instructions at the same time and therewith provide thread-level parallelism. Context switches between warps enables the SM to hide memory latencies.

2.3.3 Directive-Based Programming APIs

Parallel programming based on compiler directives is an alternative to threading APIs such as Pthreads and low-level offloading APIs such as CUDA and OpenCL. Instead of function calls, parallelization hints for the compiler are inserted as comments in the sequential source code, which maintains the code basis for sequential execution, but also allows compilers to generate efficient parallel code. Hence, compiler directives provide a non-intrusive way to port sequential code to parallel programming models.

Directive-based programming APIs typically try to hide hardware-specific details to uniformly program new architectures. Implementation effort is shifted from the application developer to compilers and runtime systems, which avoids a time-consuming and difficult rewrite of application kernels. However, it is still necessary to expose the program's parallelism in a reasonable way.

To simplify the programming of heterogeneous systems and provide a less intrusive way to introduce computation offloading in existing codes, directive-based offloading APIs have emerged. Widely used are OpenMP [OMP18] and OpenACC [OAC18]. They are both developed and standardized by a consortium and are supported by numerous commercial and open-source C, C++ and Fortran compilers which support different target platforms. OpenMP has been initially designed to enable thread-level parallelism on shared memory systems. Recent extensions to the specification also enable the programming of so called target devices that might have a separate memory address space. OpenACC is focusing on computation offloading and has established as an effective alternative to CUDA and OpenCL.

Other directive-based offloading approaches are Intel's Language Extension for Offloading (LEO) and HMPP [DBB07]. LEO is proprietary developed by Intel and has mostly been merged into OpenMP 4.0. HMPP has been discontinued and its feature set is almost fully covered by OpenACC.

OpenACC

OpenACC has been designed to create high-level host+accelerator programs without the need to explicitly initialize the accelerator, manage data or program transfers between the host and accelerator, or initiate accelerator startup and shutdown. The OpenACC standard defines a host-directed execution model with an attached accelerator and a memory model that allows data movement to be implicit or explicit between host and accelerator.

OpenACC defines a hierarchical execution model for compute regions on the device, which is similar to CUDA and OpenCL device kernels (see Section 2.3.2). A compute region is executed by a number of gangs, each with one or more workers, which may execute vector operations. A possible mapping of gang, worker, and vector to CUDA's kernel execution model are thread block, warp, and threads. However, this mapping depends on the target architecture and the compiler.

OpenACC provides the fundamental constructs *kernels* and *parallel* for computation offloading. The *kernels* construct defines a region of the program that shall be executed on the device using a sequence of kernels. It gives control over any parallel execution to the compiler that will try to parallelize loop nests in the *kernels* region. Gang size, worker size and vector length may be different for each kernel. The *parallel* construct defines the parallel execution on the device explicitly. When it is encountered during the program execution, one or more gangs of workers are created. Gang size, worker size and vector length will remain constant until the end of the region. One worker in each gang starts executing the code until a *loop* construct is encountered. Work is shared across gangs and workers for the execution of the immediately following loop. Code within a *parallel* region but outside of a *loop* construct is executed redundantly by all gangs (see [OAC18]).

The OpenACC standard defines the *data* construct and the *enter data* and *exit data* directives to create data regions. In combination with *data* clauses they are used to control data lifetime on the accelerator as well as data motion between host and accelerator. Data clauses that annotated to compute constructs implicitly create a data region. The update directive is used to update data on the device with its corresponding data on the host or vice versa.

The *async* clause enables asynchronous data movement and computation on the device. It may have a single argument to identify the respective device operation or select an asynchronous device activity queue. When there is no *async* clause on a compute or data construct, the host waits until the device finishes execution, as if a *wait* clause has been added to the construct. The OpenACC standard also specifies the standalone *wait* directive and the *acc_wait** API routines, which block execution of the host thread to synchronize with asynchronous operations or activity queues on the device. If the *async* clause is used on a *wait* directive, a synchronization point is added to the device activity queue which prevents newly enqueued activities to execute before this point.

OpenMP

OpenMP is one of the most widely used programming APIs for thread-level parallelization on shared-memory architectures. Several additional paradigms such as tasking, offloading and single instruction, multiple data (SIMD) processing have been added over time. The threading concept is a typical fork-join model. The fundamental construct *parallel* annotates code regions that shall be executed in parallel. When a thread encounters it, a team of threads is created to execute the *parallel* region, whereby the encountering thread gets the master thread of the team.

The OpenMP standard defines *worksharing* constructs to distribute the work among the members of a thread team. The following worksharing constructs are defined in the API: *loop*, *sections*, *single* and *workshare*. Consult the OpenMP specification [OMP18] for a detailed description. At the end of a *worksharing* construct there is an implicit barrier, unless a *nowait* clause is specified.

An explicit barrier is generated with the standalone *barrier* directive. This executable directive creates a synchronization point (and also a *task scheduling point*) for a team of threads, which cannot continue execution beyond the barrier until all threads in the team and all explicit tasks generated by the team are completed. When a thread is waiting in a barrier it is potentially wasted time which makes them a reasonable investigation objective for performance analysis.

OpenMP introduced a tasking model in version 3.0 of the standard, which has been revised in subsequent specifications. With OpenMP 4.0 several new constructs address the architecture of many-core processors. The *simd* construct enables vectorization of loops to leverage the full potential of processors with wide vector units, e.g. the Intel MIC architecture. The *distribute* and *teams* constructs address the hardware architecture of GPUs by enabling to distribute work across teams of threads, which is similar to gang parallelism in OpenACC or thread blocks in CUDA.

OpenMP for Computation Offloading

To address the programming of heterogeneous systems, OpenMP 4.0 has introduced *device* constructs, which enable the execution of code regions on a target device. Similar to OpenACC, target data regions are used in OpenMP to control data lifetime on the target device and data motion between host and device. Data motion operations can be explicitly performed using the *map* clause or the *target update* construct. The target construct is an executable directive that annotates code regions to be executed by a device. Except for *target* constructs, the behavior of all other OpenMP constructs is specified when enclosed in a *target* region.

OpenMP 4.5 introduces the *target task*, which integrates OpenMP's offloading concept into its tasking model and therewith enables the programmer to specify dependencies between the execution of target regions and other OpenMP tasks. A *target*, *target update*, *target enter data*, or *target exit data* construct generates a *target task*, which is executed on the host. The structured block of the target construct is executed as an offloaded task on the device, similar to a compute kernel or data transfer in OpenCL and CUDA.

The *nowait* clause on a *target* construct enables asynchronous offloading (see Section 2.3.1). If the *nowait* clause is not present, the *target* region is executed immediately by the encountering task which blocks the execution on the host thread while the device is executing the enclosed code block. Synchronization of target tasks is performed at task synchronization points such as *taskwait* and *barrier* regions. OpenMP does not define a query mechanism to request the execution status of offloaded tasks. Instead of multiple sequential device queues, OpenMP uses its tasking model with task dependencies for ordered and concurrent execution of offloaded tasks.

2.3.4 Other Programming Abstractions for Heterogeneous Systems

Besides the presented offloading APIs, several programming abstractions for heterogeneous computing have been developed. They address two major challenges: performance portability and load balancing.

The first group, which focuses on performance portability to different hardware types, are C++ abstraction libraries, such as KoKKos [ETS14], RAJA [HK14], and Alpaka [ZWW+16]. They all require compiler support for C++ lambda expressions and support several backend APIs for node-level parallelism, including CUDA and OpenMP. KoKKos and RAJA define their own programming model and have been designed to explore the parallelism of existing sequential code. Both use a forall-template method on loops with some form of execution policy, which basically means that the loop body is separated from its traversal. This allows the implementation of different hardware-specific memory access patterns. Alpaka combines concepts from CUDA, OpenCL, and OpenACC into a single source code abstraction. Instead of describing loop nests, it uses the concept of kernels, which are executed by each thread in the grid, similar to CUDA and OpenCL. Alpaka defines the three hierarchy levels known from CUDA (grid, block, and thread) and adds element as a fourth level to address vectorization. The parallelization hierarchy level corresponds to a particular memory level. Data accesses and transfers are transparent and not optimized by the Alpaka library.

The second group focuses on load balancing based on tasks and data dependencies. StarPU [ATNW11] and StarSs [PBAL09] are task-based programming models which provide a uniform way to program heterogeneous computing architectures. The respective runtime systems schedule tasks, resolve data dependencies, and automatically generate data transfers to hardware targets with physically distributed memory. Both define an own API, which allows program developers to declare task and data dependencies. OmpSs [FBM+14] is an implementation of StarSs, which supports several OpenMP task directives. It has been used as a research environment for the support of task dependencies in OpenMP. The additional *implements* clause allows the program developer to write multiple implementations of a task and gives the runtime scheduler more flexibility in distributing tasks on the available devices. StarPU also supports OpenMP task directives [AAB+17], since OpenMP 4.0 introduced task dependencies. Both runtime systems rely on the compiler to generate appropriate code for offloading devices or libraries that use the offloading devices. Hence, custom device kernels still have to be programmed in native device languages. The Parsec framework [BBD+13] is also able to dynamically schedule tasks over computing resources on distributed heterogeneous compute nodes. Furthermore, it can generate an internal directed acyclic graph (DAG) representation from the serial code, without the need for additional annotations.

Legion [Bau14] is more data-centric programming model, which includes a runtime that schedules tasks. It uses so called *logical regions* to partition program data, which is used to express data locality and also data independence. Tasks execute the code and access logical regions, which allows the Legion runtime to identify independent tasks and enable concurrent task processing on heterogeneous platforms. The hardware mapping has to be defined by the program developer based on Legion's mapping interface.

Charm++ [RBK16] also focuses on efficient workload distribution. Therefore, it provides a framework for scheduling and execution management of heterogeneous tasks, including automatic optimizations, such as overlapping between computations and data transfers. Besides the management of native accelerator kernels, it allows the automatic generation of accelerator code, which is based on source-code annotations. Charm++, Legion, and OmpSs distribute work over multiple compute nodes based on MPI.

As the presented programming abstractions build on top of programming APIs such as MPI, OpenMP, CUDA, and OpenCL, they inherit their execution inefficiencies (compare Section 3.3.2 and 4.1). Offloading libraries, such as cuBLAS and cuFFT for NVIDIA GPUs, offer an even higher level of abstraction and usually also the best performance. However, they only provide a limited set of algorithms for special device types and are therefore not very flexible to use.

2.3.5 Inter-Process Communication

To develop scalable applications for large computing clusters, communication between nodes is necessary. State-of-the-art HPC applications often use message passing to scale across compute nodes. Partitioned Global Address Space (PGAS) approaches are less often used, although they provide some advantages over message passing.

Message Passing Interface

MPI is the most widely-adopted message passing interface for communication between processes. The open-source projects OpenMPI and MPICH provide high-performance implementations of this interface. "MPI addresses primarily the message-passing parallel programming model, in which data is moved from the address space of one process to that of another process through cooperative operations on each process" [Mes15].

Most scientific HPC applications are single program, multiple data (SPMD) codes. This basically means that all processes execute the same program on different data. MPI is most often used to communicate or synchronize between processes. However, there are also MPI-based MPMD codes, where different processes execute different programs, e.g. multi-physics codes. MPI supports both execution modes, SPMD and MPMD.

The MPI standard specifies several communication types: Blocking and non-blocking point-to-point and collective operations as well as one-sided remote memory access (RMA) operations including synchronization calls. MPI point-to-point requires active communication partners: sender and receiver. In case of blocking point-to-point communication, the respective MPI routines block until the operation is completed. For example, *MPI_Recv* blocks the execution of the process until the data is received. *MPI_Send* blocks until the data has been sent, which however does not guarantee that the data has already been received. MPI collectives are communication or synchronization operations on a group of processes, e.g. a barrier, a broadcast or a gather operation. Non-blocking MPI point-to-point communication and collectives just trigger the messaging, the data transfer or synchronization itself is performed afterwards. The completion can be force with MPI wait operations. To avoid a blocking wait, it is also possible to test for the completion of a non-blocking operation.

MPI one-sided RMA communication uses a per-process memory window which is accessible by other processes. A communication requires only one active process, which specifies all necessary information. MPI one-sided also defines synchronization mechanisms, which define the ordering of data accesses or force completion of all data accesses in a window.

Blocking communication and synchronization operations induce waiting time, which makes them a reasonable point for investigation in performance analysis. Section 3.3.2 describes inefficiencies that might occur in MPI applications.

Partitioned Global Address Space

The PGAS programming model intends to combine the advantages of shared memory and distributed memory programming models. These are the simplicity of data access using a global address space (shared memory) and the data locality in message passing models. Remote references are resolved by the compiler or PGAS library into inter-process communication. The address space is partitioned in a way that each process has local memory but also shares memory with an affinity to exploit locality. The latter is useful to improve the performance, because a process can access local data faster than remote data. One-sided communication, which is also available in MPI, is intended to improve inter-process performance compared to two-sided communication.

PGAS languages can be built on top of MPI libraries since version 3.0 that introduced one-sided communication. However, this will typically result in limited performance as MPI libraries are often not optimized for one-sided communication [DBH+12, HGC14]. It is more reasonable to build PGAS languages using libraries that are optimized for one-sided communication and support the concept of a global and distributed memory. GASPI (Global Address Space Programming Interface) [ABB+13] and SHMEM are PGAS APIs. Examples for PGAS programming languages are Coarray Fortran, Chapel, and X10 [DWMDF+15].

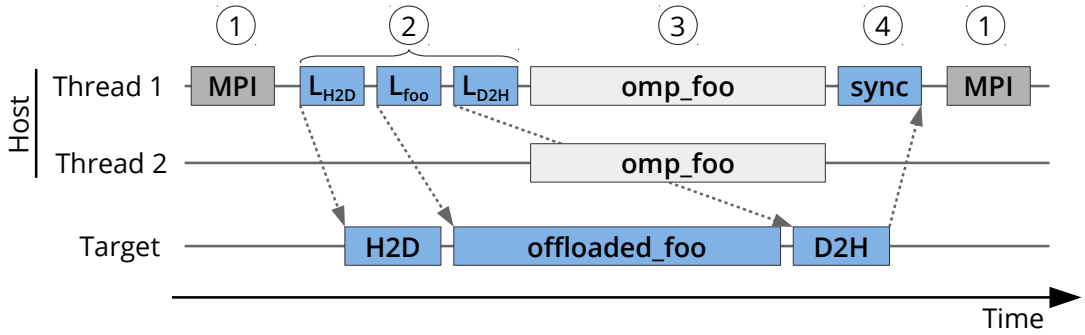


Figure 2.5: Exemplary execution pattern of hybrid MPI, OpenMP, and offloading: (1) MPI communication or synchronization with other processes, (2) host triggers offloading tasks (host-to-device data transfer, computational task, device-to-host data transfer), (3) OpenMP parallel execution (on two threads), (4) and synchronization of the asynchronously offloaded tasks.

2.3.6 Hybrid Programming Models

Hybrid programming combines multiple programming models to effectively address different parallel layers in the architecture of computing systems. Primarily, this requires an appropriate mapping of programming model and hardware resource. Message passing enables communication between compute nodes (different address spaces). Threads are used for efficient parallelization in a shared memory domain, e.g. within a compute node. Computation offloading is typically used to move highly data-parallel workloads to many-core processors.

In a hybrid MPI/OpenMP program, MPI ranks normally do the communication, while OpenMP threads perform the computation in parallel on available compute cores. As computation and communication often do not overlap, most cores are idle during the communication part. Typically, this cannot be avoided without algorithmic changes. Another challenge is the arrangement of processes and threads, as both can be used to parallelize computation. The optimal setup of threads and processes is difficult to determine [CH01, DK04]. It depends on many factors, including the system's hardware and its configuration, the MPI and OpenMP implementations being used, and the data locality [DWK+17]. On dual-socket systems, a typical setup uses one MPI rank per CPU and one thread per available CPU core.

Computation offloading adds another parallel execution layer, which usually requires data transfers between host and device (see Section 2.3.1). Host-device communication results in device idle, if it cannot be hidden behind computation. Furthermore, the memory access has to be optimized for different processors [MMBS16], which may require data transformations. Due to the different hardware characteristics of multi-core and many-core processors, the workload cannot be evenly distributed, which results in another challenge in terms of load balancing (see Section 3.1.3).

To optimize data transfers between GPUs on different compute nodes, there are GPU-aware implementations of MPI [WPB+14]. They ease the data movement in MPI+GPU applications, such that the programmer can pass GPU memory pointers to MPI functions using a unified virtual address space. This also enables a faster data exchange between GPUs, as the MPI library can directly send and receive GPU buffers without the need to explicitly stage them in host memory first.

There is no prescribed hierarchy between message passing, multithreading, and computation offloading. However, there are common usage patterns, which are driven by data access latencies and bandwidths. Data movement between nodes is typically slower than between a CPU and an offloading device, which in turn is slower than shared memory accesses of threads. Hence, many applications use the pattern shown in Figure 2.5, which allows concurrent computations on host and offload device. Potential inefficiencies in the OpenMP parallelization and waiting times in MPI communication are disregarded in the figure, but discussed separately in Section 3.3.2. Section 4.1.4 evaluates combination options of computation offloading with message passing and multithreading.

2.4 Conclusion

Heterogeneous systems have already established in HPC. Currently, the most common many-core processors are GPUs [MSD+], usually installed on expansion cards with dedicated memory. Hence, a single compute node is already a distributed-memory system, which requires memory transfers to exchange data between CPU and many-core processors. Processor chips that combine several processor types and have a common memory are so far rarely used in HPC.

Additional compute devices are usually controlled via the parallelization model computation offloading, which is implemented in four known APIs: CUDA, OpenCL, OpenACC, and OpenMP. Based on these offloading APIs, there are several abstraction libraries, which can partially hide the additional parallelization layer and simplify programming. Nevertheless, the potential inefficiencies of computation offloading remain.

There are also several programming APIs for inter-process communication and multithreading, with MPI and OpenMP being the dominating representatives. In summary, hybrid parallelization or parallelization over multiple levels of parallel compute resources is indispensable for the efficient programming of current heterogeneous systems. Nevertheless, available programming models and APIs hardly consider performance portability, which remains a challenge for software development for heterogeneous systems.

3 Application Performance Analysis

The complexity of computer architectures and software environments for parallel computing is a tough challenge for the efficient execution of parallel programs. A difficult but important task is load balancing, which causes poor resource utilization if performed inappropriately [CVKG10, PMM+15]. Performance analysis enables a more efficient use of computer resources by identifying inefficient execution behavior and other performance-critical optimization targets, such as runtime-dominating code regions. It is an essential part in the development cycle of applications and a vital step to generate optimized code.

This chapter provides an overview of the state of the art in performance analysis of parallel programs. As a starting point, Section 3.1 describes performance challenges for such programs. Subsequently, performance-analysis steps are introduced in Section 3.2, whereupon Section 3.3 focuses on the analysis of performance data. Section 3.4 presents well-known performance analysis tools. The chapter closes in Section 3.5 with a summary of the most important findings as a basis for the contributions of this work.

3.1 Performance Challenges in Parallel Programs

Parallel and serial performance aspects characterize the execution of a parallel program. The latter include the computational efficiency of the single-threaded code and the efficient use of the processor's memory hierarchy. However, the focus of this thesis are parallel performance challenges. They include the mapping of processes and threads to the compute hardware, communication and synchronization between processes, threads, and offloading devices as well as workload balancing and idle compute resources. In addition, the complexity of program codes, the need for hybrid programming (see Section 2.3.6), or inefficiencies that only appear in large-scale executions amplify the parallel performance challenge.

3.1.1 Scalability

Good scalability of an application is one of the main challenges, to make efficient use of large-scale computers. Scalability considers the speedup of an application when increasing its parallel processing resources. In an ideal case the speedup is equivalent to the number of parallel processing resources [Hil90]. Hence, the efficiency of parallelization determines the scalability of an application.

Scalability can be distinguished between computational and communication scalability. The former considers the workload or problem size of a program. In HPC, strong and weak scaling are distinguished. Strong scaling varies the compute resources for a fixed problem size, which, according to Amdahl's law [Amd67], makes the serial portion of a program a limiting factor for the speedup. Weak scaling varies the problem size and the compute resources proportionally. It is governed by Gustafson's law [Gus88], which assumes that the serial fraction of a program does not increase with the problem size.

Communication scalability determines the efficiency of data transfer or synchronization operations when the number of communication partners is increased. It depends on the communication pattern, the underlying network topology, and the implementation of the communication library in use. For example, a broadcast can be implemented in several ways, e.g. by sending individual messages from the root process to all participating receivers or with a binary tree where messages traverse a tree [PGAB+07]. Communication may also depend on the placing of processes and might be improved by matching logical communication with the underlying physical network topology.

3.1.2 Parallelization Overhead

The parallelization of a program causes overhead in terms of data movement, synchronization between concurrent control flows, and execution management. The total amount of time that is spent in such operations can be denoted as parallelization overhead or parallelization costs. It can be divided into the following categories:

Communication: data transfers and synchronization between processes, threads, and offloading devices
Management: initialization and finalization of processes, threads, and offloading devices
Extra Work: computational work that is not needed in the sequential execution

Minimizing the parallelization overhead is the key to maximizing the speedup of parallel processing and consequently to improve the parallel efficiency of a program. Formula 3.1 illustrates the impact of the parallelization overhead on the parallel efficiency as defined in [EZL89] (for n processes).

$$E_n = \frac{t(\textit{serial execution})}{n * t(\textit{parallel execution})} = \frac{t(1)}{n * \max\limits_{1 \leq p \leq n} (t_{\textit{work}}(p) + t_{\textit{parallel\_overhead}}(p))} \quad \Big| \quad t \text{ ... } \textit{time} \tag{3.1}$$

Hence, an increase in the parallelization overhead $t_{\textit{parallel\_overhead}}(p)$ decreases the parallel efficiency E_n, if the parallel computing time $t_{\textit{work}}(p)$ remains constantly. The parallelization overhead can be mostly determined from a function profile, although the extra work might be hard to expose.

Some parallelization costs are unavoidable, even in embarrassingly-parallel applications, because processes, threads, or other compute devices have to be managed (e.g. initialized) and data must be distributed to the respective compute resource. Therefore, it is reasonable to distinguish between avoidable parallelization overhead and necessary parallelization costs. For example, waiting times due to load imbalances are avoidable, whereas real communication and synchronization time might be necessary. This thesis focuses on the detection and evaluation of waiting times to expose load imbalances and their causes, which, however, requires the analysis of temporally ordered events instead of profile data (see Section 5.1).

Rosas et al. [RGL14] define the parallel efficiency as a product of the load-balance efficiency, the serialization efficiency, and the transfer efficiency, which are determined based on program traces Thus, they distinguish the impact of three potential performance problems: load imbalances, serialization due to data dependencies, and data movement costs.

3.1.3 Load Balancing and Resource Contention

Load balancing across compute resources of massively-parallel systems is a difficult task and certainly a key factor for good parallel performance. For an SPMD execution on homogeneous compute resources, load imbalance can be defined as the differences in workload between processes [Her09], which means that the optimal speedup is achieved by evenly distributing work across processes. For the MPMD model, additional balancing is required between groups of processes executing the same program. Besides the workload, application developers have to balance the communication, which is subject to dynamic effects of the interconnect network and its topology.

Workload balancing on heterogeneous compute resources introduces an additional challenge. To achieve an optimal speedup, work needs to be distributed according to the compute capabilities of the available processing hardware. Considering the energy efficiency, a reasonable mapping of workloads to suitable compute devices is required. It might be even useful to exclude compute devices from workload balancing, e.g. to minimize the power per computation. However, this thesis focuses on reducing the overall program runtime, which most often also reduces the energy footprint.

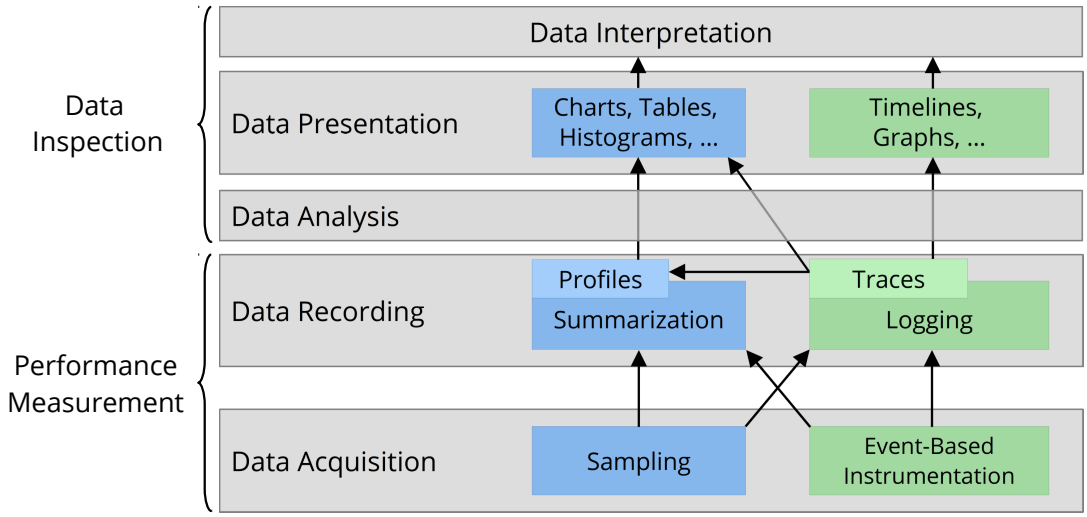


Figure 3.1: The performance analysis process can be divided into separate layers, each comprising several techniques or concepts.

Imbalanced execution between processes or threads can also be caused by resource contention. In HPC systems, the file system and the interconnect network are usually shared resources, which affects parallel input/output (I/O) operations or inter-node communication. The parallel access to the last-level cache of a CPU can cause dynamic imbalances [KBH+08]. Hence, the same operation can take a different amount of time, depending on the state of the shared resource.

The oversubscription of computing resources can also result in waiting times. For example, more threads can be spawned than CPU cores are available. An offloading device can be used as shared resource between multiple processes or threads. Sharing a resource can increase its utilization, but potentially introduces imbalances in the execution.

When imbalances are not compensated, they might force some processes to wait before they synchronize with each other. This results in so called wait states (see Section 3.3.4). As small imbalances can add up to considerable waiting times at scale, they are an important target for performance optimization.

3.2 Performance Analysis Layers

According to Jain [Jai91] a system monitor for performance analysis is composed of several layers, which are shown in Figure 3.1. Juckeland [Juc12] adopted these layers for application performance analysis, but distinguishes only between data acquisition, data recording and data presentation. However, the main contribution of this work is in the data analysis and data interpretation layer.

The first essential step in the analysis process is data acquisition. Independent of the method that generates data, they are recorded and typically made persistent as profiles or traces in a subsequent step. Whereas a profile contains already summarized data, traces can always be aggregated to profiles. The next step is the data analysis. To keep the impact on the program execution as low as possible, time-consuming and sophisticated or complex data analysis is typically performed after recording data. Simple analysis such as aggregation or counting of events can already be done during data acquisition, which can reduce the amount of data that has to be stored. The data presentation layer covers all types of visualization of profiles and traces as basis for the result interpretation, which is usually done by a performance analyst or the program developer. Automated data interpretation can also be performed directly after the data analysis.

3.2.1 Data Acquisition

The data acquisition layer extracts data from a program execution run. It generates the prerequisite for all other analysis layers and has to ensure that required information for complex analysis techniques, such as the detection of the critical path, is acquired. Hence, whether a data acquisition method is suitable or not depends among others on the intended analysis and the requested granularity of the gathered data. Another requirement for reasonable performance measurement is low overhead in terms of a tolerable program distortion.

There are two fundamental data acquisition methods: instrumentation and sampling. Metz et al. [MLG05] discussed these techniques and proposed a hybrid approach to combine the best of both worlds. A short summary of both approaches, individual advantages and drawbacks as well as their suitability for post-processing analysis are discussed in the following.

Instrumentation

Instrumentation inserts additional instructions or hooks in a program, which work as event triggers during the code execution. These events are typically handled by a measurement library which provides accurate timings and generates a profile or event log. For the user of a performance tool the complexity of instrumentation is typically hidden, e.g. the Score-P compiler wrapper uses switches to configure available instrumentation options [Sco18].

There are several possibilities to instrument a program. On the source-code level, instrumentation can be added manually by the programmer or automatically by a source-to-source transformation tool or a compiler. **Manual instrumentation** is done by inserting calls to event-handler routines in the source code, usually with predefined macros that are provided in a header file by the measurement system [Sco18]. Source-to-source transformation tools such as OPARI2 [MMSW02], which instruments OpenMP constructs, the Rose compiler infrastructure [LQPdS10], and the Program Database Toolkit (PDT) [LCM+00], enable a tool-aided instrumentation.

Compiler instrumentation, e.g. with the flag `-finstrument-functions` of the GNU Compiler Collection (GCC) or the flag `-tcollect` of the Intel C/C++ compiler, generates hooks for entry and exit to each function that is not explicitly excluded. Such calls provide the call site (location where the function is called) and the address of the instrumented function, which can be looked up in the symbol table. Compiler and manual instrumentation require recompilation of the application's source code.

A compiled program can be manipulated with **binary instrumentation**. This can be done statically to generate a persistent modified binary or dynamically at runtime without making permanent modifications to the executable. Static binary instrumentation, e.g. with PEBIL [LTCS10], usually introduces less overhead at program runtime than dynamic binary instrumentation, which performs additional tasks such as parsing, disassembly, and code generation at runtime. Both approaches perform the following steps at instrumentation points: save the program state, branch into the instrumentation code, restore the program state, and return control to the application (branch back to the program code). Dynamic binary instrumentation as provided with Dyninst [BH00] uses a mutator process, which uses operating system calls such as *ptrace* to control process execution, to branch into the instrumentation code and back as well as to read and write the address space of the application program. Binary instrumentation is platform-dependent, as it is based on the ISA.

Instrumentation by **library wrapping** adds an interception layer between a program and a library. The interception layer is typically a library by itself and implements functions with the same signatures as the original library API. These wrapper functions call the *real* function, but also trigger an event before and after the call, which can be handled by a measurement library. It is not needed to recompile the application, but the linking has to be manipulated in a way that the application calls the wrapper functions which themselves have to call the *real* functions. Another benefit of this method is the ability to access and manipulated the arguments of function calls, which allows a tool to acquire more runtime details or even change the program behavior. In contrast to binary instrumentation, it does not rely on the

symbols that are used in a specific implementation. Library wrapping as a basis for performance analysis of accelerators has been described in [DIJ10] and is implemented in the measurement tool VampirTrace, which is able to generate wrappers for arbitrary libraries based on the respective header files. Score-P, the successor of VampirTrace, uses library wrapping to measure OpenCL operations [DT15].

A main drawback of all instrumentation approaches is that the overhead is hard to predict. It depends on the number of generated events, which is typically unknown before an instrumented program is executed. Typically, the runtime overhead increases linearly with the number of triggered events, as well as the memory needed to log all events. A setup execution of the instrumented program that produces a profile can be used to estimate the memory needed to write a full trace. A main advantage of instrumentation is that it guarantees to trigger all occurrences of a certain type of events, which is important for parts of the contribution of this work. For example, library interception can guarantee that all calls to a certain library routine are gathered. The requirement to reliably gather certain operations for the analysis is discussed in detail in Section 5.1.

In the context of this work, it is reasonable to distinguish between instrumentation of arbitrary routines, code blocks or regions, and instrumentation of programming APIs. Source code and binary instrumentation are very flexible and can be used for both. However, to instrument directive-based programming APIs, respective tools have to be constantly updated with every change in the API or specification, which is costly and often not or only far later realized, e.g. OPARI2 [MMSW02] for OpenMP instrumentation.

Sampling

Sampling (or probing) is a data acquisition method that periodically interrupts the program execution to obtain state information. It can be applied on the executable without the need for recompilation or relinking. One option is the use of time-based interrupt generators with a regular sampling frequency, which is independent of the event (e.g. function enter) frequency. Other interrupt generators are for example hardware counter overflows for e.g. cache misses or (number of) executed instructions. The sampling frequency controls the trade-off between measurement accuracy and measurement overhead.

Sampling generates only statistical results on the program execution without being accurate in terms of counts or timings. Therefore, it is often referred to as statistical sampling. “If an event does not occur in a sampling log, there is no guarantee that it did not occur in execution.”[MLG05] Hence, sampling is not feasible to analyze event causality.

As the sampling overhead rather depends on the sampling frequency than the event frequency, it can be beneficial for the analysis of codes with many short-running routines, where full instrumentation would introduce a large overhead and perturb the program execution. However, sampling requires stack unwinding to generate a call-path profile or trace, which introduces an overhead depending on the call-stack depth.

Tool Interfaces

The most convenient way to acquire data for a given programming API is a standardized tool interface that is available in respective implementations. Conceptionally, tool interfaces belong to the data acquisition type instrumentation, whereby instrumentation is done by library or runtime implementors instead of performance tool developers. Additionally, such interfaces can provide state-tracking capabilities to enable a reasonable sampling-based performance analysis.

In contrast to classical instrumentation, a tool interface evolves with the specification of the standard, which moves instrumentation maintenance effort from tool developers to runtime and library developers. Furthermore, tool interfaces can export performance information that are not accessible with other approaches, e.g. library or runtime internal operations and states. A comparison between instrumentation of the OpenMP source code and the OpenMP Tools (OMPT) interface implementation in an OpenMP runtime has been investigated in [LDTW14]. The MPI standard [Mes15] defines the MPI profiling in-

terface (PMPI), which enables tools to intercept MPI calls, and the MPI tool information interface to inspect and manipulate MPI control and performance variables. Since version 2.5, the OpenACC standard specifies an interface for event-based performance analysis. NVIDIA provides the CUDA Profiling Tools Interface (CUPTI) [NVI18b] to gather performance data from CUDA programs.

Complex analysis techniques typically have strict requirements on the performance data. Similar to instrumentation, an available implementation of a tool interface with event callbacks guarantees that registered events are triggered. The extra value of tool interfaces for OpenACC and OpenMP are discussed in Section 4.2.

3.2.2 Data Recording

Independently of the acquisition method, data can be collected and recorded in an aggregated form or in its temporal order. The aggregated form is called a profile, whereas temporally ordered data is stored as a trace. The terms profiling and tracing refer to the process of generating a profile or trace and do not warrant a specific data acquisition method or presentation. Profiling is often misleadingly used as a synonym for performance analysis, as for example in the abbreviations PMPI and CUPTI.

The data recording layer also has to handle different clock frequencies and clock skews, if data is acquired from different hardware sources. In heterogeneous systems, device types, e.g. CPU and GPU, usually run on different clock frequencies and use their own hardware timer. Timer synchronization has been discussed e.g. in [DKMN08] with focus on multi-node event traces and in [DJW15] with respect to different timers on host and offloading device.

Profiling

Profiling generates an aggregated form of performance data, which has a much smaller memory footprint than trace data of the same application run. It is often used in combination with sampling, e.g. in Gprof [GKM82] and HPCToolkit [ABF+10]. A performance profile provides summary information on a program execution, which can include, among other things, the invocation count and runtime of program regions as well as hardware counter and communication statistics. There are several types of profiles which differ in its level of detail. Flat profiles provide aggregated data for each program region. Call-graph profiles preserve the calling context of program regions and summarize data into a call tree. Call-path profiles [Hal92] provide even more details as information are stored for each call path individually. It is also possible to mix these profile types, e.g. to generate a profile with focus on a specific region as Score-P allows with its *Dynamic Region Profiling* [Sco18].

Profiles can be stored per software location (process or thread) as parallel profiles, which allows the investigation of load-balancing aspects. Due to the relatively small size of profiles, a large number of them can be stored, e.g. to compare different program runs or build a performance regression database using for example TAUdb [HMBM05].

Tracing

Tracing records an ordered set of events. It is often used in combination with instrumentation to enable detailed and accurate performance analysis. Efficient event-based tracing on large distributed systems requires a minimal runtime overhead and memory footprint per event, accurate time synchronization across compute nodes, and a trace analysis that can handle billions of events [AHLT09].

A trace can be directly printed out to a command line or written to a file. Most trace formats write one trace file for each concurrent execution stream in a parallel program. Among the well-known ones are the Pajé trace file format [dOSdK13], the Open Trace Format [KBB+06], and its successor the Open Trace Format 2 (OTF2) [EWG+12]. For long running applications or high event frequencies, traces can become extremely large and might not fit into the memory that was reserved for event recording.

When the event buffer is full, it has to be flushed and written to permanent storage, which can perturb the program behavior and distorts the subsequent measurement. There are approaches such as OTFX by Wagner et al. [WDK15] to keep the trace in main memory. This is achieved by hierarchical deletion of data when the memory is full. It also introduces data compression features that could be easily integrated in OTF2, from where it has been originally been forked.

Program traces are usually visualized in a timeline view for inspection by the performance analyst or programmer. Whereas the generation of a profile only allows simple data analyses, e.g. aggregation or reduction, at runtime, tracing preserves all events in its temporal order and enables sophisticated analysis techniques to be applied in a post-processing step. Nevertheless, it is always possible to summarize a trace into a profile.

Combined Profiling and Tracing

The hybrid between profiling and tracing is a compromise between low memory footprint and detailed, dynamic runtime data. A technique, commonly referred to phase profiling [MSM05], uses a timely ordered series of profiles to enhance profiling information with the temporal evolution of the program execution. As the resulting number of profiles depends on the number of program phases or iterations, the memory footprint may still be large or even exceed the available buffer size. Clustering of profiles in a time series addresses this drawback [SWW09].

Phase profiling of GPU kernels has been investigated in [DSWB12]. Phases emerge between synchronization operations within a thread block. The execution times of individual threads are summarized to minimum, maximum and average values for a kernel phase, which are stored in their execution order.

Trace profiling by Mohror et al. [MK12] reduces the memory footprint of event traces, based on repeating execution patterns and the similarity in the behavior of processes in parallel programs. Processes with similar behavior are grouped. A process group is split into segments, which are lists of ordered events. Segments have to be marked in the source code, e.g. loop iterations. Whenever a code segments is executed again, it is compared with the representative segment. In case of a match for some similarity metric, only a reference to the representative segment and a timestamp have to be stored. However, the similarity of process traces may not be easily determined or time-consuming [WBB12, WMS+13].

3.2.3 Data Inspection

The inspection of profiles and traces can be divided into three parts: analysis, presentation and interpretation. **Data analysis** supports the inspection by highlighting code regions that might be relevant for program optimization. There are simple and fast analysis operations with low overhead which can be applied during data recording, such as runtime aggregation or counting function invocations. More complex and time-consuming analysis techniques are performed on the recorded data set after the program has been executed. Data analysis can remove information by means of summarization or simple deletion. However, the main target is to generate information by interpreting the program behavior and the semantic of program regions. It is an optional step, but essential for a systematic performance analysis as it is intended to provide guidance for the analyst or program developer. Performance data analysis is a major part in this thesis. The current state-of-the-art is discussed in Section 3.3, while an enhanced generic analysis is discussed in Chapter 5 as contribution of this thesis.

The **data presentation** layer provides a visualization and optionally a user interface to understand and interpret the recorded data. The data presentation as well as interactive visualization tools should be designed as intuitive as possible. Main challenges in visualizing performance data are the complex hierarchy of current HPC systems, the ever increasing parallelism, and the diversity of different events and metrics. Profiles are typically visualized as tables, trees or charts. Traces are most often visualized using timelines or execution graphs. The visual comparison of trace files can also support the application analysis [WBW+17].

Data interpretation evaluates the results of data analyses. This can be partly automated, e.g. based on the top-rated regions in an analysis or with thresholds to determine when a parallel program is unbalanced. The result might be a textual statement that highlights the most runtime-consuming program region or evaluates the programs waiting time in comparison to its computational regions. Automatic data interpretation can be seen as an extended data analysis. Nevertheless, it requires the program analyst to evaluate the generated results. Data interpretation can also be performed manually by an application analyst or developer based on a reasonable visualization from the data presentation layer. This might be additionally necessary to detect all performance-relevant aspects in an application.

This thesis intends to facilitate the interpretation of acquired performance data by introducing new expressive performance indicators that are suitable for the analysis of applications on heterogeneous architectures. For a better guidance to critical parts in a program, the developed analysis framework generates an optimization guidance rating based on these indicators and enables expressive metrics for in-depth program performance visualization a timeline view.

3.3 Performance Data Analysis

In the process of code tuning, the analysis of performance data is an essential step. As basis for the performance evaluation of parallel codes, Section 3.3.1 describes so called performance properties and their subcategorization. Inefficiency patterns in MPI and OpenMP are explained in Section 3.3.2 as they form a requirement for the analysis presented Chapter 5 of this thesis.

Several analysis techniques have been established to detect and quantify performance bottlenecks and expose respective optimization targets. Section 3.3.3 describes hot-spot and hot-path analysis, which might be already performed during data recording, as they are based on simple data aggregation. More complex analysis techniques are usually time-consuming and therefore performed in a post-processing step after data recording. Section 3.3.4 describes wait states, as symptoms of performance problems. Their detection is often the basis for further analyses. Section 3.3.5 and 3.3.6 study two techniques that have been proven to be effective in identifying and quantifying imbalances in parallel codes: critical-path and root-cause analysis. Figure 3.2 shows the relation between common analysis techniques, performance, properties, and presentation opportunities of the analysis results in HPC analysis tools.

3.3.1 Performance Properties

The performance of a parallel program can be characterized using *performance properties*, which comprise performance problems and "non-negative performance aspects such as computation"[WMDM07]. They include performance metrics, inefficiency patterns, and performance indicators. A *performance metric* is a simple measure such as the time, function visits, and hardware counters. An *inefficiency pattern* is an execution pattern with negative impact on the program execution. An execution pattern is a set of events on one or more execution streams that occur in a specific order. *Performance indicators* provide a higher level of abstraction [Böh13]. They are derived from one or more analyses to indicate an execution aspect, e.g. (im)balances.

Most programming paradigms, execution or communication models have inherent inefficiency patterns. An example for inefficient execution is a waiting thread or process in a synchronization operation. Wolf et al. [WMDM07] define performance metrics and inefficiency patterns for a subset of today's functionality of MPI, OpenMP and SHMEM. Inefficiency patterns have been specified for MPI-2 one-sided communication in [KHMW06]. Patterns for inefficient performance behavior on GPUs are investigated in [EBW11]. As standards and programming models have evolved since these publications, the list of patterns is not exhaustive. Inefficiency patterns for MPI and OpenMP are relevant for this thesis and therefore explained in more detail in Section 3.3.2.

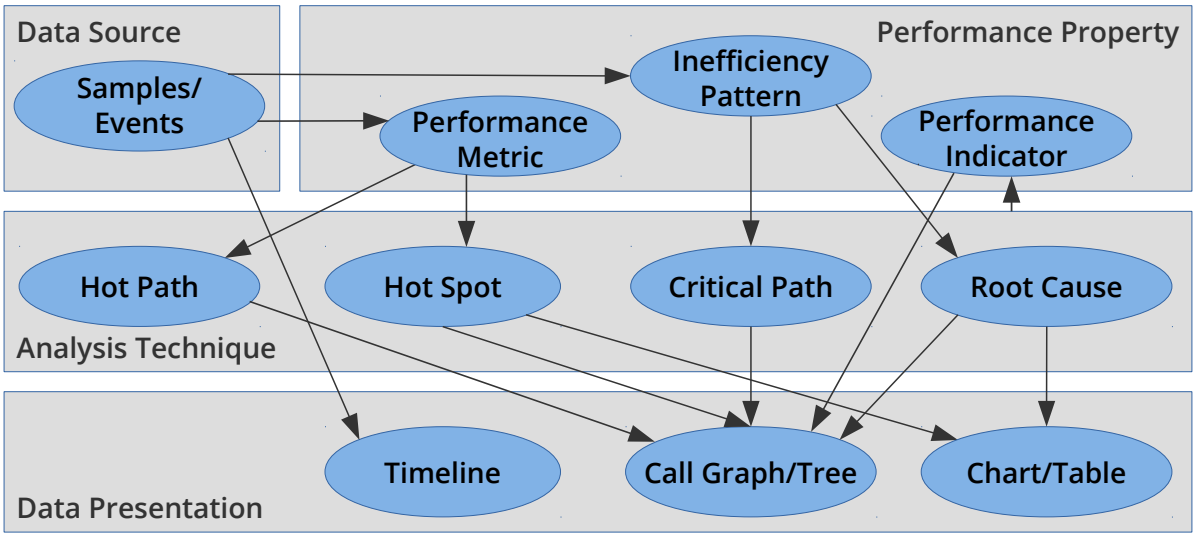


Figure 3.2: Several generic analysis techniques have been established for the performance evaluation of parallel applications. The starting point of the analyses are so-called performance properties. The analysis results are visualized to highlight important execution aspects. Call trees and call graphs are the most popular type of data presentation for all kinds of analyses. Timelines are mostly used for manual analysis.

3.3.2 Inefficiency Patterns in MPI and OpenMP Applications

Inefficiency patterns for blocking and non-blocking MPI point-to-point communication and MPI collectives have already been defined in [Vet00]. The analysis framework, that has been developed as part of this thesis, detects and evaluates these inefficiencies (see Chapter 6).

An inefficiency occurs in MPI blocking send-receive operations when `MPI_Send` and `MPI_Recv` overlap and either of them starts before the counterpart. The parallel detection of the respective *Late Sender* and *Late Receiver* pattern is explained in [GWWM09]. Blocking MPI collectives are inefficient, if the participating processes do not enter the operation at the same time, as this causes some processes to wait for the process that enters last. Figure 3.3 illustrates inefficiency patterns for MPI blocking communication.

Non-blocking MPI communication can produce the same inefficiency patterns, when either of the communication participants triggers a communication, which another process is already waiting for. Hence, long running `MPI_Wait` or `MPI_Waitall` routines are an indicator for these inefficiencies. As described in [Vet00], inefficient non-blocking MPI communication can also be detected with empirical means. An approximation can use the minimal runtime of the `MPI_Wait` routine for both `MPI_Isend` and `MPI_Irecv` separately as reference to expose longer running `MPI_Wait` routines as inefficiency.

According to [HMBW13] the main inefficiency patterns in applications with MPI one-sided communication are *Late Post*, *Early Wait*, and *Wait at Fence*. *Late Post* and *Early Wait* can occur only in *active target communication*, which is similar to non-blocking MPI communication, except that one process provides all arguments of a data transfer. The processes that are involved in the communication have to start and complete an exposure epoch to provide access to their data. A *Late Post* occurs when one or more processes are blocked waiting for the target process to start its corresponding exposure epoch. An *Early Wait* occurs when the target process closes its exposure epoch before the last process completes its access epochs. *Wait at Fence* occurs in *passive target communication* when one or more processes are waiting in an `MPI_Win_Fence` operation before the last process enters, which is similar to blocking MPI collectives. Long waiting times at blocking collective operations, where one or more processes are waiting for the last process to enter the synchronization point, are an indicator for load imbalances.

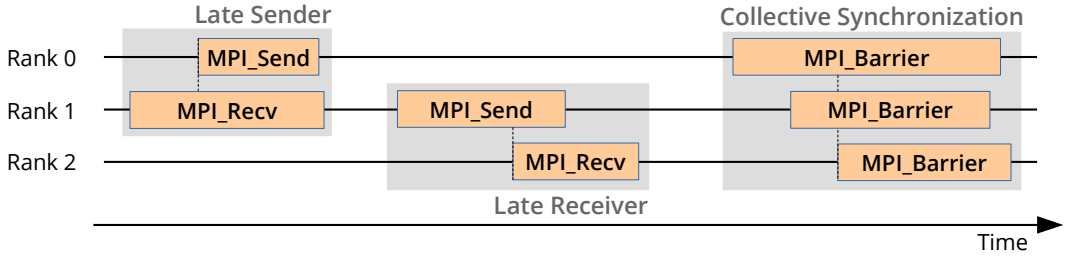


Figure 3.3: The timeline of three MPI ranks illustrates three known inefficiency patterns of MPI blocking communication. In a *Late Sender* pattern, the receive operation starts early and waits for the send operation to start. In the *Late Receiver* pattern, the send operation waits for the receive operation to start. It can occur if the send is implemented synchronously (e.g. `MPI_Ssend`) or the send buffer is full. In a *Collective Synchronization*, such as an `MPI_Barrier` or `MPI_Allgather`, all processes wait for the latest one to arrive.

Inefficient execution in OpenMP applications might occur at barriers, locks, and critical regions. A barrier synchronizes all threads in its enclosing parallel region. Threads that enter the barrier early have to wait. Long waiting times at an OpenMP barrier indicate load imbalances between threads in a parallel region. OpenMP locks and OpenMP critical regions serialize the execution of code regions. An inefficiency occurs when one or more threads are waiting to enter a locked or critical region, which is executed by another thread. Inefficiencies in OpenMP programs can also occur based on other constructs such as *taskwait* or in the context of *cancellation*, which however are less often used in practice.

3.3.3 Hot-Spot and Hot-Path Analysis

The execution time of code regions is one of the most often used metrics to identify regions that dominate the program runtime. Profilers typically provide this metric as aggregated time over all executions of a code region. The visits of code regions are used to generate average values or identify code regions that are executed frequently. An analysis that rates program regions according to their aggregated execution time is also called *hot-spot analysis*. Aggregations, such as building the sum, are very fast operations, which enable tools to perform them already in the data recording layer without introducing significant runtime overhead. The *hot-path analysis* exposes the most runtime-consuming call paths, based on a call-path profile. Hence, it additionally considers the calling context of program regions.

Hardware counters are also an effective means to detect hot spots in the program execution [TJYD10, THW10]. Compared to the sheer execution time of program regions they can be used to evaluate their execution efficiency. Hence, they identify local optimization potential, e.g. underutilized processing units or bad memory access behavior.

Hot-spot and hot-path analysis can be used to detect load imbalances in SPMD programs. For this purpose, performance metrics have to be collected per execution stream. Differences between execution streams, e.g. differences in the execution time of a code region between individual threads, indicate an unbalanced execution. DeRose et al. [DHJ07] defined new metrics to evaluate load imbalances based on a call-graph profile. The *imbalance time* represents an upper bound on the potential saving that could result from perfectly balancing the particular code region. It is defined as the difference between the maximum and the average execution time of a region. The *imbalance percentage* quantifies the severity of an imbalance as a percentage of potentially-wasted resources. Hence, the approach distinguishes between the size and the impact of an imbalance.

Hot-spot and hot-path analysis cannot reliably expose load imbalances, because the underlying profile data do not cover dynamic runtime effects in parallel programs. For example, an OpenMP parallel region might be executed sequentially by different threads due to resource contention, which is typically not intended and hence a potential inefficiency. The analysis of program traces can overcome this issue, as data are not aggregated.

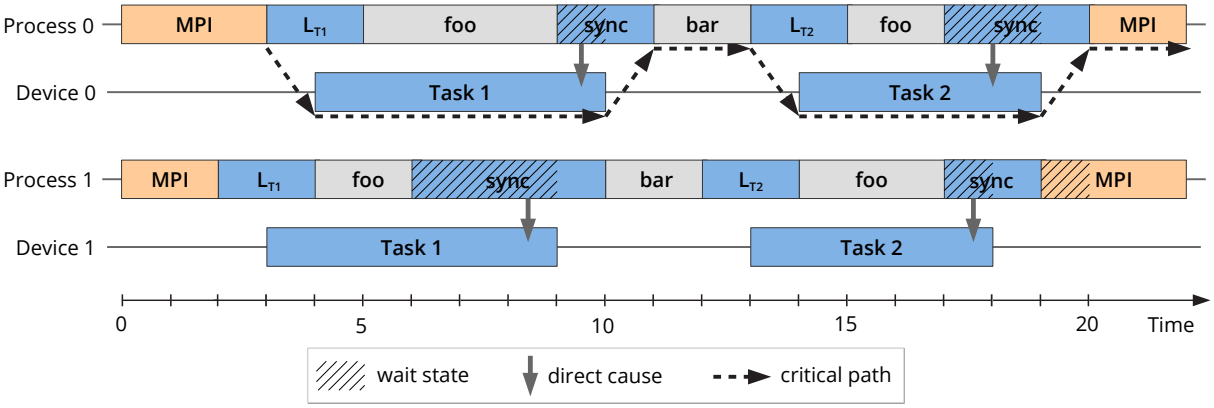


Figure 3.4: Heterogeneous execution scenario with MPI and offloading. Most analysis techniques identify `Task 1` as region with the most runtime impact, as it takes the most execution time, has the largest share of the critical path, and is the direct cause for the most waiting time. However, the optimization of `Task 2` or `bar` has more effect on reducing the total time.

The heterogeneous execution in Figure 3.4 illustrates that hot-spot analysis can fail on detecting the program region with the most potential to reduce the program runtime. The most potential to reduce the runtime have `Task 2` and `bar` with two time units, whereas the most runtime-consuming region, `Task 1`, can reduce the total runtime only by one time unit. In Chapter 5, this thesis presents an analysis, which identifies `Task 2` as most promising optimization candidate.

3.3.4 Wait-State Analysis

Wait states signal inefficiencies in the execution of a program. There are several definitions of wait states in computational science. A generic definition of a wait state is the following: "A situation in which one component of a system is unable to proceed until some other component has completed an operation."[ENC04] In the context of this thesis, the term *wait state* refers to a parallelization wait state according to Definition 4 in Section 5.3.1.

Wait states can be used to detect the critical path, a fundamental runtime characteristic in a parallel program [Sch05]. Previous research on critical-path analysis is presented in detail in Section 3.3.5.

To eliminate wait states in parallel programs, it is reasonable to detect their cause. Different terms have been established for a metric that quantifies the cause of a wait state. It is called *blame* in the terminology of HPCToolkit [ABF+10]. The Scalasca analysis [GWW+10] calls the same metric *cost of idleness*. Techniques to detect and quantify the cause of a wait state are explained in more detail in Section 3.3.6.

Wait-State Detection

A reliable detection of wait states can be performed based on inefficiency patterns. In MPI and OpenMP programs, it basically relies on the comparison of the start timestamps of blocking, joint communication or synchronization operations. A wait state manifests on all processes or threads that enter the joint operation early. Wait-state detection for computation offloading is part of the contribution of this work and explained in Section 4.1.1.

Böhme et al. [BGWA10] describe a scalable technique which enables the detection of wait states in MPI programs. Therefor the analysis uses the same number of processes as the original program and reenacts the program's MPI communication to exchange timestamps. For example, point-to-point MPI communication is replayed twice, as in the original program and in the inverse direction with interchanged sender and receiver. MPI collectives may be replaced with an `MPI_Allgather` to exchange timestamps between processes. The trace analyzer that has been developed as part of this thesis also uses the parallel MPI replay technique.

Waiting Time

Mao et al. [MBH+14] presented an approach to estimate *waiting time* based on profile data without the need to locate inefficiency patterns in a trace or detect blocking operations during the program execution. Their core assumption is that the duration of wait-state-free communication calls is minimal. Therefore, they determine the minimum duration for each call category across the entire execution and assume that every communication in a call category with a longer duration is waiting time. Call categories could be distinguished by the MPI function and its parameters.

The approach has been integrated into Score-P's call-path profiler and is therefore based on instrumentation. Their experiments for selected blocking MPI communication operations have shown that waiting time is overestimated in most cases, in some cases significantly. However, it is still a good indicator for load imbalances which introduces a negligible overhead in the measurement.

3.3.5 Critical-Path Analysis

The analysis of the critical path originates from the domain of planning and scheduling to minimize a project's duration [KW59]. Yang and Miller showed in [YM88] that the critical path is also helpful to guide the programmer to performance problems in a parallel program. For parallel programs, the critical path is defined as follows:

> **Definition 1 (Critical Path)** *In a parallel program, the critical path is the temporally longest execution path that does not contain any wait states. The critical path determines the runtime of the parallel program. If its execution time increases, the total program runtime increases equally.*

The critical-path analysis aims for the runtime reduction of parallel processing. Knowing the critical path enables optimization efforts to be restricted to the relevant parts of the execution, because it includes only the program regions contributing to the overall execution time. However, the critical-path metric itself is meaningful to only a limited extent. For example, it does neither allow a statement about the severity of an unbalanced execution nor does it provide information on the overall runtime benefit when optimizing a specific activity. Already a minimal change of an activity on the critical path can change its course.

Performance metrics derived from the critical path can improve its significance. A contribution of this thesis is a performance metric that is derived from the critical path and overcomes the limitations of a pure critical-path analysis (see Section 5.4.2). In the following, the state-of-the-art in critical-path detection and enhanced critical-path-based analysis techniques are described.

Critical-Path Detection

There are several techniques to detect the critical path of a program execution. Most of them require to build a DAG that models the precedence relations between program activities. The DAG is constructed from a set of runtime events or a complete execution trace. Hendriks and Vaandrager [HV12] describe the construction of the critical path from execution traces. They use a rephrased version of the critical-path method in [KW59] on a task graph. The critical-path method is an algorithm to compute the so-called float (or slack value) of each task, which is the time a task can be extended without increasing the critical-path duration. Then, a task is critical if its float is 0. Single-source shortest-path algorithms, e.g. by Dijkstra [Dij59] or by Thorup [Tho99], are an alternative to compute the critical path in a DAG. They can be directly applied on a task graph, in which all wait states have been assigned an infinite weight.

In the context of performance tuning, critical-path analysis has been extensively discussed for MPI. Schulz [Sch05] describes a technique which uses a wrapper library to manipulate MPI communication calls at program runtime. Timestamps are attached in send operations and evaluated in receive operations. Each process builds a local graph at program runtime and only stores edges that are potentially part of the global critical path. The final critical-path detection is performed sequentially via backtracking the edges of the global graph, which is merged from local graphs in a post-processing step. As collective

communication is treated as a set of individual point-to-point messages, this approach might slow down the program significantly.

Böhme et al. extended Scalasca's parallel MPI communication replay technique [BGWA10] to additionally obtain the critical path from an MPI execution trace [BWdS+12]. Therefor Scalasca reenacts the MPI communication of the program in temporally reverse order to identify the path without wait states (compare Section 3.3.4). The time a program region executes on the critical path is stored as additional metric in a call-path profile (critical-path profile). The replay of MPI communication can be implemented as parallel program, which scales across processes similarly to the program under investigation. Other graph-based approaches are sequentially or do not scale well.

Based on a set of critical-path candidates, a critical-path profile can also be determine by simply attaching additional information to send operations in message-passing applications [Hol96]. Each receive operation compares its longest path length with the one that has been sent from another process and stores the longest path along with the time the candidates spent on the currently longest path. (Remember that wait states do not contribute to the path length.) At the program end, the candidates' critical-path times from the longest path constitute the critical-path profile. This approach enables the detection of a partial critical-path profile at program runtime. However, it does neither expose the temporal critical-path execution nor a complete critical-path profile. Furthermore, it strongly depends on a reasonable detection of critical-path candidates, which might be chosen from a previously generated runtime profile.

Near-Critical Path Analysis

The benefit of optimizing regions on the critical path may provide little overall performance improvement if the second, third, etc., longest paths have a similar duration and are in large parts disjunct from the primary path. In this case, a new path with different activities becomes the limiting factor after the primary path has been optimized. Alexander et al. describe in [ARH94] the benefit of investigating near-critical paths to determine an activity's potential to reduce the total program runtime. There are algorithms that compute all near-critical paths whose duration is within a certain percentage of the critical path, all k near-critical paths, or near-critical paths until either of the two criteria is fulfilled. The proposed algorithms are based on best-first search (BFS) or use a modified version of a shortest-path algorithm. Both require global searches across the entire graph and therefore are not scalable. In many scientific simulations, and especially SPMD programs, near-critical paths vary little from the critical path.

Performance Indicators based on the Critical Path

Böhme et al. showed in [BWdS+12] that performance indicators based on the critical path are useful to detect load-imbalances in both SPMD and MPMD programs. They present a scalable technique to calculate such indicators for MPI execution traces, which has been implemented in the performance analysis tool Scalasca [GWW+10].

The critical-path imbalance indicator is defined as the difference between an activity's duration on the critical path and its average duration over all processes, or zero if the difference was a negative value. It relies on the assumption that all processes execute the same or at least a similar mix of activities. Thus, it is only suitable for SPMD programs. Figure 3.4 illustrates another weak spot where offloading is used in addition to MPI. In this specific execution scenario the imbalance indicator is 0 for all activities.

The performance impact indicator focuses on the evaluation of load balancing in MPMD programs and combine the critical-path profile with costs of wait states. It basically maps the time spent in wait states as imbalance costs onto activities on the critical path. The total performance impact is then defined as the total runtime of an activity across all processes and the imbalance costs it caused. To address MPMD programs in particular, imbalances are categorized in inter- and intra-partition, where a partition represents a program with similar call paths.

The performance impact indicator tries to identify the program activities on the critical path that caused the most imbalance and hence have the most impact on improving the program performance. As the

imbalance costs are not necessarily mapped to their direct cause, the computation of these indicators is only a heuristic to provide an upper bound of the achievable performance improvement. This is because the critical-path profile does not provide a chronological order to correctly assign activities with the imbalance costs. Even though critical-path activities and imbalance costs were assigned exactly, the performance impact indicator can only detect the waiting time a critical-path activity caused and not its performance impact in terms of total program runtime. Considering the MPI/offloading example in Figure 3.4, the imbalance costs would reflect the accumulated waiting time. Hence, the performance impact indicator would identify `Task 1` as the activity with the largest performance impact. A contribution of this thesis is a similar performance indicator that evaluates critical-path activities according to their optimization impact on the global program execution (see Section 5.4.2).

3.3.6 Analyzing the Cause of Wait States

Resource contention and load imbalances are most often the cause of wait states in SPMD-parallel applications. MPMD-parallel applications can additionally have imbalances between the programs. Wait states manifest themselves at the synchronization point that is following a delayed execution. Hence, the actual cause of a wait state is called *delay*.

The increasing parallelism in HPC applications and long execution runs allow a potentially large temporal and spatial distance between the cause and its symptom — the wait state. As this is hard to track manually, analysis techniques that analyze the cause of wait states have been developed. Depending on the research group different terms are used. Meira et al. [MLA98] use the terms cause-effect or waiting-time analysis. Böhme et al. [BGWA10] call it root-cause or delay analysis. Tallent et al. [TMCP10] developed a technique that they call blame shifting.

Trace-based Approaches

The *cause-effect analysis* (CEA) by Meira et al. [MLA98] analyzes the cause of wait states in distributed programs based on detailed execution traces. Initially, a weighted causality graph is built from the program trace, which is used as input for the analysis. Edges represent tasks or program regions. They are weighted with a duration, which allows an analysis to quantify wait states and their cause. The actual CEA traverses the graph between two consecutive synchronization points of two processes and compares their execution paths. Differences are identified as the cause of the subsequent synchronization delay. If more than two processes are involved in the synchronization (e.g. MPI collectives), the synchronization point is split into sets of process pairs which are individually analyzed [Mei97].

Scalasca's *root-cause analysis* [BGWA10] is based on the work by Meira et al. [MLA98]. It refines their CEA in several ways and introduces a new terminology. Wait states are distinguished into direct and indirect wait states as well as terminating and propagating wait states. A *direct wait state* is caused by an activity that does not include a wait state. An *indirect wait state* is caused by another wait state, which is therefore called a *propagating wait state*. A *terminal wait state* is at the end of the causation chain and does not propagate any further. *Delay costs* are defined as the waiting time that is caused by a delay. Finally, the delay analysis computes the *short-term costs*, which cover direct wait states, and the *long-term costs*, which cover indirect wait states.

Technically, the delay analysis by Boehme et al. [BGWA10] is performed in two stages, a parallel forward and a parallel backward replay of MPI communication. During a forward stage, wait states are identified and quantified according to their duration. Synchronizing communication is also annotated with the involved remote ranks. During a backward stage, the delays are identified for all wait states. This stage also classifies the wait states. As in [MLA98] the algorithm works on synchronization intervals. Using the backward replay, the costs are propagated back to their source similar to propagating waiting time in the forward stage. Long-term costs are only relevant for complex blocking point-to-point communication patterns, which is an unusual case in optimized parallel applications. However, the example code Zeus-MP/2 was characterized with complex non-blocking MPI point-to-point patterns and the delay analysis exposed a large amount of long-term costs.

Sampling-based Approaches

In contrast to the previous analysis approaches, *blame shifting* techniques are developed to work with sampling. Hence, their accuracy is highly dependent on the sampling rate. HPCToolkit [ABF+10] was used to implement the following strategies.

The term blame shifting has been introduced by Tallent et al [TMCP10] in the context of lock contention analysis of multithreaded applications. It basically blames lock holders for the waiting time that occurs on other threads during lock acquisition. The respective implementation uses *atomic add* to increase the lock's waiting time counter and *atomic swap* to blame the lock holder (read and reset counter). For every active lock a respective counter is needed. Based on an instrumented GNU OpenMP runtime Liu et al. [LMCF13] extended the blame shifting technique to support the analysis of OpenMP programs. They distinguish between directed blaming of lock holders and undirected blaming of busy threads for causing idleness on other threads. The concept supports OpenMP locks and barriers. The implementation supports OpenMP tasks in general, but it does not evaluate constructs such as *taskwait*. Blame shifting for multithreaded applications is done at runtime, instantaneously, when a thread takes a sample.

Blame Shifting on GPUs

Blame shifting for CPU-GPU programs has been introduced by Chabbi et al. [CMFMC13] and implemented in a special version of HPCToolkit, "G-HPCToolkit". The principle of blaming the non-idle resource is maintained, similar to the undirected blaming of OpenMP threads. In CPU-GPU programs, the resource or perpetrator of idleness can be either GPU kernels or code regions that run on the CPU. Symptoms of idleness are for example the CPU waiting at `cudaDeviceSynchronize` or an idle GPU. Blame is quantified according to the idle or waiting time and attributed to full call paths at program runtime.

As GPUs do not support traditional call-stack-based sampling, a hybrid approach with sampling on the CPU and selected instrumentation of GPU operations is applied. It uses device events in combination with event query routines to obtain the device execution state on the host thread. Therefore, the approach is restricted to offloading APIs that provide this functionality, such as CUDA and OpenCL.

G-HPCToolkit instruments CUDA kernels with CUDA events and wraps all CUDA stream management and CUDA synchronization routines. CUDA events are used to query for kernel execution on the GPU and potentially blame the CPU for not keeping the GPU busy. Blame for the GPU is computed using timestamps that are acquired by wrapping CUDA synchronization routines. GPU kernels are blamed at the end of a CUDA synchronization and therefore temporarily stored in a list. CUDA events of completed CUDA kernels are used to get their execution time and distribute the blame accordingly. Measuring the duration of CUDA kernels via CUDA events, as performed in this approach, introduces more overhead and is less accurate than kernel measurement via the CUPTI activity API. However, according to [DIJ10], it provides acceptable accuracy in most cases.

Blame Shifting on Call-Path Profiles

Tallent et al. [TAMC10] developed a post-mortem blame shifting to identify load imbalances in call-path profiles of SPMD executions. At runtime, each thread collects profile information in a calling context tree (CCT), which is a fully context-sensitive call graph, where each call stack of a function has a separate node. The thread-local CCTs are summarized in a post-processing step into a global CCT, for which balance points are identified. A node (code region) is balanced, if every instance takes the same amount of time. To tolerate measurement noise, the root node is defined as the most-balanced node and other CCT nodes are compared to it. Waiting routines are identified by name. As the CCT does not store temporal information, it is not generally possible to identify the precise cause of idleness, which is why blame can only be shifted to possible suspects. Blame for idleness is shifted to its deepest balanced ancestor node, which can by definition not contribute to imbalance. The idea is that if a descendant

	Programming APIs					Analysis Techniques				Presentation	
	MPI	OpenMP	OpenACC	CUDA	OpenCL	Hot Spot	Wait State	Cause Effect	Critical Path	Profile	Time-line
HPCToolkit	✓	✓	✗	✓	✗	✓	✓	✓	✗	✓	✓
Cray Tools	✓	✓	✗	✓	✗	✓	✓	✗	✗	✓	✓
TAU	✓	✓	✗	✓	✓	✓	✗	✗	✗	✓	(✓)
OSS	✓	✓	✗	✓	✓	✓	✗	✗	✗	✓	(✓)
CEPBA Tools	✓	✓	✗	✓	✓	✓	✗	✗	✓	✓	✓
Vampir	✓	✓	✓	✓	✓	✓	✗	✗	✗	✓	✓
CUBE	✓	✓	✓	✓	✓	✓	✗	✗	✗	✓	✗
Scalasca	✓	✓	✗	✗	✗	✗	✓	✓	✓	✓	✗

Table 3.1: Features of performance analysis tools for HPC applications. Mature functionality of MPI and OpenMP as well as hot-spot analysis are supported by most tools. Offloading APIs are often only supported in basic (statistical) analyses. The visualization of the analysis results in a timeline view is missing for the more complex analysis techniques.

of a balanced node contains idleness, that idleness must be caused by one or more descendants of that balanced node. Compared to blame shifting at runtime, the post-mortem approach does not introduce measurement overhead at all. The main drawback is that it cannot directly identify the perpetrators of imbalances but only suspects, which makes it a heuristic approach.

3.4 Related Performance Analysis Tools

The performance tool landscape comprises a large number of software tools. This section presents state-of-the-art performance analysis tools that are related to this work and implement various concepts for a systematic performance analysis. As most HPC applications, e.g. large scientific simulations, can be executed process-parallel and run on a variety of HPC system architectures, the focus is on scalable and portable tools. Neither the list of software tools that are presented in this section nor their description is intended to be complete. The objective is to provide an overview on a representative set of performance tools in HPC, their analysis approaches and result presentations as well as the support for programming paradigms that are used in HPC.

The techniques developed in this thesis extends the Score-P performance tool suite, a well-known and widely-used set of performance tools with a common measurement infrastructure. Tools such as Scalasca and HPCToolkit implement similar analysis techniques as used in this thesis. This section also gives a short overview on known tools from vendors that develop many-core technology or build heterogeneous HPC systems, because they usually provide a more detailed insight into the execution on the vendor's hardware. The performance tools Carnival and IPS-2 provide early implementations of sophisticated analysis techniques such as critical-path and cause-effect analysis for program tuning. However, they have not been further developed, and hence, do not support recent programming models.

Table 3.1 gives an overview on selected state-of-the-art performance analysis tools. It characterizes tools according to three categories: support for programming models, applied analysis techniques, and presentation of the analysis results. Mature functionality in the dominating HPC paradigms MPI and OpenMP are supported by all available tools. Recent features such as one-sided communication in MPI 3.0 or OpenMP device directives are usually not implemented. Tools that run simple aggregation techniques tend to support a wide range of programming models. More complex techniques that analyze execution inefficiencies are often developed and implemented only for a single paradigm, most commonly only for MPI. This significantly restricts their usability and validity for multi-paradigm applications.

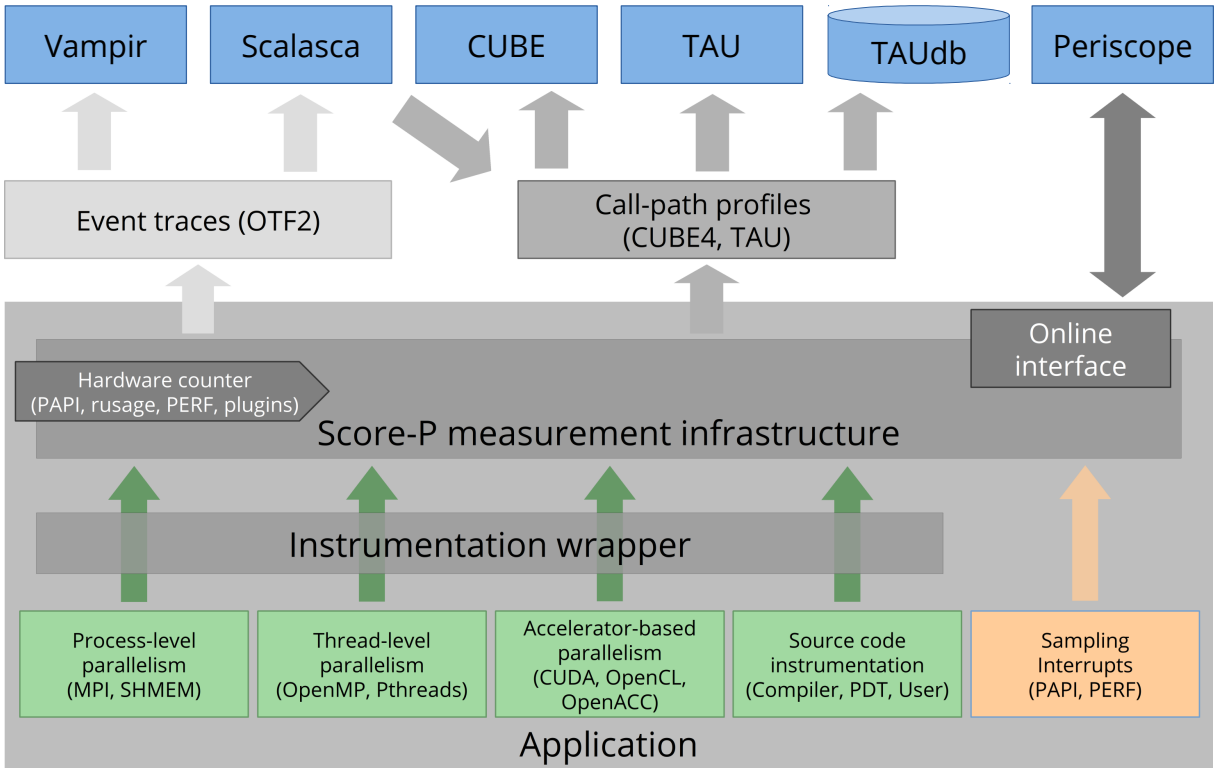


Figure 3.5: The Score-P performance tools comprise a measurement infrastructure as well as several utilities, visualization and analysis tools. Adapted from Score-P user manual[Sco18].

This thesis contributes by specifying inefficiency patterns and respective detection rules for offloading programming models. Furthermore, it enhances available analysis techniques and implements them for the combined usage of the most commonly used programming APIs in HPC (MPI, OpenMP, CUDA, OpenCL, and OpenACC) [MP14]. Besides a profile, the analysis results are added to the original trace file, which makes them available for a detailed time-line view.

3.4.1 Score-P Performance Tools

Score-P [MBB+12] is a community effort on a common infrastructure for performance measurement. It supports a number of analysis tools such as Periscope [BPG10], Scalasca [GWW+10], Vampir [KBD+08], and Tau [SM06]. Figure 3.5 depicts the Score-P measurement infrastructure and its analysis tools.

Score-P

The Scalable Performance Measurement Infrastructure for Parallel Codes (Score-P) is a highly scalable tool suite for profiling, tracing, and online analysis of parallel programs. It is shipped together with the OTF2 [EWG+12], the Cube4 [SKVM15] reader and writer libraries, and the OPARI2 instrumenter. Among the currently supported programming APIs are MPI, OpenSHMEM, OpenMP, OmpSs, HMPP, Pthreads, CUDA, OpenCL, and OpenACC. According performance events are acquired via instrumentation wrappers or respective tool interfaces. The measurement of the offloading APIs CUDA and OpenCL has been extended in the scope of this thesis, whereas support for OpenACC has been developed and implemented from scratch (compare Section 6.2).

Score-P supports compiler instrumentation and sampling to acquire data on a function level. A more fine-grained data acquisition is available via PDT [LCM+00] or manual user instrumentation. Hardware counter are measured based on PAPI, PERF and rusage metrics. MPI library calls are intercepted via PMPI, which provides all required information to rebuild MPI communication dependencies.

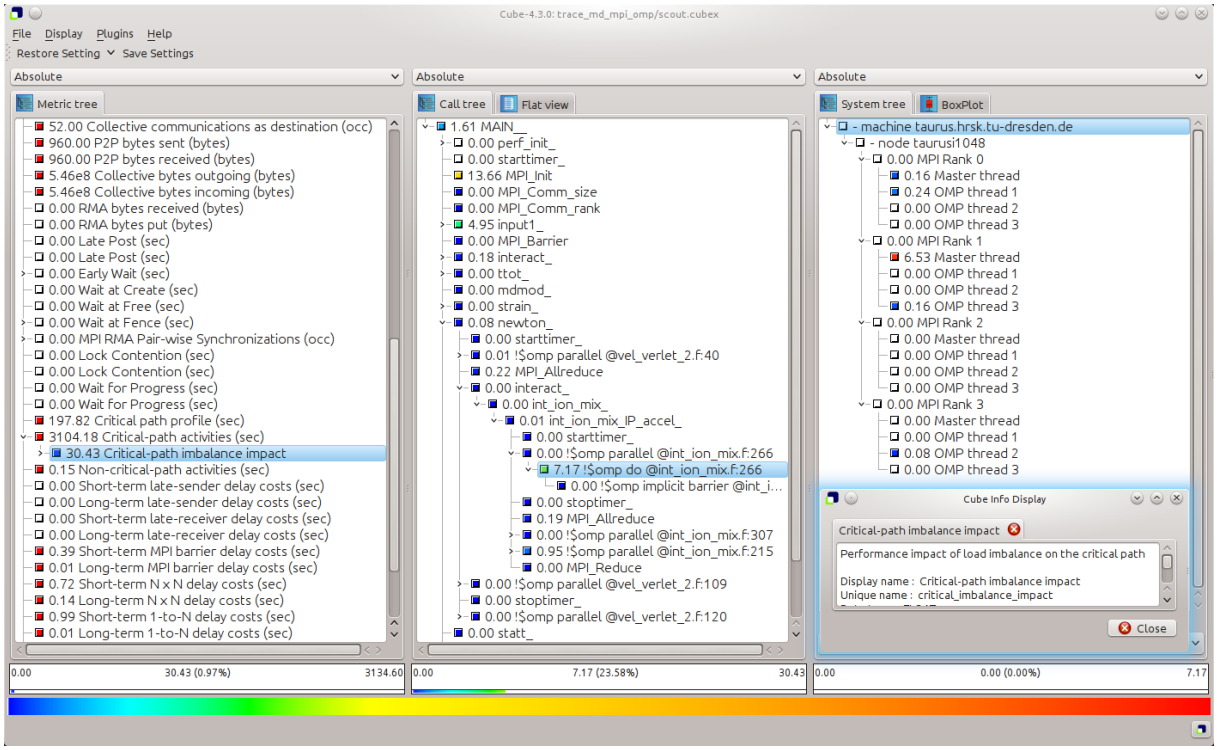


Figure 3.6: Visualization of call-path profiles using the Cube GUI. Metrics are selected in the left panel. Program regions are presented as a call tree in the middle. The right panel shows the distribution of the selection (metric and region) across the system hierarchy. Metric values are presented as numbers and additionally highlighted in a color coding.

As the OpenMP standard (up to version 4.5) does not specify a tools interface, Score-P uses OPARI2 to instrument OpenMP directives and library calls. OPARI2 is the second generation of the source-to-source instrumentation tool OPARI (OpenMP Pragma and Region Instrumenter) [MMSW02]. It implements the POMP2 monitoring interface, which defines, similar as the OMPT interface (compare Section 4.2.2), events for the begin and the end of an OpenMP construct as well OpenMP API functions.

Score-P implements a call-tree-based profiling on a per-thread basis [Sco18] that is written in the Cube4 profiling format. The Cube graphical user interface (GUI) is used to evaluate the profile data of an application run. As even profiles can get very large at scale, Lorenz et al. [LSW15] developed an approach for Score-P that keeps the profile size small by aggregating thread profile data. Score-P also allows the generation of OTF2 program traces, which can be inspected with the trace browser Vampir or automatically analyzed by Scalasca. The Scalasca analysis results are stored in the Cube4 format. OTF2 traces store data transfers between host and device as remote memory access operations [KDD+13].

Cube

Cube [SKVM15] is a visualization tool for call-path profiles in the Cube4 format. Figure 3.6 shows Cube's profile visualization for a molecular dynamics (MD) code [HHdSSB11], which has been executed with four MPI processes, each using four OpenMP threads. The visualization concept is based on three windows: metric, call tree and system tree. The severity of a selected metric is highlighted in a color-coded fashion. The call tree allows visualizes individual call paths. The system tree shows how a metric for a selected function is spread across the system topology (nodes, processes, threads, etc.), which allows analysis of statistical load-balancing aspects. For example, a (statistically) perfectly balanced function has the same execution time on all execution streams.

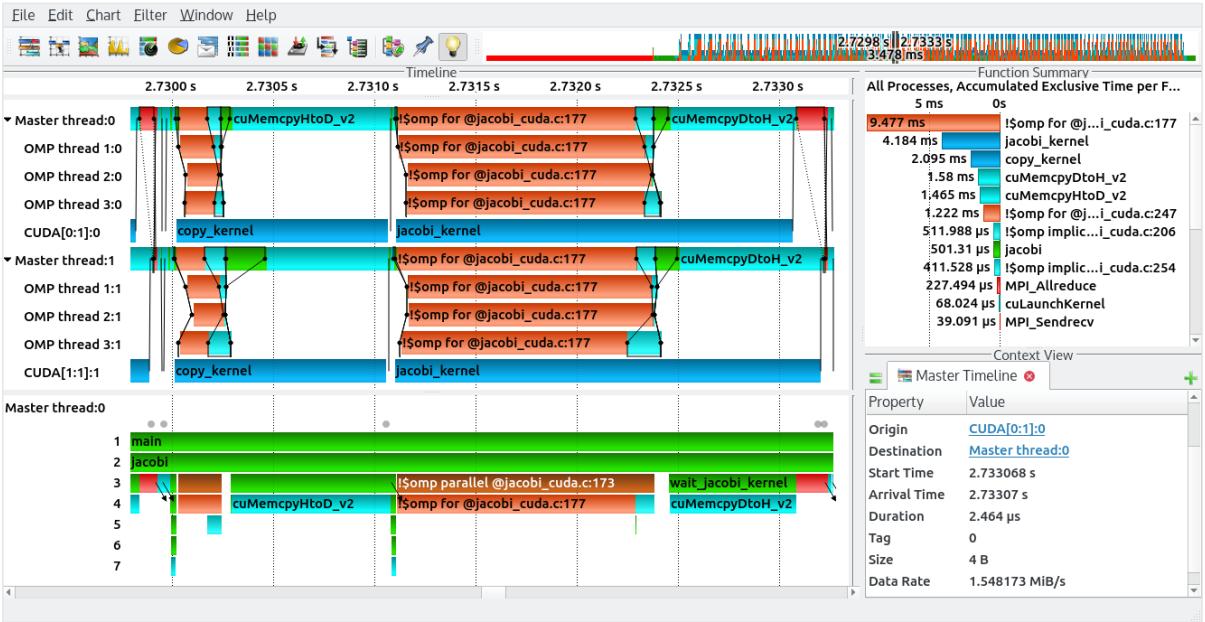


Figure 3.7: Visualization of trace files with the Vampir performance browser. The *Master Timeline* on the top left is the main display. The selected interval shows one Jacobi iteration on two MPI processes, each running four OpenMP threads and one CUDA device. The bottom left timeline shows the call stack for *Master thread:0* (MPI rank 0). A region profile for this time interval is shown on the top right. The bottom right display shows details on a selected communication between an MPI process and a CUDA stream.

Vampir

Vampir [KBD+08] is a visualization tool for OTF and OTF2 trace files. It uses a client front end (GUI) for user interactions and enables scalable processing of large traces with its parallel server backend. The performance data are visualized in various displays, which include function and counter timelines as well as a message communication matrix, summary views, and call stack presentations. Figure 3.7 shows the main display, the *Master Timeline*, together with the *Process Timeline*, the *Function Summary*, and the *Context View*.

Due to its generic visualization approach, Vampir supports most parallelization concepts that are used in HPC application. This includes parallelization with processes, threads and offloading. Among the supported runtime layers are MPI, SHMEM, OpenMP, Pthreads, CUDA, OpenCL, OpenACC and HMPP. A systematic performance analysis of hybrid applications with Score-P and Vampir is presented in [JD17]. The Vampir toolchain for a holistic performance analysis of an MD code that uses MPI, OpenMP, and OpenACC or CUDA simultaneously is shown in [DWW+13]. Vampir focuses on an intuitive graphical representation of the program execution. It provides many useful displays to visually identify inefficiencies and determine their cause. The powerful zooming feature allows performance analysts to get detailed information on time intervals of interest. This thesis uses Vampir to visualize the analysis results such as the critical path, waiting time, and the costs of wait states in a timeline view.

Scalasca

Scalasca [GWW+10] is a performance tool for automatic analysis of OTF2 traces that have been created by Score-P. The analysis results are added to a Cube4 profile and can be visualized in the Cube GUI (see Figure 3.6). Among the Scalasca metrics are a variety of delay costs for individual MPI inefficiency patterns such as later sender, barrier, and other collective operations. Scalasca also determines a set of wait operations and computes critical-path profile metrics. A complete list of analysis metrics is available in the Scalasca documentation [SPP19].

In contrast to its predecessor tools EXPERT [WM03] and the KOJAK toolkit [MW03], Scalasca performs a parallel analysis, based on MPI communication replay to detect wait states, delay costs, and the critical path (see Sections 3.3.4, 3.3.5, and 3.3.6). Due to the fine-grained classification of performance phenomena, expert knowledge is necessary to interpret the results and identify relevant optimization opportunities.

Scalasca supports MPI and OpenMP without the offloading capabilities introduced in version 4.0. Although offloading models such as CUDA, OpenCL or OpenACC are not supported, Xeon Phi programs can be analyzed based on the native or symmetric execution model, where MPI is used on both, the host and the coprocessor [WF13].

Periscope Tuning Framework

The Periscope Tuning Framework (PTF) incorporates performance analysis and automatic performance tuning. It consists of Periscope [BPG10], a performance tool for online analysis, and tuning plugins [CUG15] that optimize specific aspects of the application performance. The tuning model first creates the tuning parameters that define the search space and runs one or more tuning steps. A tuning step is the execution of a program with a specific configuration. It starts with a pre-analysis using Periscope, which identifies tuning alternatives (i.e. different parameter configurations) to reduce the search space. The following search step executes the experiment to determine the effect of the tuning configuration. After all search steps are finished the results are used as input for additional tuning steps or provided to the user.

There are several tuning plugins available so far. Among them are plugins to tune MPI parameters, master/slave patterns, and energy via voltage and frequency scaling as well as a pipeline tuning plugin for programs that deploy heterogeneous many-core architectures [BB14] and a plugin to evaluate the best parameter configuration for HMPP codelets [DBB07]. The goal of most of these plugins is to identify the combination of adjustable parameters that enable the fastest or most efficient execution. To avoid a complete brute-force approach, performance analysis is used to guide the search and evaluate the tuning effect. The Periscope analysis uses a set of agents that autonomously search for predefined performance properties during the program execution. Similar to Scalasca, performance properties are defined for MPI and OpenMP. The PTF workflow is setup via a graphical interface based on Eclipse. Instrumentation and performance measurement is based on the Score-P infrastructure.

TAU

The TAU (Tuning and Analysis Utilities) parallel performance system [SM06] is a toolset for instrumentation, performance measurement, analysis and visualization. It supports several instrumentation approaches and provides a sampling interface to acquire performance data. Furthermore, it can analyze call-path profiles that have been generated with Score-P. The program database toolkit (PDT) [LCM+00] has been developed within the TAU project to support automatic instrumentation at the source code level, e.g. to instrument functions or even code blocks such as loops. Among the supported runtime layers are MPI, several threading models such as OpenMP and Pthreads as well as accelerator paradigms such as CUDA, OpenCL and OpenACC.

TAU is able to generate call-path, call-depth and phase profiles, which can be investigated with ParaProf [BMS03], TAU's parallel profile analysis and visualization tool. ParaProf provides many possibilities to visualize and compare profiles. Outstanding are the three-dimensional views on performance profiles, which enables an additional metric to be correlated with others using a single diagram. TAU does not provide a trace visualization tool, but it enables the translation from its own trace format into other trace file formats to be visualized with Vampir, Jumpshot or Paraver. Another unique feature of TAU is the profile database management framework TAUdb (former PerfDMF [HMBM05]). It can import and store a large number of profiles from different performance measurement tools and provides a convenient way to compare multiple performance analysis experiments.

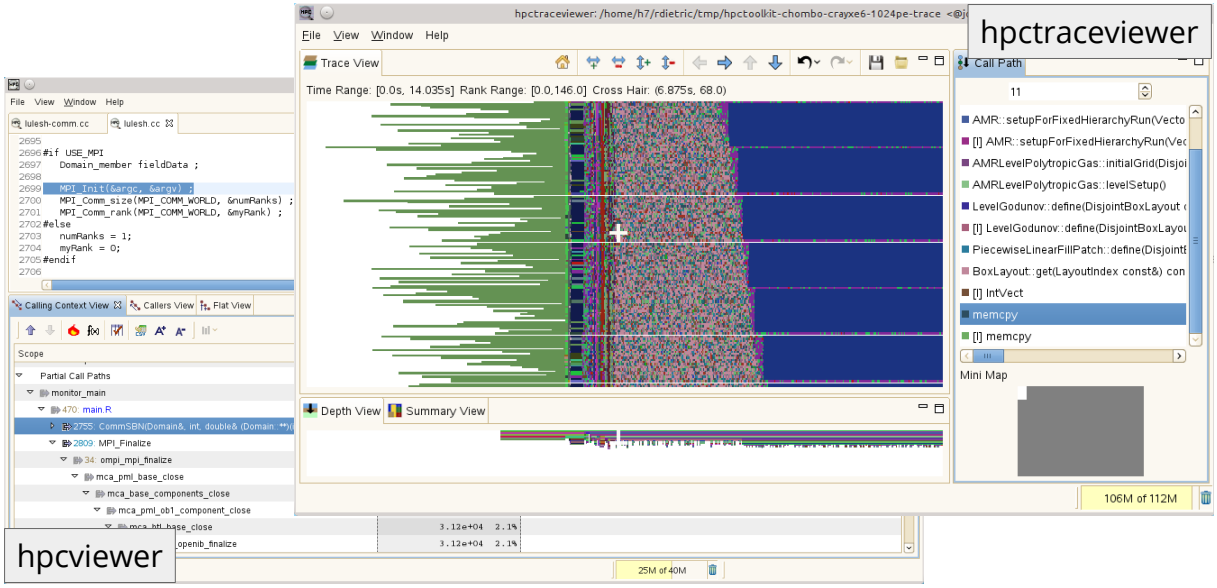


Figure 3.8: Visualization of performance data with HPCToolkit. *hpcviewer* visualizes call-path profiles in a top-down calling-context view, a bottom-up callers view, or a flat view. It enables the computation of derived metrics and a correlation to the program source code. *hpctraceviewer* visualizes traces in a space and time view with color-coded functions.

3.4.2 HPCToolkit

HPCToolkit [ABF+10] is an integrated suite of tools to measure, analyze and visualize performance data. The measurement is based on sampling, which enables the analysis of binaries without the need for instrumentation. Collecting data on the binary level is programming-model agnostic, which basically means that the translation from binary symbols (e.g. function calls) to user-level constructs (e.g., OpenMP directives) has to be done by the user. As described in Section 3.3.6, HPCTookit implements blame shifting at runtime (local load imbalance detection) for CUDA, OpenMP barriers and lock contention, as well as blame shifting on call-path profiles for MPI. It also supports the OMPT interface to query state information for OpenMP threads.

HPCToolkit focuses on the generation of call-path profiles, which are visualized with *hpcviewer*, a GUI that associates measured and derived metrics to full calling contexts to correlate measurements with program structure. Performance data are presented in a top-down view. HPCToolkit traces represent sequences of asynchronous samples for process and threads, which can be visualized with *hpctraceviewer*. A sample contains the entire call stack of active functions. In contrast to Vampir, where functions are grouped and the group is represented with a color, *hpctraceviewer* uses distinct colors for different functions. Figure 3.8 shows HPCToolkit's profile and trace visualization.

3.4.3 CEPBA-Tools

The CEPBA toolkit is a set of performance tools that are developed at the Barcelona Supercomputing Center. The main tools are Extrae [SLH+13] for performance data acquisition and trace generation, Paraver [PPL+95] for trace analysis and visualization, and Dimemas [LGP+96] for performance prediction. Extrae supports established HPC programming models such as MPI, OpenMP, Pthreads, CUDA, and OpenCL. It uses various event-based data acquisition mechanisms, e.g. library interposition, binary instrumentation, compiler instrumentation, and tool interfaces such as CUPTI, but also supports timer and counter-driven sampling.

Paraver is a performance analysis and visualization tool for program traces with similar functionality as Vampir. As the trace format has no semantics itself, Paraver provides a set of semantic modules

and the means to customize the semantics by the user. Although this allows a flexible presentation of arbitrary performance metrics, expert knowledge is necessary to choose appropriate data sources and their respective visualization. Paraver also features analysis tools for clustering to identify the program structure from computational bursts [GGL09], for folding to identify performance phases within repetitive computational regions [SLG+14], and for spectral analysis to detect periodic patterns in program traces [CBL10].

Dimemas is a simulator that predicts an application's behavior on an abstract machine. Based on an event trace that includes timing information about CPU and network operations, it generates a Paraver trace file representing the simulated execution. The abstract machine is modeled by a set of key factors for network and CPU resources. This includes linear factors such as execution and transfer times as well as non-linear factors such as resource contention and synchronization. Dimemas performs critical path analysis to report the importance of code blocks for the program execution time. It supports message passing via MPI and task-based parallelism with OmpSs.

3.4.4 Vendor Tools

HPC system and hardware vendors such as AMD, Cray, Intel, or NVIDIA provide their own performance tools. One fundamental drawback of such tools is the lack of portability, as they typically support only the vendor's products. However, the knowledge of the specific hardware and software stack enables a more detailed performance analysis by accessing and interpreting internal data that is not available to third-party tools.

AMD developed CodeXL to optimize applications for AMD platforms. The tool suite includes capabilities for GPU debugging, GPU and CPU profiling as well as static OpenCL kernel analysis. As AMD CodeXL does not support paradigms such as message passing with MPI or SHMEM it cannot be used to analyze scalable applications. However, to investigate the intra-node performance for AMD GPUs and AMD Accelerated Processing Units (APUs), it provides a unique level of detail. Furthermore, it is one of few tools that support HSA applications.

Allinea MAP is a sampling-based profiler for parallel codes. It supports MPI, OpenMP, Pthreads, CUDA, and several PGAS languages and libraries. Instead of performing any complex analysis techniques, it provides a performance summary, which categorizes the runtime in I/O, compute, thread, and MPI. MAP also highlights the most runtime-consuming code region.

Performance tools by Cray, Intel, and NVIDIA support the analysis of scalable heterogeneous applications and provide features that guide the performance analysis process.

CrayPat & Cray Apprentice

The Cray performance tool infrastructure [DHJ+08] consists of CrayPat and Cray Apprentice. CrayPat provides an instrumentation utility, a data collection library, and a performance report generator. Instrumentation is performed at the binary level using binary rewriting and automatic relinking with `pat_build`. To control the instrumentation the user selects function groups such as MPI, CUDA, or OpenMP. The CrayPat API also enables manual source code instrumentation. Performance data are collected based on sampling or instrumentation and stored as a profile or trace. The report generator `pat_report` reads the performance data and generates a text report. The Cray performance tools support several programming APIs and I/O libraries, e.g. MPI, SHMEM, OpenMP, Pthreads, OpenACC, CUDA, ADIOS, and HDF5. However, no information about offloaded code regions (to accelerators or coprocessors) is collected.

Cray Apprentice is a visualization tool for Cray performance data. It provides several basic views on the collected data. Two views are shown in Figure 3.9, which visualize the performance results of the Lulesh benchmark (default configuration with 30^3 elements) for eight MPI processes, each running four OpenMP threads on two compute nodes of Titan (Oak Ridge National Laboratory). The *Overview* sum-

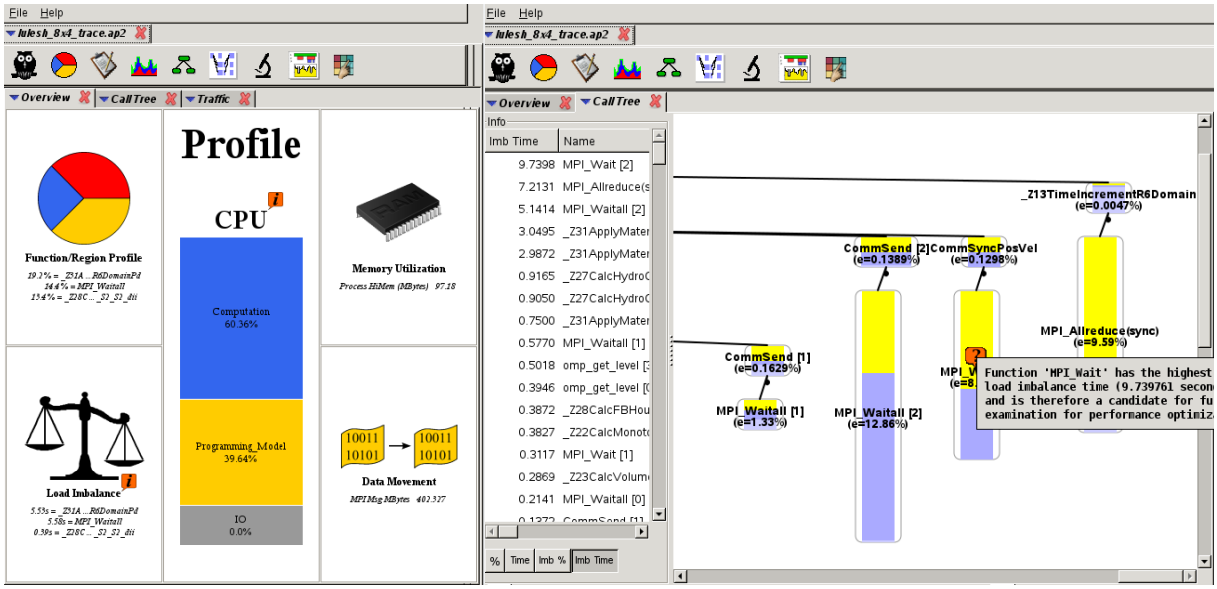


Figure 3.9: Cray Apprentice visualizes performance data generated with CrayPat. The *Overview* (on the left) provides global program statistics as well as a guidance towards performance issues (exclamation marks next to *CPU* and *Load Imbalance*). The *Call Tree* (on the right) visualizes the caller-callee relation and additionally provides runtime and load-balancing information of individual code regions.

marizes the execution scenario and allows the analyst to easily spot the context of a potential performance issue. Other views such as the *Call Tree* or profile charts provide more details on the execution of individual code regions. The *Mosaic* view visualizes data movement between processes similar to the message matrix in Intel's Trace Analyzer and Vampir.

If CrayPat has collected a trace, more detailed information and views are available. The *Call Tree* provides additional load-balancing information and the *Load Balance* view shows the runtime share of regions on each process. The list next to the call tree rates regions according to their imbalance time (compare Section 3.3.3). A question mark highlights the region with the highest load imbalance. The nodes or boxes in the call tree are stacked bar charts, where the lower bar (light blue) represents the shortest execution time, the middle bar (purple) the average time, and the upper bar the longest execution of a region. The height of a box represents the total execution time of the respective region. This node visualization enables the performance analyst to easily spot long running and imbalanced regions. The *Traffic* view presents data movement between processes in a timeline, if event-based tracing has been used to collect the performance data.

Intel Performance Tools

Intel's software stack includes two sophisticated performance analysis tools: the *Intel Trace Analyzer and Collector* and the *Intel VTune Amplifier XE*.

The Intel Trace Analyzer and Collector enables the collection and analysis of traces for MPI programs. The trace visualization is, similar to Vampir, realized with timelines as well as several profile views and a message matrix. Figure 3.10 shows a screen shot for the execution of the Lulesh benchmark [KKN13] (30^3 elements) with eight processes, each running three OpenMP threads. As the Intel Trace Analyzer and Collector is limited to MPI support, OpenMP activity is only observed on the MPI processes. Furthermore, there is no support for accelerators or coprocessors. The analysis detects MPI inefficiency patterns such as *Late Sender* or *Wait at Barrier* and provides a load balancing view, similar to the Cube system tree (compare Figure 3.6). It is also possible to compare traces files.

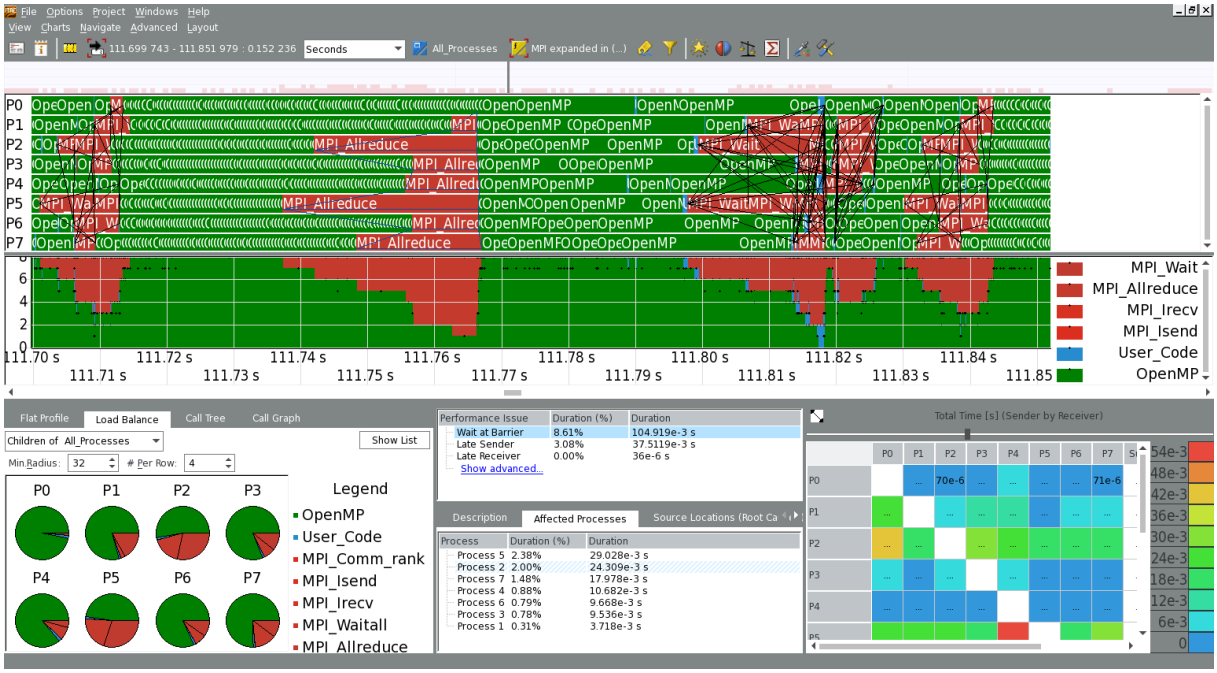


Figure 3.10: The Intel Trace Analyzer and Collector uses several timeline views, such as the *Event Timeline* on the top and the *Quantitative Timeline* below, for a detailed analysis of previously collected program traces. Program regions are color-coded according to their programming API. Black lines between processes visualize MPI messages. The *Load Balance* view (bottom left) exposes the runtime share of regions on each process. The *Performance Issues* view (bottom center) highlights known inefficiency patterns. The message matrix (bottom right) presents the MPI communication behavior with color-coded metric values.

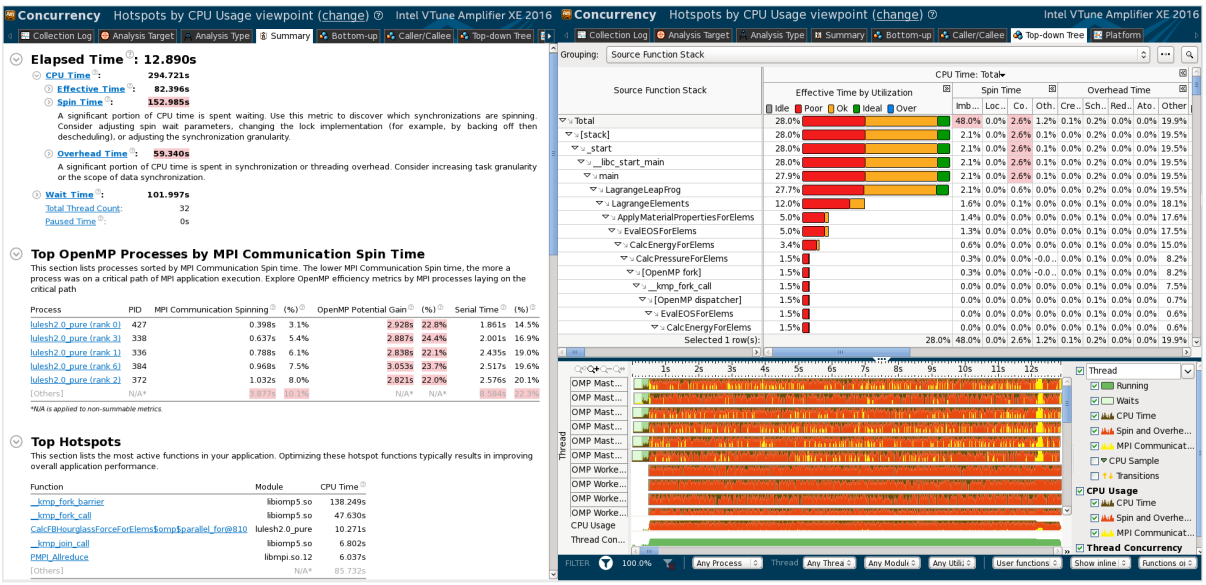


Figure 3.11: The Intel VTune Amplifier XE 2016 has a *Summary* view (left window) to provide an initial overview on the program execution and a first glance on the overall parallel efficiency. The *Top-down Tree* (right window) is used for in-depth analysis of individual functions in the call stack as well as the temporal progress of the execution.

Intel VTune Amplifier uses periodical sampling to generate performance profiles. Besides MPI it supports the threading APIs OpenMP, Intel Threading Building Blocks (TBB), and Intel Cilk Plus. OpenCL analysis is available for recent Intel processors with integrated GPU as well as Intel Xeon Phi processors. Depending on the application scenario, the user has to choose between several different analysis strategies. *Basic* and *Advanced Hotspots* analysis identify code sections where most of the execution time is spent. The *Concurrency* analysis is used to evaluate how effectively an application uses the computing resources, e.g. CPU cores and the integrated GPU. The *Lock and Waits* analysis exposes synchronization objects that prevent efficient parallelization.

VTune presents the analysis results in several views. Figure 3.11 shows two views based on the concurrency analysis of the default Lulesh configuration (30^3 elements) with eight processes, each running three OpenMP threads. The *Summary* view highlights the time spent in different categories for the total program execution. To evaluate the parallelization and execution efficiency, it contrasts effectively used time with time that is spent waiting or overhead time. The rating *Top OpenMP Processes by MPI Communication Spin Time* sorts MPI ranks according to their MPI communication spin time, claiming that less spin time means more critical-path time of the respective process. Due to the nature of a profile analysis this might be misleading when load shifts dynamically between processes during the program execution.

In hybrid MPI/OpenMP programs, VTune also computes the potential gain when optimizing OpenMP regions on specific MPI ranks. The *Top Hotspots* rating is based on the function activity, which refers to the exclusive runtime. The screenshot on the right of Figure 3.11 shows the analysis results in more detail for every function in the call stack. It is possible to choose between a bottom-up, top-down, and caller-callee tree presentation. The *Effective Time by Utilization* evaluates the parallelization efficiency of a program region. *Spin Time* details synchronization behavior and *Overhead Time* refine the parallelization overhead such as thread creation and scheduling overhead. The timelines (at the bottom) visualize the execution over time. Different colors are used to distinguish CPU execution, thread spinning and overhead as well as MPI communication.

NVIDIA Profiling Tools

The NVIDIA Visual Profiler (NVVP) [NVI18e] is a performance analysis tool that is part of the CUDA toolkit. It uses the NVIDIA profiler *nvprof* to collect performance data for CUDA C/C++ applications as well as OpenACC and OpenMP applications with CUDA as compilation target. *nvprof* is also available as a command-line utility, which generates independent profiles and traces for each executed process. The result files can be imported in NVVP. Although *nvprof* does not collect information on MPI and OpenMP, the analysis of the CUDA part of respective programs is supported. Furthermore, it integrates with the OpenACC profiling interface (see Section 4.2.1).

The NVVP analysis of MPI/CUDA applications has been described in [Kra14]. Since version 7.5 *nvprof* supports the measurement of CPU activities based on program counter and call stack sampling at a certain frequency. CUDA activities such as CUDA API calls as well as GPU kernels and data transfers can be visualized in a timeline as shown in Figure 3.12. Low-level insights into CUDA kernels are gained via performance metrics and events that are directly collected from GPU hardware counters and other counter sources. To provide derived software metrics, many patch-based counters, which are generated via binary instrumentation, are collected. Furthermore, it is possible to observe GPU power, thermal, and clock values.

NVVP provides a guided analysis mode, which supports the user to systematically detect potential inefficiencies in a top-down approach. In a first step, the *CUDA Application Analysis*, global GPU usage information is presented and significant underutilization of GPU resources is highlighted. The second stage identifies *Performance-Critical Kernels*, which are investigated in subsequent analysis steps. NVVP automatically identifies an extensive set of performance issues in CUDA kernels and suggests optimization opportunities. In most cases, it reruns the application to gather the required performance counter.

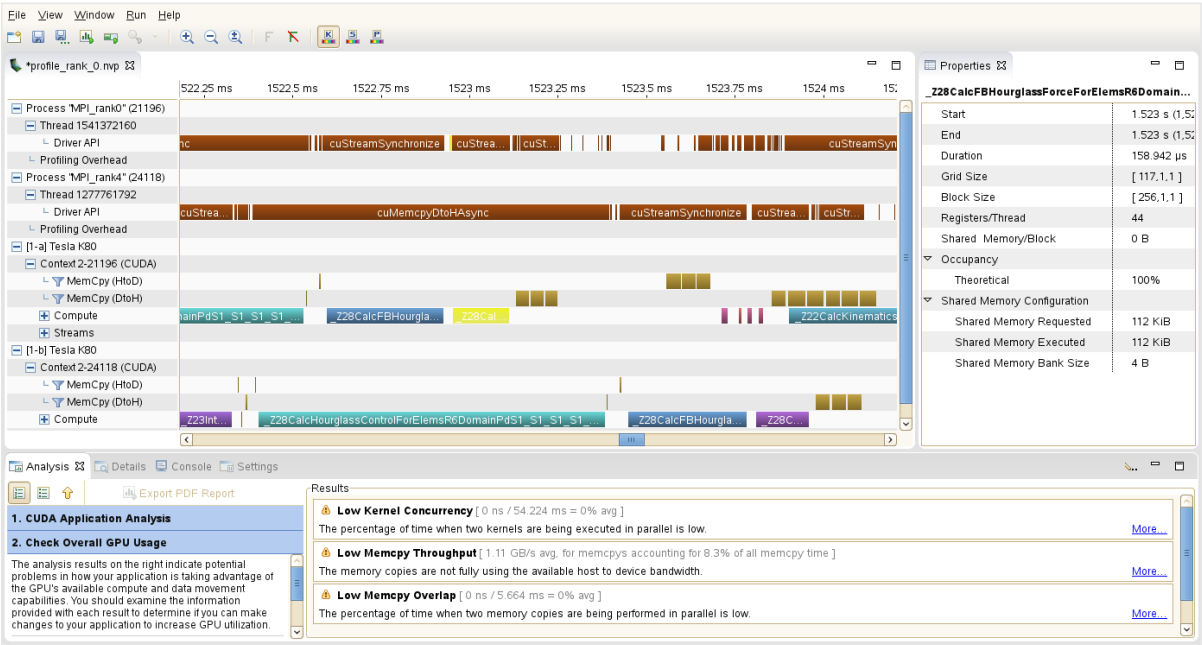


Figure 3.12: The NVIDIA Visual Profiler provides a timeline visualization of CUDA activities on the CPU and GPU. The execution context is labeled on the left. The properties view on the right shows details on the selected activity. The analysis tab shown at the bottom indicates potential inefficiencies in the overall GPU usage.

The automatic analysis for multi-process programs is limited to the initial analysis stage, the evaluation of the global GPU usage. Nevertheless, it is possible to manually specify kernel counters that shall be collected. Since CUDA 8.0, NVVP supports critical-path detection for the CUDA programming model, which originated from work developed as part of this thesis. Details on this analysis are described in Section 4.1.1.

With CUDA 10, new NVIDIA developer tools have been introduced: Nsight Systems [NVI19b] for system-level performance analysis based on statistical sampling and tracing as well as Nsight Compute [NVI18c] for interactive profiling of CUDA kernels.

3.5 Conclusion

Performance analysis of HPC codes is a complex process with a diverse set of available analysis techniques, which all have their advantages and disadvantages. Sampling has a more predictable and easier adjustable runtime overhead than event-based instrumentation, but it can provide only statistical performance data. Profiles have a smaller memory footprint than traces, but they discard the temporal order of events, which prohibits the analysis of dynamic runtime effects. Keeping in mind the drawbacks of event traces, they can preserve very detailed and accurate runtime information, which is a proper basis for the performance analysis presented in this thesis.

Due to its long history in HPC, message passing has been extensively investigated, potential performance bottlenecks are well understood, and most performance tools support it with the MPI standard. The same applies to the fork-join thread model, where OpenMP specifies the de facto standard for multithreading. To detect performance bottlenecks, such as imbalances, sophisticated techniques have been developed. Elaborated performance analysis tools automatically identify inefficiency patterns, the critical path, and the root cause from the execution of MPI and OpenMP application codes and thus provide some guidance in the analysis process.

In contrast, performance analysis for computation offloading, as an additional parallel execution layer, is in its early stages. Several programming APIs with similar offloading functionality exist, but only a few analysis techniques, such as *blame shifting*, have been adopted. The offloading process and its interaction with MPI and OpenMP received little or no attention so far. Performance tools implement either a sophisticated analysis for MPI and OpenMP or an in-depth analysis on offloaded tasks of a specific offloading API. The support for directive-based offloading is either missing or relies on a the availability of a lower-level device-offloading API.

This thesis abstracts common inefficiencies from available offloading APIs (see Section 4.1), which extends the set of already defined inefficiency patterns for MPI and OpenMP. As described in Section 4.2, it also contributes in specifying a portable way to acquire and analyze performance data on the execution of OpenACC and OpenMP directives. To enable a consistent and systematic top-down analysis, the computation offloading model is put in context with message passing and multithreading. A scalable and generic performance analysis approach is presented in Chapter 5. As proof of concept, this thesis extends the performance analysis universe of Score-P by implementing a scalable trace analysis tool, which enables an automatic analysis of scalable heterogeneous applications (see Chapter 6).

4 Performance Analysis for Computation Offloading

Computation offloading creates an additional parallel execution layer next to process and thread-level parallelism. Inefficient offloading can manifest in an idle device or a blocked host thread that is waiting for device activity [CMFMC13]. Both cannot be completely avoided but should be minimized to enable optimal (parallel) resource utilization. A prominent reason for idle or waiting time is a non-existing or suboptimal workload balancing between host and device. This might be due to algorithmic reasons, e.g. data dependencies, or other aspects, such as incorrectly predicted or unexpected processing performance of computing resources.

The proposed analysis methods for computation offloading assume a program trace as input, which provides time and dependency information between events. They do neither consider details on the hardware utilization nor do they consider whether code is suitable for an offloading device or not. Hence, the usage of a processing unit is considered inefficient, if it is idle or waiting for work.

This chapter extends the state of the art in performance analysis for computation offloading models by proposing methods for detecting and analyzing host waiting time, device idle time as well as inefficiencies in the data management between host and device. Section 4.1 describes inefficiency patterns that are used to identify, categorize, and evaluate the severity of inefficient computation offloading. Section 4.2 covers the data acquisition for programs with offloading directives, which is a prerequisite for the pattern analysis. Large parts of current tool interfaces for OpenACC and OpenMP have been designed or influenced in the context of this thesis.

4.1 Inefficiency Patterns in Computation Offloading Models

Offloading inefficiencies occur when the host synchronizes with the device, the device idles, or device data is not efficiently managed by the host. An early synchronization with the offloading device wastes CPU time by delaying the execution of subsequent tasks on a host thread. An idle offloading device does not execute any task and hence wastes device resources. Furthermore, device data allocation and data movement between host and device can delay the execution on host and device. Table 4.1 summarizes inefficiency patterns for computation offloading and specifies a respective severity, which is the time that is wasted in non-productive work.

Computation offloading models have inherent execution dependencies. For example, a device task cannot start before it has been triggered by the host and it has to end before a successful device synchronization ends. The knowledge about constraining dependencies between offloading operations on the host and offloaded tasks can be used to detect and analyze patterns of inefficient execution behavior. The following sections describe inefficiency patterns for computation offloading, their detection, and their usage for performance analysis. They revise and extend the work presented in [Sch13], [SDJ16], and [DTJK17].

4.1.1 Early Device Synchronization

Programming models for computation offloading provide various possibilities to synchronize the host execution with device tasks. Considering common offloading APIs, the synchronization object is either a whole device, a device stream, a device task, or a device event. Global device synchronization ensures that all tasks in all device streams are finished. The synchronization with a device stream is completed when all previously launched tasks in this stream are finished. The host synchronizes with a device task or device event to ensure it is completed. Figure 4.1 compares device, stream, and task synchronization in a time-space diagram.

Pattern	Accounting/Severity
Early Device Synchronization / Idle Host	
Early blocking device synchronization	Time between synchronization start and end of temporally last associated device task
↳ Synchronous offloading	Time between the start of the first and the end of the temporally last associated device task
Early non-blocking device synchronization	Execution time of unsuccessful tests
Idle Offloading Device	
Idle	Time when no device task is active
Compute idle	Time when no computational device task is active
Inefficient Device Data Management	
Late data transfer	Time when a transfer delays the execution of any compute task on the device
Multiple consecutive data transfers	Accumulated overhead of excessive transfers

Table 4.1: Inefficiency patterns for computation offloading

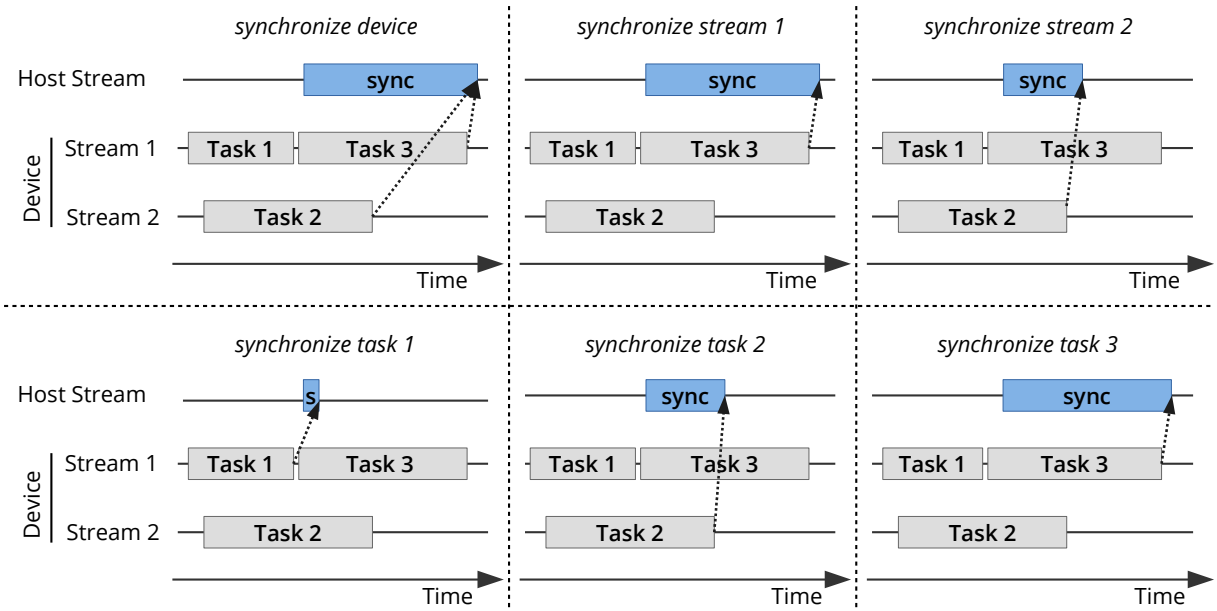


Figure 4.1: Offloaded tasks can be synchronized on the device, stream, or task level. The timelines show one host stream and three tasks that have been previously launched on two device streams. The arrows between device streams and the host stream represent dependencies resulting from the specific synchronization.

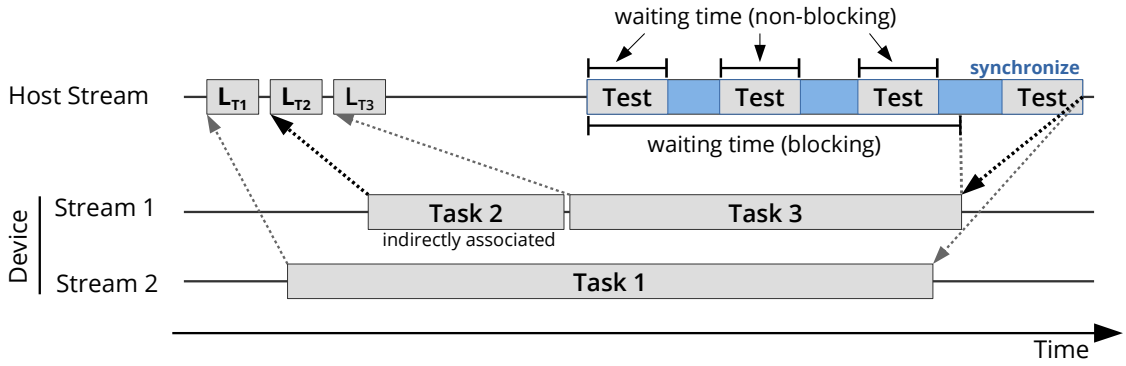


Figure 4.2: Blocking and non-blocking early device synchronization are handled similarly. The synchronization comprises all device streams. As *Task 2* delays *Task 3*, it is indirectly associated to the synchronization. Dependency paths are visualized as dashed arrows. The critical path follows the last task in the synchronization (*Task 3*), its directly preceding task (*Task 2*) to the begin of the launch operation of the timely first associated task (L_{T2}).

Device synchronization is early when it starts before the synchronization object has finished execution. It is performed either as blocking wait operation or non-blocking query mechanism. Figure 4.2 illustrates both patterns in a timeline view. There is little difference in the analysis concept, which also results in similar analysis requirements. The detection of device tasks that are associated to the synchronization is required in either case, while waiting-time analysis and critical-path detection follow the same fundamental rules. The differences of blocking and non-blocking synchronization, in terms of usage and analysis details, are discussed separately.

Analysis Requirements

Assuming that the execution stream (process, thread, or device stream) of operations is known, the following additional information is required to analyze early blocking device synchronization:

- launch/trigger operations of device tasks and device events – begin time and target device stream
- device tasks – begin and end time
- device events – time of occurrence
- synchronization operations (wait and test) – begin time, end time, and synchronization object

If the begin timestamp of the synchronization is less than the timestamp of the last ending associated device task (synchronization and device task overlap), early device synchronization is detected. The synchronized object and the target device stream are needed to identify associated device tasks. To determine the waiting time and the critical-path time, the begin time of device tasks is required, too. The begin of task launch operations is used to create a path from device tasks back to the host, which is a requirement for the critical-path detection. Device events and their triggers are a requirement for event-based task synchronization.

Detecting Associated Device Tasks

The overlap between the synchronization operations and their associated device tasks determines the waiting time on the host and therewith the severity of an early device synchronization. It can be distinguished between directly and indirectly associated device tasks. A device task is directly associated to a host-device synchronization, if it overlaps with the synchronization operation on the host, and

- it is the synchronized object,
- it is immediately executed before the synchronized device event, or
- it is executed last on a synchronized device stream.

Due to task dependencies and the property of device streams to execute tasks sequentially[1], the synchronization of a (directly associated) device task can also affect other device tasks. In most offloading programs, the synchronization of a device task does not finish before all previously executed tasks in the same device stream are completed. Hence, a device task is indirectly associated to the host-device synchronization, if it delays an associated task and has not been previously synchronized by the same host thread.

Furthermore, the device synchronization can affect multiple device streams. Indirectly associated device tasks can include tasks from device streams that are not synchronized, due to task dependencies[2]. Section 5.1.1 – *Synchronization Types* describes the requirements on performance data to rebuild inter-stream dependencies.

Waiting-Time Analysis

The associated device task that ends last in a synchronization determines the end time of the potential wait state on the host, and is a potential continuation point on the device for critical-path detection. It can be identified performing a linear search ($O(n)$) over n associated tasks, or more efficient with complexity $O(s)$ for s device streams, using the property that events in a single device streams are in chronological order. The extend of the host wait operation over the end of the last synchronized device task is not accounted as waiting time, but interpreted as parallelization overhead (device management).

To account the cause of host-blocking device synchronization, all associated device tasks are blamed. The waiting time in the synchronization equals the blame that is distributed over the tasks (see Definition 5 in Section 5.3.1). Section 5.3.2 – *Blame the Shortest Path* describes the used root-cause analysis in detail. The total amount of distributed blame per wait state can exceed its waiting time, when associated device tasks are executed concurrently.

Critical-Path Detection

The critical-path detection starts on the host at the end of the execution and progresses backwards in time. It changes the execution stream when it encounters a wait state. If a wait state is enclosed in a device synchronization, it continues at the end event of the last ending task that is associated to the synchronization. It proceeds on this device stream until there are no more immediate predecessor (associated) tasks. The critical path changes to an immediate predecessor task on another device stream, if such a dependency exists. From the temporal first device task on the critical path, it continues at the begin event of the task's launch operation. The task's launch operation on the host has to carry an identifier to the device stream of the launched task or to the task directly. Due to the property of device streams to execute tasks in their launch order, the matching is unique.

Early Blocking Device Synchronization

The host performs an early blocking device synchronization when it cannot continue processing until the offloading device finishes execution, e.g. due to data dependencies. Hence, a host thread begins to wait for one or more uncompleted device tasks.

Table 4.2 gives an overview of host-blocking device synchronization options in CUDA, OpenACC, OpenCL, and OpenMP. It occurs explicitly as a result of a dedicated API routine or directive, or implicitly as side-effect of an operation. However, the pattern analysis does not need to distinguish between explicit or implicit synchronization. CUDA and OpenACC provide API routines to explicitly execute a global device synchronization. In OpenMP programs, there is an implicit global device synchronization at the

[1] In CUDA and OpenACC, device streams execute tasks sequentially. OpenCL provides out-of-order device streams. OpenMP does not use the concept of device streams.

[2] In CUDA and OpenCL, dependencies between tasks on different device streams are created with device events. The OpenACC API provides the *async* clause to describe dependencies between device execution streams.

Offloading API	Device	Stream / Queue	Event / Task
CUDA	explicit & implicit	explicit	explicit
OpenCL		explicit	explicit
OpenACC	explicit & implicit	explicit	explicit
OpenMP	implicit		explicit & implicit

Table 4.2: Blocking device synchronization can be explicit or occur implicitly in common computation offloading APIs. Triggered by a dedicated API routine or directive, it is explicit. It is implicit, if the offloading API specifies routines or directives which require a host-synchronous behavior, but have a different primary task than synchronization. The OpenMP and OpenACC APIs allow the program to wait for specific device tasks. The CUDA and OpenCL APIs provide device events to enable device-task synchronization.

end of target regions, if the *nowait* clause has not been used. As CUDA, OpenCL, and OpenACC use the concept of asynchronous non-blocking execution streams, they provide routines for explicit stream synchronization. Implicit host-blocking stream synchronization is not specified in offloading APIs.

Synchronous device tasks are a special case of early blocking device synchronization, where the host triggers (or launches) device tasks and immediately blocks the execution of the host thread until the device tasks are completed. The accounted waiting time on the host is the time between the start of the first and the end of the last associated task. Blame distribution and critical-path detection regard synchronous offloading as a normal blocking synchronization. OpenACC and OpenMP allow the programmer to describe synchronous execution for all device tasks, whereas neither CUDA nor OpenCL provide an API routine to trigger a host-synchronous compute task on the device.

The pure detection of synchronous device tasks requires only host events for CUDA, OpenCL, and OpenACC. In the CUDA and OpenCL APIs, they can be detected by function name and arguments. In the OpenACC profiling interface, all event callbacks provide information on the *async* clause, which can be used to expose synchronous operations. However, the current OpenMP 5.0 [OMP18] tools interface does not provide the means to expose whether a device task is triggered synchronously or asynchronously.

The optimization of host-synchronous data operations is most often a replacement with a respective asynchronous version, which enables the host to perform any work or trigger other device tasks during the data operation. However, using a synchronous device task is reasonable, if the host cannot continue to work its completion, e.g. synchronous data transfer from device to host. For energy reasons, the implementation of the wait operation is of importance, e.g. it can be implemented as spin wait or thread yield, which makes it a real wait consuming CPU clock cycles or an idle operation. The analysis may consider this difference to amplify the severity of the pattern or distinguish between an idle and a busy-wait pattern.

Early Non-Blocking Device Synchronization

In contrast to blocking synchronization, a query, probe, or test for completion does not block the execution of a host thread until the tested device object has finished execution. This allows the host to perform any work in between test operations, which enables an adaptive load balancing. Depending on the offloading model, the execution status of individual device streams, tasks or events can be requested. Table 4.3 summarizes non-blocking synchronization options in CUDA, OpenCL, and OpenACC. The OpenMP API specification [OMP15] does not describe a query mechanism to request the execution status of an offloading device.

The detection of non-blocking synchronization assumes that the test operations are used for the purpose of synchronization with the device. A test that returns *completed* is called a *successful test*. It signalizes that the object under test has finished execution, and completes the host-device synchronization. A test

Offloading API	Device	Stream / Queue	Event / Task
CUDA		cuStreamQuery	cuEventQuery
OpenCL			clGetEventInfo
OpenACC	acc_async_test_all	acc_async_test	acc_async_test

Table 4.3: Query routines for the device execution status

that returns another status than *completed* is called an *unsuccessful test*. All unsuccessful tests that precede a successful test for the same object under test introduce an inefficiency as they waste time on the host. Another restriction is that there is no other synchronization in between any of the test operations which affects a common device stream or device task. An inefficiency emerges if there is at least one unsuccessful test that fulfills the previous conditions. Figure 4.2 illustrates non-blocking synchronization with test operations.

Non-blocking host-device synchronization becomes increasingly inefficient with the number of unsuccessful test operations. The first unsuccessful test in the group of associated test operations determines the begin of the synchronization. Each unsuccessful test is considered unnecessary or redundant and is therefore assigned waiting time for its duration. All associated tasks can be blamed for causing this waiting time. Reducing the frequency of test operations usually decreases the negative impact of this pattern, which, however, can also result in more device idle time, since the host may be informed late about completed device tasks.

As non-blocking synchronization does not generate wait states in offloading scenarios, it can be argued that the critical path stays on the host. However, the associated test operations can also be interpreted as a *non-blocking wait state*. In the latter case, the critical path detection works similar to host-blocking synchronization. To decide whether non-blocking synchronization introduces a wait state, the execution time of associated unsuccessful tests has to be determined. If more time is spent in unsuccessful tests than in other work on the host between the first unsuccessful test and the successful test, an analysis can assume a non-blocking wait state.

4.1.2 Idle Offloading Device

An idle offloading device is, similar to a waiting host thread, a waste of compute resources. It is unavoidable during the program startup and shutdown, as the device has to be initialized, workload generated by the host, device memory allocated, data moved to the device, and finally results be read back. Offloading is often used to accelerate portions of a code that can be more efficiently processed by a respective offloading device other than the host processor. Remaining code parts are executed on the host, which potentially leaves the device in an idle state.

Device idle can be further refined into *idle* and *compute idle*. An offloading device is considered *idle* when it does not execute any task. It is considered *compute idle*, when it does not execute any compute kernel. The begin, the end, and the device stream of devices tasks are required to analyze idle times on offloading devices.

Detection of Device Idle

Event traces are an effective technique to detect the idle time of offloading devices. By post-processing such traces, an event analyzer might iterate over the events of streams that refer to the same device and detect regions when no task at all or no compute task is active on the device, e.g. based on reference counting of active tasks. However, program traces are limited to the perspective of the observed program. Device idle cannot be reliably detected when the device driver allows multiple concurrently executed programs to utilize the same offloading device, e.g. with time slicing. As HPC applications typically run on exclusive compute resources, this limitation may be irrelevant for a reasonable program analysis.

Another approach to detect the device idle time are device libraries such as the NVIDIA management library (NVML) [NVI18d]. In combination with periodical sampling, a performance tool can track the device utilization over the execution of a program. Typically, this information does not include data movement, which is why only compute idle can be detected. To correlate this information with device compute tasks and create a temporal context, a sampling instance can be executed concurrently to event collection on the device, which however combines the limitations of both, sampling and instrumentation (compare Section 3.2.1). In addition, NVML's measurement granularity is quite sparse, which prevents an exact distinction between the execution of fine-grained tasks and device idle.

Cause of Device Idle

The device idle time is a measure for wasted computation-offloading resources. All host streams (processes and threads) that access the device can be blamed for not keeping it busy. In this case, the blame distribution starts from the task trigger that ends the device idle until a device synchronization or a task trigger for the same device backwards in time. Figure 6.5 in Section 6.5.1 illustrates the procedure using an example execution. However, computation offloading is often used to accelerate only portions of the code, which is why the application analyst has to decide whether the host should be blamed for device idle.

Device idle can be a symptom of data management inefficiencies. The difference between the device compute idle time and the device idle determines the device data movement overhead. This pure device communication time is a part of the program's parallelization overhead, which might be reduced by splitting a transfer into chunks to allow a computation to start earlier and overlap with the remaining data movement (see *Late Data Transfer* in Section 4.1.3). Such an optimization reduces both device idle and device compute idle time.

4.1.3 Inefficient Device Data Management

Patterns of inefficient device data management comprise device memory allocation and data movement between host and device. The symptoms are in both cases a blocked host thread or device idle time. Device memory allocation is a prerequisite for moving data to the device and often implemented as operation that blocks the host and device execution. Data movement is a potential bottleneck in programs that access physically distributed memory. It often provides a high potential to improve the overall program performance, which makes it an optimization target for paradigms such as message passing and offloading.

Device data operations are discussed separately from generic device tasks, as they can overlap with device computation, even if all compute units on an offloading device are occupied. With this possibility, additional inefficiencies arise when device data operations are not overlapping with device computation. Three patterns of inefficient device data management can be distinguished and are explained in the following.

Host-synchronous Data Operations

Most computation offloading APIs enable synchronous and asynchronous device data transfers, whereas device memory allocation and deallocation is usually synchronous with respect to the host execution. If host-synchronous data operations are considered as device tasks, their detection and evaluation follow the same rules as described for host-synchronous operations in Section 4.1.1. In the CUDA programming model, synchronous data operations implicitly synchronize all device tasks in all device streams [NVI18a], which has to be considered in blame distribution and critical-path detection.

To minimize the device allocation overhead, a single allocation can be used to create a device memory pool. Further allocations are managed virtually by the host and mapped to the already allocated device memory. Device memory allocation might be performed with an additional thread as early as possible

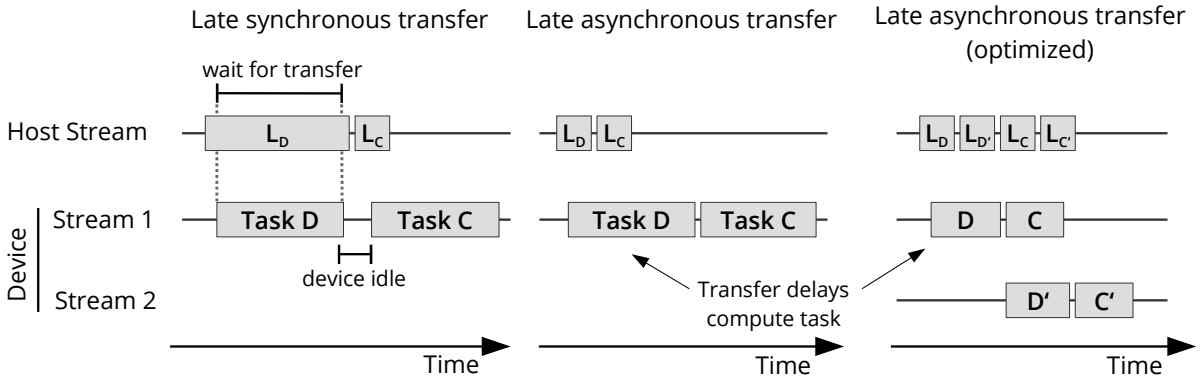


Figure 4.3: A late data transfer can occur with different severity. *Task D* denotes a data transfer, *Task C* a compute task. (1) A late synchronous transfer blocks host and device execution, and induces additional device idle time. (2) Late asynchronous transfers still delay the execution of a compute task on the device. (3) Splitting asynchronous transfers into smaller chunks can reduce the delay and enables overlapping of data transfer and device computation.

to avoid delaying a device task. Alternatively, offloading models should enable asynchronous device memory allocation similar to other device tasks to reduce the host waiting time and enable a concurrent execution to compute tasks.

Late Data Transfer

A device data movement is considered late if it delays the execution of any compute task on the same device. The worst case emerges when the late transfer is a host-synchronous operation, as it also delays the host thread's execution and additionally introduces device idle time between the communication task and the following compute task. Figure 4.3 illustrates the inefficiency caused by late data transfers. It also shows an optimization opportunity, where the transfers are split into parts, to reduce the delay and enable some device task overlap.

Late synchronous data transfers are detected by identifying a synchronous data transfer with a directly following compute task trigger operation. This is done based on the host-side events. However, the detection of device idle requires the end event of the transfer task and the begin event of the compute task on the device.

A late asynchronous transfer delays the execution of a device compute task and does not fully overlap with a device compute task. The detection of this pattern requires device tasks and their trigger operations on the host (see the detection of delayed tasks in Section 6.3.3). If a list per event stream is used, a successive task can be detected with constant complexity $O(1)$. An overlapping compute tasks can be detected using a binary search on event lists with logarithmic complexity $O(s * \log n)$, where n is the number of elements and s the number of device event streams. The severity of this pattern is the difference between the compute begin and the transfer begin.

Consecutive Data Transfers

The efficiency of data transfers depends on the per-transfer overhead and the achieved overall bandwidth. Each device-data transfer has a certain overhead [Har12] and an execution latency. The overhead includes the transfer's launch operation on the host and extra management data to be transferred. Hence, the accumulated overhead decreases by reducing the number of transfers. Another optimization opportunity is the overlapping of multiple transfers, which can increase the used bandwidth [SRC17]. As described previously for the late data transfer pattern, overlapping transfers with computation also reduces the program's communication overhead.

In short, consecutive data transfers are inefficient, if all or some of them can be batched into a larger one, or one of them can be overlapped with a device task. A respective analysis considers two consecutive data transfers as inefficient, if the following conditions are true:

- they share the same source (device or host) and the same destination (device or host)
- neither of them is overlapping with any device task
- no computation is performed between them on the same device
- no communication of any other paradigm is initiated or ends between the triggers of the device data transfers

Based on these conditions, the detection of inefficient consecutive data movement requires begin and end timestamps of device-transfer tasks, their host triggers, and their device stream. To decide on overlapping or intermediate compute tasks, either begin and end of compute tasks or their triggers on the host including the device stream are examined. Furthermore, information on data movement operations from non-offloading models is needed to consider additional data dependencies.

In a sequence of inefficient consecutive data movement tasks, all but the first contribute to the inefficiency. The severity of this pattern cannot be determined reliably without knowledge about the overhead of a single transfer. In theory, it is the accumulated overhead of excessive transfers, hence, the overhead of all transfers but the first one. This also includes the execution time of the excessive transfer trigger operations on the host. As there is usually a bandwidth advantage of large data transfers over multiple small ones, the benefit of combining multiple transfers might be even larger than the theoretical severity. However, combining data transfers might require to reallocate data, which in turn introduces overhead. As the transfer overhead is typically not available, an upper bound of the severity is used. It is the accumulated time of all excessive transfer tasks and their trigger operations.

This pattern often occurs in programs that use directive-based offloading models. One reason are implicit data movements between host and device. This happens, for example, if variables that do not appear in a *copy* or *map* clause are referenced in an offloading compute region. Due to different memory addresses of the referenced host variables, multiple data transfers are invoked. Furthermore, there are compilers which split transfers to reduce the delay of a late data movement, which can nevertheless result in multiple inefficient data movements.

4.1.4 Interference of Inefficiencies by Other Paradigms

Although parallelization on process, thread, and offloading level can be orthogonally applied to each other, inefficiencies within a programming model can influence several levels and thus also the course of the critical path in the entire program, and the distribution of blame, which is explained in detail in Section 5.3 and Section 5.4.

In the following, the cooperative usage of computation offloading with message passing and multithreading is discussed. A common usage pattern of them has already been depicted in Figure 2.5. Different usage scenarios emerge from algorithmic requirements and optimization opportunities.

Message Passing and Offloading

Programs that scale across compute nodes most often use MPI to communicate between processes. Two usage patterns that emerge in combination with offloading are visualized exemplarily in Figure 4.4.

The usage of offloading in between MPI communication or synchronization operations is a common case. The consequences are device idle during MPI operations, which increases with the duration of MPI operations. MPI implementations that are GPU-aware[3] foster this usage pattern. Alternatively, MPI operations can be performed concurrently to device tasks, e.g. to hide the MPI communication overhead. Although this might reduce device idle time and the severity of early synchronization as

[3]GPU-aware MPI implementations allow the usage of GPU memory addresses as arguments to MPI communication routines.

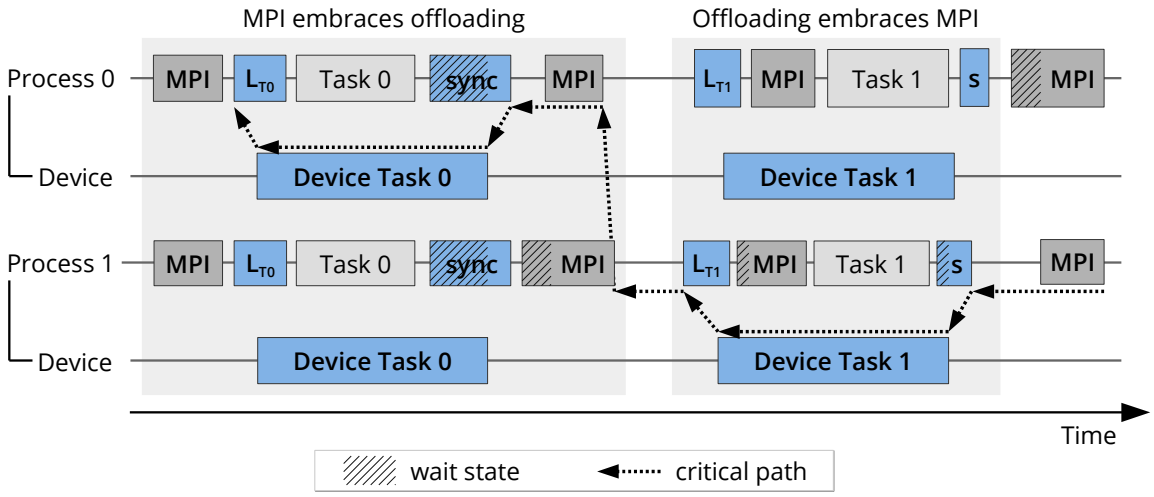


Figure 4.4: Scalable heterogeneous programs often use computation offloading between MPI communication. However, MPI communication can also be hidden behind offloaded tasks, which might reduce device idle time and waiting time in early device synchronization.

shown in Figure 4.4, it holds the risk that device idle emerges, if the MPI operations run much longer than the concurrent device tasks. Similar usage patterns emerge by combining message passing and multithreading.

Device idle can also be reduced when an offloading device is utilized as a shared resource from multiple processes within a program, which is possible with NVIDIA's Multi-Process Service (MPS) and the Hyper-Q technology on NVIDIA GPUs [NVI19a]. However, this potentially increases the host waiting time, as device tasks may be deferred due to resources contention.

Multithreading and Offloading

With an increasing number of available CPU cores, it is reasonable to balance workload between multiple host threads and the offloading device. A possible sequence is to trigger offloaded tasks, start multithreaded execution, and synchronize with the offloading device. As a result, host threads are idle while triggering device tasks and during an early device synchronization, which wastes CPU compute resources. In return, multithreading overhead such as fork, join, and barriers are hidden behind device tasks. Hence, this usage pattern is reasonable, if the offloading device is the critical resource which processes most of the workload. In this case, the waste of compute resources on the host (early device synchronization) is rather tolerated than wasting compute resources on the offloading device.

Instead of controlling offloading devices in the non-threaded part of the program, it can be reasonable that each thread controls a different offloading device, which moves the offloading overhead from serial to thread-parallel execution. Compared to sequential offloading to multiple devices, this reduces the total program runtime and device idle, since tasks on all devices can be triggered in parallel. However, thread-parallel offloading might also increase the severity of early device synchronization. Load balancing between offloading device and multiple host threads can be performed with nested parallelism, when offloading has been started.

Multiple threads may also share an offloading device to increase the device utilization, which is similar to offloading from multiple MPI processes. There are also other possibilities to combine offloading and multithreading, which may have their use case, e.g. sequentially offload tasks with parallel device synchronization or vice versa, or the usage of offloading-control threads while other threads process parts of the workload.

4.2 Data Acquisition for Directive-Based Computation Offloading

Compiler directives introduce an additional challenge for program analysis. Several execution details are implicitly managed by the compiler or runtime and cannot be measured by simply taking a timestamp before and after a directive, or the associated structured block. For example, the fundamental construct of OpenMP, the parallel construct, is executed by a team of threads and ends in an implicit barrier, which is not visible with simple instrumentation and thus cannot be analyzed for imbalances. The execution time of computation offloading directives might be measured with enclosing timestamps, but the host-device interaction and device activities depend on the compiler and runtime library, which have some freedom in the implementation of most directives.

Source-to-source transformation tools such as OPARI2 enable the instrumentation of OpenMP directives. However, the approach is intrusive, as it does not only take additional timestamps, but modifies and adds new directives, e.g. to measure the runtime of an implicit barrier. Although this gives additional insight into the execution, it might prevent the compiler from optimizations and lead to a different runtime behavior. Support for offloading directives in OPARI2 has been investigated and prototypically implemented in [DSGS14].

OpenACC and OpenMP are widely-used directive-based approaches that support computation offloading. Their tool interfaces provide a portable way to collect performance data on the execution of directives from the perspective of the OpenMP or OpenACC runtime. In contrast, instrumentation with OPARI2 generates a view from the source-code perspective. A comparison between the OpenMP Tools (OMPT) interface and OPARI2 was carried out in [LDTW14]. Both tool interfaces, OMPT and the OpenACC profiling interface (ACCT), have already been discussed in the context of Score-P and compared in [DTC+16]. Their scope and functionality are presented in the following sections. As sampling is an important alternative to event-based instrumentation, it is shortly discussed with focus on computation offloading for both OpenMP and OpenACC in Section 4.2.4.

In the context of the thesis, many contributions to the OMPT and the ACCT interface have been made. As a member in the OpenMP tools subcommittee and the OpenACC consortium, I actively participated in the process of specifying these tool interfaces with focus on enabling event-based tracing. Next to the design of the initial version of the OMPT interface [EMCS+13] within the OpenMP tools group, which later became the OpenMP tools subcommittee, I contributed significantly in the conception and definition of the OMPT tracing interface, which enables the collection of events from target devices. The contributions to the OpenACC profiling interface include the specification of requirements for performance analysis and event-based data acquisition as well as the revision and extension of the initial proposal from PGI.

4.2.1 The OpenACC Profiling Interface

Version 2.5 of the OpenACC specification [OAC15] introduces an interface for performance tools. It specifies a set of runtime events which can be used for instrumentation-based data acquisition. The following sequence of actions shows the typical interaction between tool and runtime based on the ACCT interface:

1. Load the tool library by linking it into the application
2. OpenACC runtime: call tool registration routine `acc_register_library`
3. Tool: register for event callbacks
4. OpenACC runtime: trigger event callbacks

A tool attaches to an OpenACC runtime by linking it into the application and implementing the tool registration routine `acc_register_library`. In case of static linking the OpenACC runtime will directly invoke the registration routine. If the tool is dynamically linked, either the environment variable ACC_PROFLIB or LD_PRELOAD has to be set and point to the tool's OpenACC profiling library. It is also possible that the application invokes the tool registration routine. Addresses to routines for registering and unregistering of events is passed to `acc_register_library`. Once a tool has registered

Event `acc_ev_`	**Occurrence**
Kernel Launch Events:	
`enqueue_launch_[start|end]`	before/after a kernel launch operation
Data Events:	
`enqueue_upload_[start|end]`	before/after a data transfer to the device
`enqueue_download_[start|end]`	before/after a data transfer from the device
`create/delete`	when the OpenACC runtime associates/disassociates device memory with host memory
`alloc/free`	when the OpenACC runtime allocates/frees memory from the device memory pool
Other Events:	
`device_init_[start|end]`	before/after the initialization of an OpenACC device
`device_shutdown_[start|end]`	before/after the finalization of an OpenACC device
`runtime_shutdown`	when the OpenACC runtime finalizes
`wait_[start|end]`	before/after an explicit OpenACC wait operation (clause, directive or API call)
`compute_construct_[start|end]`	before/after the execution of a compute construct
`update_construct_[start|end]`	before/after the execution of an update construct
`enter_data_[start|end]`	before/after the execution of an enter data directive or before/after entering a data region
`exit_data_[start|end]`	before/after the execution of an exit data directive or before/after leaving a data region

Table 4.4: Runtime events as specified in the OpenACC profiling interface. Adapted from [DS17].

events with a respective implementation of callback routines, the OpenACC runtime triggers the event callbacks, whenever respective events occur during the program execution. All callback routines have the same type signature. The first argument provides general information on the execution context, e.g. device type and number, host thread ID, *async* value that triggered the callback, and the source code location with file name and line number.

Runtime Events

An OpenACC runtime dispatches events from three different groups: kernel-launch events, data events, or other events. Depending on the event group, different information are provided in the second argument of event callbacks. Table 4.4 lists all events with a short description of their occurrence and their classification into one of the three groups. Independent of the group, all events have a field that specifies the event type, the parent construct, and whether the event belongs to an implicit or explicit activity.

Data events as listed in Table 4.4 carry the name of the variable, the number of bytes as well as a host and a device pointer to the corresponding data. The kernel-launch group contains only the start and end event of a kernel launch. Event-specific information are the kernel name as well as the gang, worker, and vector size. The group of other events do not carry any additional information. Eminently important for performance analysis are the wait events, which enable waiting time on the host to be exposed without additional information from low-level APIs.

Integration of Low-Level APIs

OpenACC can be transformed into a specific (lower-level) device API. Therefore, predefined types for the low-level APIs CUDA, OpenCL, and COI (Coprocessor Offload Infrastructure) are defined, although the specification is not restricted to specific offloading targets or device APIs. Nevertheless, CUDA and OpenCL are the most prominent target platforms.

The third argument of each event callback provides information on the OpenACC vendor, the device API, and three handles to low-level device-API data structures. This enables a tool to gather additional information from low-level APIs that is not accessible with ACCT or hook into the execution of the low-level programming model. An example is the measurement of device kernels and data transfers. For the CUDA API this can be accomplished by recording CUDA events in OpenACC enqueue launch, enqueue upload, and enqueue download callbacks. When the respective device task is completed, the CUDA API routine `cuEventElapsedTime` can be used to determine the elapsed time between two events, e.g. a CUDA event launched in `acc_ev_enqueue_launch_start` and `acc_ev_enqueue_launch_end`.

The integration of low-level APIs enables a correlation between device tasks and OpenACC runtime activities, if the low-level API provides respective means to measure device tasks. Together with the parent construct and the source code information from the second argument in each event callback, device activities can be mapped back to the OpenACC API in the program's source code. Another option for performance tools is the combination of tool interfaces from low-level APIs, such as CUPTI, with the OpenACC profiling interface. This additionally enables the correlation of low-level API calls with OpenACC runtime activities, which provides a detailed insight into the OpenACC implementation.

Analysis Limitations

Up to the currently most recent OpenACC 2.7 [OAC18], the profiling interface does not cover the collection of device tasks. If corresponding events are not supplementary collected, e.g. with CUPTI or CUDA events for CUDA devices, the runtime of individual device tasks cannot be determined. This restricts the offloading analysis [DTJK17], which has been presented in Section 4.1. Except for synchronous device tasks, no other inefficiency pattern can be reliable detected and analyzed. Blocking synchronization is exposed with wait-begin and -end events, but the overlap with device tasks cannot be exactly determined.

Some approximations can be performed based on the begin and end events for wait operations and device-task launch on the host. As these events provide information on the associated device execution stream, an upper limit of the runtime of device tasks can be determined. It is the difference between the wait end and the task launch begin. As a result, it is possible to determine the lower bound of device idle.

If more than one device task is synchronized in a synchronization operation, the tasks have to be handled as a group, because the runtime of individual device tasks is not available. The group can be on the critical path or blamed for causing waiting time. To identify the device tasks of such a group, the interface provides source code information together with the name of compute tasks and the variable name of data tasks in the respective launch operations.

Operations that query the device status such as `acc_async_test` and `acc_async_test_all` are not covered in the profiling interface. To detect early non-blocking synchronization nonetheless, calls to these routines including information on their arguments can be intercepted based on library wrapping.

For a transparent performance analysis of OpenACC applications, host- and device-side information on the execution of OpenACC directives are necessary. Low-level interfaces cannot trace back to higher-level programming abstractions by themselves and therewith they cannot enable the correlation with the source code in OpenACC applications. However, they can be used for the analysis presented in Section 4.1. Currently, only the combination of the ACCT interface with a low-level device interface allows a reasonable performance analysis of OpenACC programs.

4.2.2 The OpenMP Performance Tools Interface

The OpenMP 5.0 specification [OMP18] contains a revised version of the OMPT interface that has already been defined in the OpenMP technical report 2 [EMCS+14]. In the context of this thesis, many contributions have been made to the current performance tools interface for OpenMP. The OMPT interface has been designed for tools rather than for direct use by applications. It defines an API that allows first-party tools to explore an OpenMP implementation by means of sampling and event-based instrumentation.

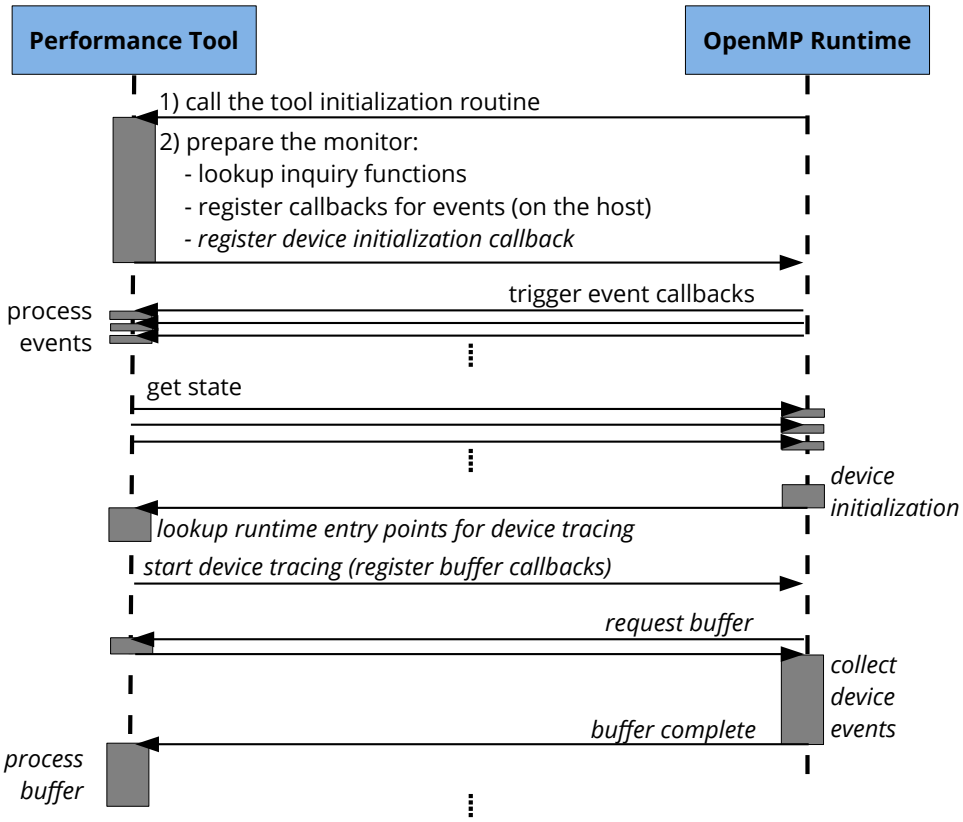


Figure 4.5: Example sequence of interactions between a performance tool and an OpenMP runtime with OMPT. Device-related activities are highlighted in italic font. Adapted from [DTC+16].

As usual, a tool is either statically or dynamically linked into the application. The OMPT-enabled OpenMP runtime triggers the tool initialization before any other OpenMP event occurs. This enables a tool to register for event callbacks and the initialization of target devices. Figure 4.5 illustrates the interaction between a tool and the OMPT-enabled OpenMP runtime. The tracing interface for target devices is depicted in Section 4.2.3, as it has been significantly influenced by the work on this thesis.

Instrumentation-based tools can register event callbacks that are dispatched by an OpenMP runtime when a respective event occurs. There is a set of mandatory events an OpenMP runtime has to implement for compliance with the specification. They include fundamental events such as the begin and end of a parallel and target regions. Other mandatory events include the begin and end of implicit tasks, the lifetime of a thread, and device initialization as well as task creation and task scheduling. An OpenMP runtime must guarantee that mandatory event callbacks are invoked every time an associated event occurs.

Event callbacks that may not be dispatched by an OpenMP-compliant runtime include the group of events for blame shifting. These events expose several kinds of synchronization regions, i.e. barrier, taskwait, and taskgroup regions. Other optional events are related to the occurrence of task dependences, target data mappings, locks, mutexes, thread idle, and cancellation as well as worksharing constructs, master, distribute, and taskloop regions. Callbacks for optional events are implementation-defined. Depending on the return value of `ompt_set_callback` these callbacks can be invoked *always*, *sometimes*, *sometimes paired* or *never*. If *sometimes* is returned, an OpenMP implementation may not invoke a callback for each occurrence of the associated events, e.g. it may skip a callback on a critical execution path. *Sometimes paired* is similar to *sometimes*, but it guarantees that naturally paired events, such as the begin and end of a region, are either both or none invoked.

With regard to computation offloading, the OMPT interface specifies several event callbacks on the host that handle similar events as specified in Table 4.4 for ACCT. This includes the begin and end of target, target data, target data enter, target data exit, and target update regions. Instead of providing access to

data structures for low-level offloading models, a tracing interface for target devices has been defined (see Section 4.2.3). The *target submit callback* signals that a device kernel is triggered. It is called on the host and can be used to correlate a device-kernel with a target directive in the program code. The equivalent for data operations on the device is the *target data operation callback*.

4.2.3 Portable Acquisition of Device Activity

Neither the OpenACC nor the OpenMP standard are restricted to specific device APIs for computation offloading. Moreover, performance tools cannot rely on the availability of low-level device APIs. To enable a portable acquisition of device activity, a standardized interface is needed. Begin and end timestamps of device tasks are a requirement to effectively analyze offloading inefficiencies (see Section 4.1).

OpenMP

The OMPT interface defines an API for tracing on a target device. It is inspired by NVIDIA's CUPTI activity API [NVI18b], which has been developed for collection of asynchronously executed device activities. A performance tool registers a buffer-request and a buffer-complete callback. With the buffer-request callback, the tool provides a buffer where the OpenMP runtime library can store records. With the buffer-complete callback, the tool receives a buffer with device records. It is the runtime's responsibility to manage buffers and trigger these callbacks. This concept has been implemented in the open source version of Intel's OpenMP runtime and the corresponding *liboffload* [CDT+15].

A tool requests device tracing by registering a *device initialization callback* during the tool initialization routine. When the OpenMP runtime initializes a device, it dispatches this callback, which provides access to OMPT's device tracing API. A tool starts device tracing by calling a *start trace routine* with two callbacks for buffer request and buffer complete as arguments. Depending on the implementation, a runtime library requests a buffer for each device thread, device stream, or a single buffer for all device records. The buffer complete callback is invoked to empty a buffer and enables a tool to process the device records by iterating over the entries. Figure 4.5 illustrates the usage of device tracing with the OMPT tracing interface. Device tracing may be paused, resumed or stopped at any time.

The tracing buffers can store predefined OMPT records or native records. The latter enable the collection of device traces in its native trace format, e.g. CUPTI activity records for CUDA devices. In case the native format is not known to the tool, a respective record can be decoded in an abstract format. Target devices such as Intel Xeon Phi might be able to collect traces in the OMPT trace format, which specifies record types that provide similar information to respective OMPT event callbacks.

As host and target device may use different clock generators, there may be no common time base, which is required to write correctly ordered OTF2 traces. With the OMPT tracing interface, a device timestamp can be converted to a host timestamp, which has the same time base as values returned by *omp_get_wtime*. Alternatively, the device time can be immediately requested together with the host time, which enables a linear interpolation between two points in time.

OpenACC

The profiling interface ACCT that has been specified with OpenACC 2.5 does not contain a portable method to collect device activity. However, a respective extension has been conceptionally proposed in [DJW15]. It is similar to the CUPTI activity API and the OMPT native tracing interface. The ACCT device tracing also specifies the buffer request and complete callback while the OpenACC runtime is responsible for managing the activity buffers. Device records represent device activities with start and end time stamp along with the record type. According to the record type, e.g. compute region or data movement, different fields are available, similar to the ACCT event groups. Device time stamps are converted either via a routine to get the device time or by passing an address to a function that returns a host time stamp to the OpenACC runtime.

4.2.4 Sampling the Runtime State

Analyzing the execution of compiler directives with sampling requires symbol information of the respective runtime. Depending on the symbol information, which are acquired with each probe, the execution state in a runtime might be interpreted from symbol names. However, symbol names are different in individual runtimes, can change between revisions, and may not provide information on the execution state, which makes a reasonable performance analysis complicated. The definition of a standardized interface, which enables sampling of the OpenMP and OpenACC runtime state, resolves this issue. Nevertheless, sampling the runtime state of an offloading library does not cover device activities, which introduces the several analysis limitations (compare Section 4.2.1 – *Analysis Limitations*).

OpenMP

The OMPT interface defines an API for asynchronous sampling [EMCS+13]. This requires an OpenMP runtime to maintain a state for each thread. The thread's OpenMP runtime state can be queried with a respective query function, which returns the thread's runtime state and a wait identifier. The wait identifier is passed as pointer and provides an opaque handle (internal to the OpenMP runtime implementation) of the data object a thread is waiting for. If the reported state is not a wait state, the value of the wait identifier is undefined and can be ignored.

The available states are grouped into work states, wait states, and miscellaneous states. Work states are defined for serial, parallel, and reduction work. Miscellaneous states are the idle, overhead, and undefined state. The wait states are further split into barrier, task, mutex, and target wait states. An OpenMP runtime has some flexibility in distinguishing between different barrier, mutex, and target wait states. For example, it may report the same target wait state for waiting for a target region to complete, a target data mapping to complete, and a target update to complete or distinguish between these states.

OpenACC

The extension to the ACCT interface that has been proposed in [DJW15] specifies a set of states and a state query routine to support asynchronous sampling in OpenACC applications. Similar to OMPT, sampling is only supported on host threads and not on the device. Host threads that execute in an OpenACC runtime trigger accelerator operations or wait for the accelerator to complete an operation. Hence, the OpenACC runtime has to maintain a state for each host thread that executes in the runtime.

Similar to OMPT, groups of wait states have been defined for data, compute, and miscellaneous accelerator operations. Dependent on the OpenACC implementation a runtime might distinguish between wait states in these groups to provide more detailed information on the cause of the wait operation. According to the ACCT data events (compare Table 4.4), data wait states can be distinguished between *upload*, *download*, *create*, *delete*, *alloc*, and *free*. Wait compute states are waiting for a kernels region to complete or a parallel region to complete. In case an OpenACC runtime might not be able to distinguish between data or compute states, e.g. it is waiting for compute and data activities, it reports a waiting for device activity state. All states can be further distinguished in an implicit and an explicit version.

5 Generic Performance Analysis for Heterogeneous HPC Applications

The heterogeneous architecture of HPC systems poses a challenge for the development and optimization of programs that aim to utilize all available computing resources. An efficient execution with proper load balancing usually requires performance tuning and thus a qualified performance analysis. In this context, it is essential to consider all levels of parallelism and their interaction. This chapter proposes a distributed and generic performance analysis which detects execution imbalances and their cause as well as runtime-relevant regions in scalable heterogeneous applications. Based on wait states that have already been extracted from inefficiency patterns, the analysis can be applied without knowledge about the concrete programming model and thus also across different levels of parallelism. Eventually, this enables the detection of the root cause of imbalances, which in combination with the critical path provides even more revealing results than the individual analyses.

Section 5.1 discusses the requirements for the generic analysis and proposes a suitable data representation. A scalable event-trace analysis, which can be used for both, root-cause and critical-path analysis, is described in Section 5.2. The actual analyses are explained in Section 5.3 and Section 5.4, whereby the latter also discusses the combination of both analyses.

5.1 Requirements on Performance Data

Performance monitors typically collect data as an application profile or trace. Both represent execution data about regions in the program code. However, only program traces preserve the temporal sequence of program activities on all execution streams, which is required for critical-path detection and root-cause analysis. Definition 2 specifies the usage of the terms program region and program activity.

Definition 2 (Region, region instance, and activity) *A program region specifies a section in the code. Parallel processing allows a program region to be executed on several streams simultaneously. The single execution of a program region, serial or parallel, is called a region instance. The single execution of a program region on a specific stream is called a program activity.*

An important analysis requirement are existing dependencies between program activities. Common trace formats, such as OTF2, do not explicitly store dependencies, but it is assumed that they can be reconstructed. Dependencies between the events of the same execution stream are implicit, whereas dependencies between program activities on different execution streams require additional records. Section 6.2 describes information that is additionally stored to reconstruct dependencies for parallelization with MPI, OpenMP, and computation offloading.

5.1.1 Generic Data Representation

Analyses such as critical-path detection and root-cause analysis can be applied without knowing the underlying programming models. They are based on the evaluation of synchronization between streams, which is characterized by wait operations of the streams among each other. For the proposed generic analysis, the following types of runtime events have to be distinguished:

- begin and end of operations that wait or test for completion of events on other execution streams (stream synchronization)
- begin of operations that trigger the execution of a program region on another execution stream

These events must be stored together with their timestamps and the stream on which they are executed. As trigger and synchronization operations imply inter-stream dependencies, they must include the referenced execution stream(s). The root-cause analysis that is presented in Section 5.3 also requires these events to be annotated with their programming model to determine associated tasks and synchronization operations on other execution streams and to distinguish the type of parallelization (task-based or stream-centric). For the proposed analysis parallelization (see Section 5.2), process streams and process-local streams must be distinguished. For example, MPI processes are represented by process streams, while threads (other than the master thread) and offloading streams are represented by process-local streams.

Synchronization Types

To reconstruct dependencies, different types of wait operations have to be accompanied with different additional information.

(1) A wait for a specific task or event requires a task or event identifier.
(2) A wait for one or more execution streams requires the referenced streams.
(3) A collective wait on two or more execution streams requires a collective identifier.

For example, the first type of waits occurs, when an offloaded task is synchronized and at an MPI wait, which indirectly waits for a remote send or receive to complete. The identifier of the referenced task might be a combination of referenced stream and stream-local identifier. The second type of waits occurs in computation offloading, when a stream or a device is synchronized. Both types of waits are denoted as one-sided waits in the following, as the synchronization induces the wait operation only on one stream.

The third type of waits represents collective operations, where at least two streams are actively synchronizing with each other. Examples are barriers in MPI and OpenMP. OpenMP barriers can be identified based on its occurrence within the enclosing parallel region, whereas MPI collectives have a communicator and a tag. Blocking MPI point-to-point operations are similar to MPI collectives, since send and receive operations wait on each other and thus, are actively synchronizing. However, the collective identifier for MPI point-to-point operations also contains the referenced stream in addition to the communicator and the tag.

Event Dependency Graph

As depicted in Section 3.3.5, a DAG is a suitable data structure for critical-path analysis. By modeling dependencies, a DAG is also suitable for root-cause analysis of parallelization wait states. Program traces can be easily converted into a DAG, where vertices represent events and edges the happens-before relation of events. The latter means that the start event of an edge happens temporally before the end event of the same edge. An edge either represents a dependency between events on different execution streams or a program activity such as a task on a specific stream.

Each edge $e = (v_i, v_j)$ is assigned with a duration $d(e) = t(v_j) - t(v_i)$ which represents the time between two vertices. The duration is a natural number including zero ($d(e) \geqq 0$), since the tail's timestamp of an edge is always greater or equal to the corresponding head's timestamp ($t(v_i) \geqq t(v_j)$). Edges with an infinite duration represent wait states. In this thesis, such a weighted DAG is called event dependency graph (EDG). It is defined as a tupel of vertices V, edges E, timestamps t, and durations d:

Definition 3 (Event Dependency Graph [SSD14]) *The tuple $G = (V, E, t, d)$ with a set V of vertices and a set E of edges such that $E \subseteq V \times V$, E is acyclic, $t : V \to \mathbb{R}_0^+$, and $d : E \to \mathbb{R}_0^+$ is called an event dependency graph.*

To enable a distributed analysis, a distributed data layout is required. The EDG can store all events in a group of execution streams and respective dependencies. Each event stream (process, thread, or offloading stream) is represented as a path in an EDG, as successive vertices are connected. Several distributed EDGs are connected via remote edges. Remote edges start or end with a remote vertex, which enables the data exchange with a remote analysis process. Figure 5.1 illustrates an EDG.

5.1.2 Impact of Data Reduction

Data reduction might be necessary to keep the memory footprint of the performance data feasible. However, all events that are involved in the formation of wait states have to be available, to enable the analyses that are proposed in Section 5.3 and 5.4. Such events are primarily those on the critical path, the begin and end of wait operations, and the begin of task trigger operations. Other events from compiler- or user-instrumented code regions are not required. Nevertheless, program activities that are not recorded can neither be assigned with critical-path time nor blamed for causing wait states. If only events required for the proposed analysis are available, only time ranges of execution streams can be highlighted.

The following two data reduction approaches keep all required information and only marginally reduce the accuracy of the analysis. The combination of trace and profile data can be used to store all events that are required for the analysis and profiles on the execution of program regions in between the events. However, there is currently no performance data format that supports this. Another approach is the combination of sampling and instrumentation, where required events are instrumented and everything else is measured via sampling. Score-P enables sampling to be used in conjunction with instrumentation of parallel programming APIs since version 2.0.

In case of data aggregation over streams, the presented generic analyses and the inefficiency detection will not work properly. The need for data aggregation and its influence in GPU kernels and OpenMP target regions on Intel Xeon Phi has been discussed in [DSWB12] and [DSGS14]. However, it has also been shown that critical-path and root-cause analysis can be applied, if the implementation is aware of the data reduction strategy and implements respective rules [DSGS16]. Nevertheless, the aggregation of performance data reduces the accuracy of the analysis and thus its significance.

5.2 Parallel Event Trace Analysis

Event traces of parallel applications can easily consume several gigabytes or even terabytes of memory. To analyze such a large amount of event data in a timely manner, an efficient parallel analysis is required.

Assuming that the trace contains multiple execution streams with temporally ordered events, which is the case for most trace file formats such as OTF2 [EWG+12], there are two ways to portion the data for a parallel analysis: temporal and spatial segmentation.

5.2.1 Temporal and Spatial Trace Segmentation

Temporal segmentation divides the set of events at specific points in time, e.g. at global collective operations. As analyses, such as the critical-path detection, require a global blocking collective operation over all execution streams as starting point, the global temporal segmentation depends on the program. Hence, a respective parallel analysis is not reliably scalable, e.g. a global collective other than initialization and finalization might not occur in a parallel program.

The spatial segmentation of performance data refers to the distribution of execution streams (process, thread, or offloading streams) to analysis processes. This approach scales well with the number of execution streams, if each stream requires a similar analysis effort, which is roughly proportional to the number of events that are part of an inter-stream dependency.

With an increasing number of inter-stream dependencies the required data exchange between analysis processes increases. To reduce the communication overhead between analysis processes, it is reasonable to assign a group of streams to each analysis processes. Dependencies within the stream group can be detected and evaluated without costly inter-process data exchange. An optimal spatial segmentation uses the sweet spot between the maximum number of analysis processes and the minimal communication between analysis process (and stream groups respectively). However, this sweet spot cannot be determined without additional analysis of the performance data.

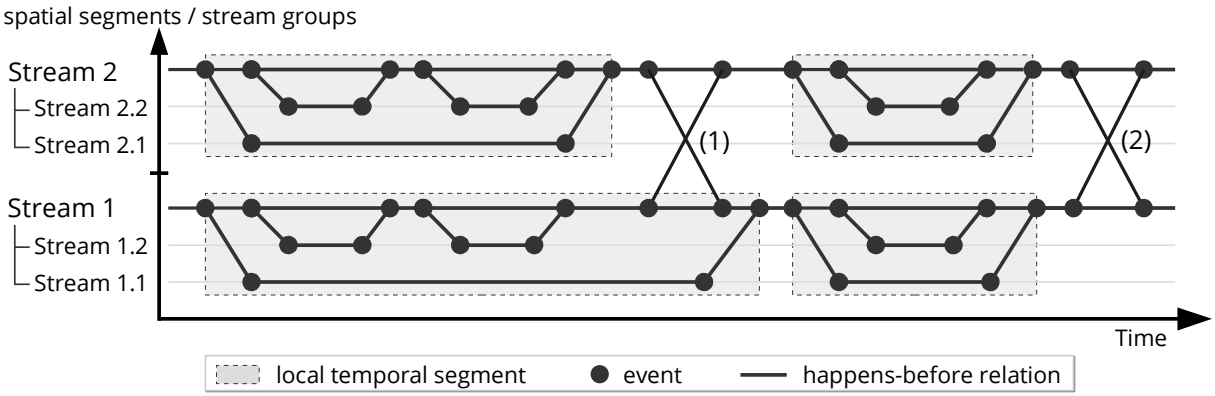


Figure 5.1: The segmentation of event data enables the parallelization of the analysis. The execution path can fork or join at events. (1) and (2) represent blocking collectives between stream groups, but only (2) can serve as global temporal segment boundary, as each stream group may have only one local execution path during the operation.

This thesis proposes to reuse the hierarchy of execution streams from the original program. Accordingly, a process of the original program and its descendant execution streams are combined into a stream group, which forms a spatial segment. It is assumed that communication between processes is kept to a minimum, which should be the case for effectively parallelized applications.

Figure 5.1 illustrates potential temporal and spatial segmentation of event data. A global temporal segment is surrounded by two global collectives, which are operations that join all execution paths. Local temporal segments emerge within a stream group between an event that forks the only local path and an event that joins all local paths.

5.2.2 Distributed and Local Trace Analysis

The mapping of stream groups to analysis processes (spatial segmentation) enables a distributed analysis, which potentially scales with the number of processes. The load is well balanced between analysis processes, if the number of analysis-relevant events is similar in each stream group, which should at least be the case for SPMD-parallel applications. The distributed analysis can be performed independently for each global temporal segments. It is assumed that each segment has a definite start and end of the critical path, and blame cannot be shifted between segments.

The distributed analysis of a global temporal segment has to communicate between analysis processes. Scalasca's MPI communication replay (see Section 3.3.4) is an intuitive implementation of data exchange between analysis processes. To avoid frequent communication, data on inter-process communication can be collected in batches and exchanged with other analysis processes later. Compared to Scalasca's MPI communication replay, the analysis does not depend on the scalability of the original program. Furthermore, each communication in a batch can be analyzed in parallel (local parallelization), as different events are accessed. Summarizing all communication into one batch requires at least one data exchange operation between each communicating process pair ($n \cdot (n-1)/2$ exchanges for n processes).

Local analyses do not require communication between analysis processes (stream groups), as they just affect a process and its associated streams, e.g. OpenMP threads and offloading streams. This enables an embarrassingly parallel processing over local temporal segments, e.g. via multithreading.

5.2.3 Shared Offloading Devices

A distributed trace analysis, e.g. via MPI communication replay [BGWA10], faces another challenge. An offloading device might be utilized as a shared resource from multiple processes within a program.

This can be used as an optimization to reduce idle times of the device and improve its utilization. If the analysis is not aware of device sharing between processes, potentially more device idle and host wait time will be noted.

There are several possibilities to handle device sharing. The first exchanges information on the device state between processes that share an offloading device. However, the additional exchange of data might introduce an intolerable communication overhead. It is also possible that each analysis process reads all device streams that are associated with a physical device. This leaves inter-process communication unchanged, but it slightly increases the trace read time. As offloading devices cannot be shared across nodes, another option is to use one process per compute node. This requires a mapping of execution streams to the hardware topology, where the program was executed. In addition, deadlock-free communication between analysis processes must be guaranteed, which is not trivial if one analysis process is replaying the communication of several processes in the original program.

Assuming that devices are not shared across processes and exclusively used by a single program, the device is in idle state when no process-local device stream executes a task. Eventually, device sharing makes little sense if each process fully utilizes a device.

5.3 Distributed Root-Cause Analysis for Hybrid Programs

The root-cause analysis detects and quantifies the causes of imbalanced execution in parallel programs. An elaborated approach is Boehme's delay-cost analysis [BGWA10], which identifies unbalanced regions as cause of wait states in MPI applications. It assumes that perfectly balanced program regions do not contribute to an imbalance, which is a reasonable approach for SPMD-parallelization. Other parallelization types, such as tasking and computation offloading, require a different analysis approach.

The proposed root-cause analysis combines the delay-cost analysis with a more generic blame shifting technique to generate accurate results for the parallel execution of code regions and to cope with other parallelization types. It abstracts the MPI delay-cost analysis from the MPI messaging model and introduces so-called parallel execution blocks to better distinguish between local and propagating imbalances in hybrid programs. In addition, the cause of wait states is distinguished in unbalanced execution of program regions and those whose runtime reduction will decrease waiting time to the same extent.

Section 5.3.1 presents wait states as the basis for the root-cause analysis, which is triggered, whenever a wait operation is detected. The analysis itself is presented in Section 5.3.2 and 5.3.3, whereas the critical-path detection as appropriate supplement is explained in Section 5.4.2.

5.3.1 Parallelization Wait States

Wait states are a symptom of imbalances or serialization in the execution of a parallel program. They expose a waste of (compute) resources and, to the same extent, they reveal optimization potential. Parallelization wait states are defined as follows:

> **Definition 4 (Parallelization Wait State)** *In a program with multiple parallel execution streams, a wait state is a situation in which one stream is waiting for the completion of an event on another stream. An execution stream might be a process, a thread, or a sequence of tasks on resources of an offloading device.*

Optimizing the execution balance can reduce or eliminate wait states and possibly reduce the total program runtime. This can be achieved by putting more workload on the execution stream, where the wait state occurs, or by reducing the workload on streams that cause the wait state, or both. Neither is a trivial task in SPMD-parallelization, where code changes may affect all execution streams participating in the imbalance. In contrast, workload of host and device can be changed independently of each other in computation offloading scenarios, because host and device execute different code paths.

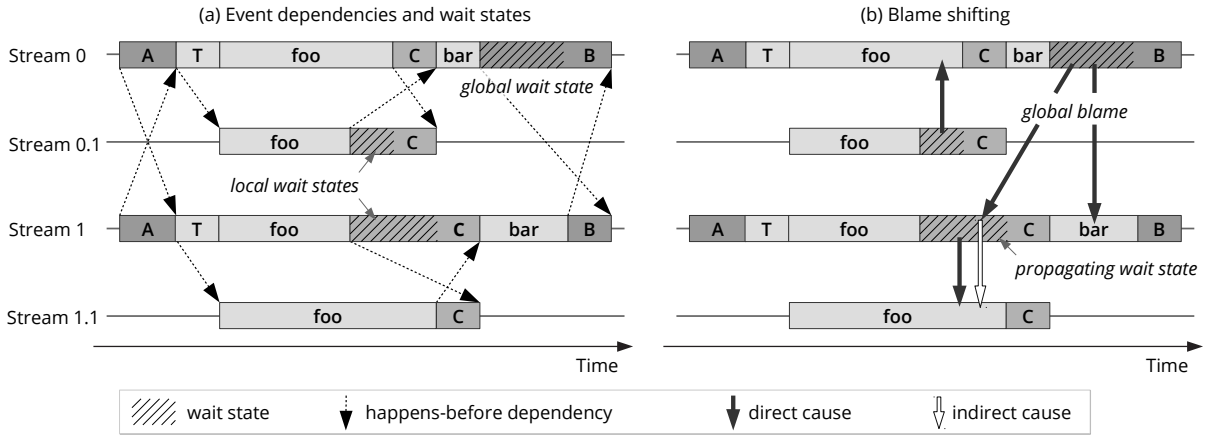


Figure 5.2: Event dependencies, wait states, and blame shifting. Synchronizing operations (*A*, *B*, *C*) are the basis for wait-state detection and blame shifting. *T* represents trigger or fork operations, whereas *foo* and *bar* represent computing regions. *A*, *B*, *C*, and *T* induce dependencies between execution streams.

Figure 5.2a illustrates different types of wait states. Assuming a constant total workload, a wait state is of particular importance, if its elimination reduces the program runtime. Such a *global wait state* affects the global balancing and its cause contributes to the critical path. Imbalances within a stream group (local imbalances) and their *local wait states* may or may not affect the critical path of a program, but wait states in a global collective (global imbalance) definitely affect the program runtime.

Wait-State Detection

Wait states are detected on the basis of synchronizing operations, which can typically be identified by name. However, the performance data may already include a tag that identifies such operations. The resulting inter-stream dependencies are reconstructed by evaluating the parameters of API calls (compare Section 5.1.1). Since the semantics of the programming model or API must be known, the detection of synchronizing operations and respective dependencies is not generic.

Section 3.3.4 has already explained the detection of parallelization wait states in MPI and OpenMP programs, while synchronization for computation offloading has been described in Section 4.1.1. Depending on the number of streams involved in the synchronization, several wait states can occur. Since there must be at least one path without a wait state, the number of wait states in a synchronization is at least one less than the number of streams involved. In one-sided synchronization only one wait state can occur. The detection requires the dependencies between the wait end and all associated tasks (see Section 4.1.1 – *Detecting Associated Device Tasks*). For collective synchronization, dependencies between all begin and end events of the wait operations on all participating streams are required. Usually there is only one stream without wait state.

The wait-state detection is generic and performed autonomously on each execution stream. For each local wait operation, its begin time is compared with the temporal begin of all dependencies that are connected to the wait's end event. The other end of the dependency is either the end of a directly associated task (one-sided synchronization) or the begin of an associated wait operation (collective synchronization). If the wait's begin is earlier than the temporal start of a dependency that is connected to the wait's end event, a wait state was found. This detection approach works also for non-blocking synchronization, as the temporal dependency begin can be the end of a test operation, e.g. in an unsuccessful test.

The proposed parallelization from Section 5.2 requires communication between analysis processes for MPI wait states, whereas determining wait states from threading or computation offloading can use comparatively much faster shared memory accesses.

Waiting Time and Blame

Waiting time and blame are determined on the basis of wait states. The duration of a wait state reflects its waiting time. The total waiting time during a blocking synchronization equals the total duration of all wait states on all participating streams. In a non-blocking synchronization, the total waiting time is the sum of all unsuccessful test operations (compare Section 4.1.1 – *Early Device Synchronization*). The blame of a wait state is defined as follows:

Definition 5 (Blame) *Blame reflects the duration of a wait state at its causes. In a synchronization, the value of blame equals the accumulated waiting time in all wait states over all participating streams. The process of assigning blame to causes, e.g. to execution streams, program activities, or tasks, is called blame shifting.*

As shown in Figure 5.2, wait states and blame can be further distinguished. Local blame is assigned to the causes of *local wait states*, while *global blame* is assigned to the causes of *global wait states*. When a wait state is blamed, it propagates the blame to its cause and is therefore called a *propagating wait state*.

Synchronization Interval

The direct cause of a parallelization wait state is an execution imbalance between two or more execution streams in a synchronization interval. Such an interval emerges by a synchronization and includes all streams that are involved in the synchronization. It is used to limit the search for the cause of a wait state to a time frame. For SPMD parallelization, it also defines the time interval, for which runtime data of program regions has to be passed from waiting streams to the blamed stream (see Section 5.3.2 – *Blame SPMD-parallelized Program Regions*).

Synchronization intervals occur within a programming model and are determined depending on the parallelization model. For SPMD parallelization, it emerges between two consecutive synchronization points of the same programming model and the same involved streams. In case of MPI, a synchronization point is induced by a blocking MPI operation. Synchronization points in OpenMP are the begin of OpenMP parallel execution (begin of implicit tasks in the same parallel region) and OpenMP barriers. They might also occur at OpenMP locks, and OpenMP critical regions, if a thread has to wait before it can acquire a lock or enter a critical region, due to another thread possessing the lock or executing the critical region. In hybrid programs, synchronization intervals of different programming models can overlap.

In task-based models such as computation offloading, the end of a synchronization interval is the end event of a wait operation. The interval begin is determined by following dependent tasks along the critical path backwards in time, disregarding inter-stream dependencies distinct from the wait's programming model. It is the first task trigger that is encountered (along the path) on a stream that is not associated with the wait state. The latter condition is implicit for computation offloading. Typically, task trigger (interval begin) and synchronization operation (interval end) are on the same stream.

5.3.2 Blame the Cause of Wait States

Blame quantifies the cause of wait states. It is shifted in spatial dimension to execution streams and in temporal dimension to program activities. Depending on the wait state's programming model either of the following two blame distribution approaches is applied. The distinction is made between SPMD and MPMD (or task-based) parallelization.

Blame SPMD-parallelized Program Regions

For SPMD parallelization, e.g. with MPI processes or loop parallelization via threading, blame is distributed in the synchronization interval on the stream that caused one or more streams to wait. The stream that enters the synchronization last (collective wait) or is not actively synchronizing (one-sided wait) is

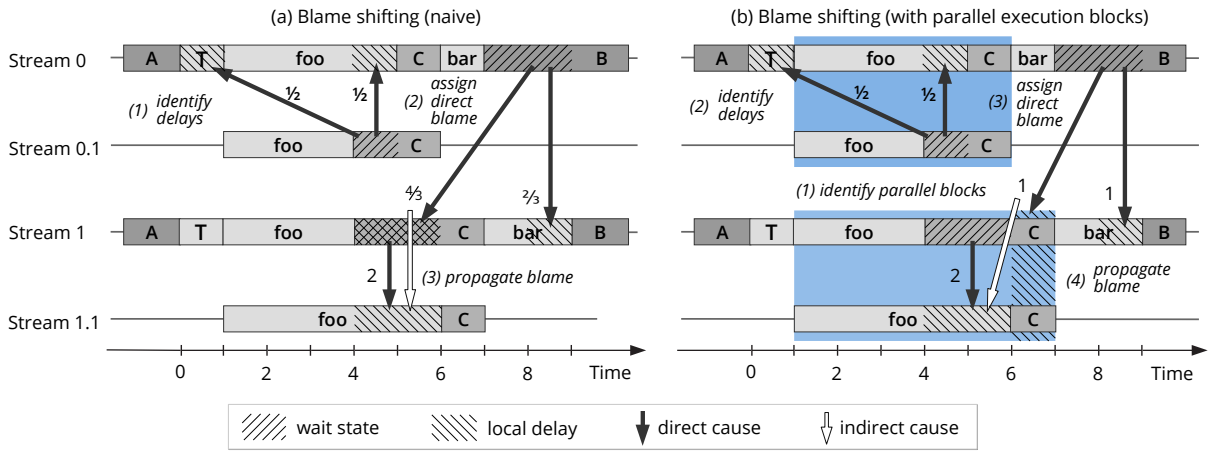


Figure 5.3: The root-cause analysis for SPMD-parallel execution is based on the detection of delays in unbalanced executed program regions. To better distinguish between local and global imbalances, the concept of parallel execution blocks has been introduced. The execution scenario is the same as in Figure 5.2. Absolute blame values were determined with Equation 5.1.

blamed. It is possible that several streams may be the last to enter the synchronization. If at least one further stream in the same synchronization has a wait state, several streams can be blamed for causing it.

Figure 5.3a illustrates blame shifting to activities in unbalanced executed program regions for a hybrid-parallel example. Similar to Boehme's delay-cost analysis, blame is distributed to so-called delays, which are identified by comparing the runtime of program activity between the waiting and the blamed stream. A delay is detected if the execution of a program region on the blamed stream exceeds its duration on the waiting stream. The comparison can be performed based on a profile or a trace. Call-path profiles are a compromise of both, which can be represented as time vectors and therefore easily compared. In traces, the delays are determined by comparing each occurrence of a program region between the streams.

In the synchronization interval, the total blame is apportioned to all activities with a delay on the blamed stream. The blame of such an activity a_i is determined by the total blame $blame_{total}$ (duration of the wait states also caused by a_i) multiplied with the share of the delay of the activity a_i in all delays $\{delay(a_1), ..., delay(a_n)\}$ (see Equation 5.1).

$$blame(a_i) = blame_{total} \cdot \frac{delay(a_i)}{\sum_{j=1}^{n} delay(a_j)} \quad , \; i = \{1, ..., n\} \wedge n \; ... \; number\ of\ activities \tag{5.1}$$

The comparison of profiles or traces between execution streams also enables the distinction between unbalanced execution of program regions and program activities that are executed on the blamed stream exclusively. The former, unbalanced execution as with the regions `foo` and `bar` in Figure 5.3, is the expected case for SPMD parallelization. Accordingly, the optimization goal is better balancing.

The maximum runtime gain RB_{max} by re-balancing the parallel execution of a program region within a synchronization interval is determined as shown in Equation 5.2. A perfect balancing is achieved if the waiting time is evenly distributed to all synchronized streams and therefore no stream has to wait. Consequently, the average duration d of the wait states $\{w_1, ..., w_m\}$ limits the runtime gain. Another limitation is the duration d of the blamed activity a_{blamed}, which represents the execution instance of region r on the blamed stream. In the best case it could be removed or moved to another stream. If program regions are exclusively executed on the blamed stream, their runtime should be reduced or a parallelization considered.

$$RB_{max}(r) = \min\left(d(a_{blamed}), \frac{\sum_{i=1}^{m} d(w_i)}{m}\right), \quad m \; ... \; number\ of\ synchronized\ streams \tag{5.2}$$

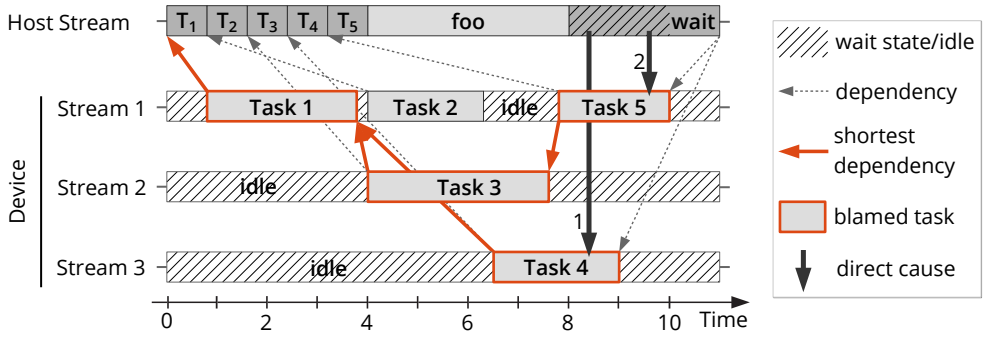


Figure 5.4: The root cause of wait states in computation offloading and tasking follows the shortest paths, starting at the last task that is associated to the wait operation in each execution stream. Dependencies between tasks on an execution stream are implicit and therefore not highlighted. Task 1 is blamed on two different paths.

Blame the Shortest Path

Parallelization models such as computation offloading and tasking focus on splitting the code into parts (tasks) instead of balancing a code part over similar compute resources (execution streams). It is assumed that different tasks can be executed on different execution streams at the same time. Therefore, it is not reasonable to consider the balanced execution of program regions between streams for such programming models and thus it is not necessary to compare runtime information between streams.

As shown in Figure 5.4, blame is distributed over tasks on the shortest paths along the dependencies in the synchronization interval, starting at each stream's last ending task that is directly associated to the wait state (see Section 4.1.1 – *Detecting Associated Device Tasks*). It is assumed that device tasks cannot start before the begin of their trigger operation but before its end.

Blame is determined similar to Equation 5.1, whereas a task can receive blame from different paths. Hence, a task a_i on a path is blamed according to its runtime share of all tasks on the same path. The total blame $blame_{total}$ that is distributed over the tasks on a path results from the overlap of the wait operation on the host with tasks on a device stream. The dependencies are either explicitly specified or can be determined implicitly from the properties of the programming models (compare Section 6.2 – *Offloading Dependencies*).

To reduce the wait state, the runtime of blamed tasks could be reduced, starting at the task that received the most blame. However, the scheduling of tasks and the revision of dependencies between tasks are also promising optimization approaches. On the other side, activities before wait states can be extended to reduce or eliminate the latter, e.g. activity *foo* in Figure 5.4.

The proposed analysis distinguishes between idle and actively waiting streams. Blame is shifted only in the latter case, since the cause of idle time can usually not be clearly determined. Any execution stream with access to the idle resource could be blamed for not keeping it busy, which would give a fuzzy result in many cases. Assuming that a mapping of stream to hardware was available, idle time and possibly blame could be more precisely determined. Compare Section 4.1.2, which discusses idle streams on offloading devices.

In the synchronization interval, the maximum runtime gain RO_{max} by optimizing a region that is only executed on the blamed stream can be determined with Equation 5.3. RO_{max} is limited by the shortest directly caused wait state w_i (among all associated streams) and the duration d of the blamed activity a_{blamed} itself.

$$RO_{max}(a_{blamed}) = \min(\{d(w_1), ..., d(w_k)\} \cup \{d(a_{blamed})\}),$$
$$k \text{ ... number of directly caused wait states} \quad (5.3)$$

5.3.3 Propagation of Blame

A wait state that causes another wait state propagates the received blame to its cause (see Figure 5.3a). As a result, a program activity can be assigned with local and propagated blame for directly and indirectly causing a wait state.

Böhme's delay cost analysis [BGWA10] also distinguishes between short- and long-term costs of delays, which corresponds to direct and propagated blame according to the notation used in this work. As optimizing the execution of a program region that received propagated blame does not necessarily affect the overall program runtime, the proposed analysis additionally distinguishes *global blame*, which denotes blame that has been directly assigned or propagated from a global wait state (see Figure 5.2b). The optimization of correspondingly blamed activities reduces the overall program runtime.

The implementation of blame propagation between MPI processes in a parallel trace analysis has been discussed in [BGWA10]. However, batching of MPI communication operations, as described in Section 5.2 as optimization, cannot be used with this implementation.

In hybrid programs, the cause of local and global imbalances can become blurred by propagating blame across programming models. The grouping of local streams in a synchronization interval to parallel execution blocks refines the propagation of blame and enables a distinction between local and global imbalances. The timelines in Figure 5.3 illustrate the effect of execution blocks on the blame distribution. Instead of distributing blame to individual activities of the stream that is associated with the wait state (timeline (a)), the entire execution block is considered a single activity (timeline (b)), which obviously affects the overall blame distribution in the synchronization interval. For the delay analysis, parallel execution blocks have to be compared, which can be done using key properties such as the associated programming model and the regions or tasks it contains.

5.4 Distributed Hybrid Critical-Path Analysis

The critical path is an important property of parallel program execution. Program activities on the critical path contribute to the overall program runtime and, thus, are valuable optimization targets. The detection of such activities in local graphs and from MPI programs has been discussed in Section 3.3.5. This thesis proposes a critical-path analysis for hybrid programs, which combines a scalable critical-path detection in a distributed graph with a fast critical-path algorithm for local graphs. The distributed graph is generated in parallel from an execution trace. A combination of distributed and local critical-path detection is depicted in Section 5.4.1. Since the critical path can also be determined on the basis of wait states, the findings of a previous root-cause analysis can be used. A respective weighting of the critical path using the results of the root-cause analysis is discussed in Section 5.4.2.

5.4.1 Combination of Distributed and Local Critical-Path Analysis

Although parallelization models such as message passing, threading, and computation offloading can be applied orthogonally within an application, the critical path cannot be detected independently for each model. This is mainly because it is not known whether the critical-path detection encounters a specific wait state. Figure 4.4 illustrates a situation, where the critical path is dominated alternately by MPI and computation offloading.

Consequently, the critical path cannot be detected hierarchically, as proposed in [SSD14]. It must consider wait states of any type when they are encountered during the analysis. Local and global analysis differ only in the handling of wait states, the principle of detection is similar. The proposed method is executed in two stages, a forward and a backward stage. Parallelization is performed as proposed in Section 5.2. Thus, each analysis process handles all events from its associated stream group and creates its own EDG from the program trace.

Forward Stage: Graph Preparation

In the forward stage, events are processed in chronological order and represented as vertices in the EDG. One EDG is created per execution stream as only vertices from events of the same stream are connected with edges. As described in Section 5.3.1, wait states are detected and dependencies extracted. The latter are added as edges to the EDG. The edge between the begin and end of a wait state is set to infinite time, whereas the newly added edges create paths without wait state. The critical path of an EDG is defined as follows:

Definition 6 (Critical Path in an EDG [SSD14]) *Let $G = (V, E, t, d)$ be an event dependency graph. A path of G is a sequence $\pi = (v_1, ..., v_n)$ of n events in V with $n \in \mathbb{N}$. If $n = 0$, π is the empty path. We further define $d_w : E \rightarrow \mathbb{R}_0^+$, which denotes the duration for which an activity edge is a wait state. Thus, $E_w = \{e \mid e \in E \wedge d_w(e) > 0\}$ is the set of wait states. The critical path π_c is the longest path in G without wait states $(v_i, v_{i+1}) \in E_w$. Its length $d(\pi_c) = t(v_l) - t(v_f)$ is determined by the difference between the timestamps of the temporally last vertex v_l and the temporally first vertex v_f.*

Several critical paths can exist in an EDG, e.g. if two parallel execution paths with the same duration do not contain wait states. In practice, two or more execution streams can be the last to enter a barrier at the same time and thus do not induce wait states.

Dependencies between processes are represented by remote edges, which connect several local EDGs to a global distributed EDG. A remote event is identified by an execution stream and an event identifier, which are exchanged between the analysis processes (see Section 5.2.2).

In case of wait operations on two or more execution streams, e.g. two-sided or collective MPI operations, there is a dependency between each end of a wait operation to all other wait begins or vice versa. For the pure critical path, only the edges to the begin of the wait operation without wait state are relevant. Thus, only these edges are kept respectively all others are discarded.

Backward Stage: Critical-Path Tracking

The backward stage is, due to the trimmed EDG, only a tracking of the critical path and thus faster than typical critical path methods. It starts at the temporally last event and follows events on the same execution stream (the only path in the EDG) backwards in time until a wait state (an edge with infinite weight in the EDG) is encountered. From the end event of a wait state, the critical path follows the only remaining edge to another stream.

The distributed analysis uses a master-slave concept. The analysis process that contains the temporally last event is the first master. All other processes are initially in slave mode and wait for a message. When the master process encounters a wait state from inter-process communication, e.g. an MPI wait state, a message is sent to the slave process without wait state, which is the new master, whereas the previous master changes to slave mode. The critical-path detection ends when the master reaches its temporally first node and, hence, has no more local events to process. Finally, the master sends a message to all other slaves that the analysis is completed.

5.4.2 Weighted Critical Path

Although being a symptom of unbalanced execution, the critical path does not quantify the imbalance. Nor does it provide information on a potential runtime gain by optimizing program regions. Besides the time that a region is executed on the critical path, an additional weighting can significantly increase its expressiveness.

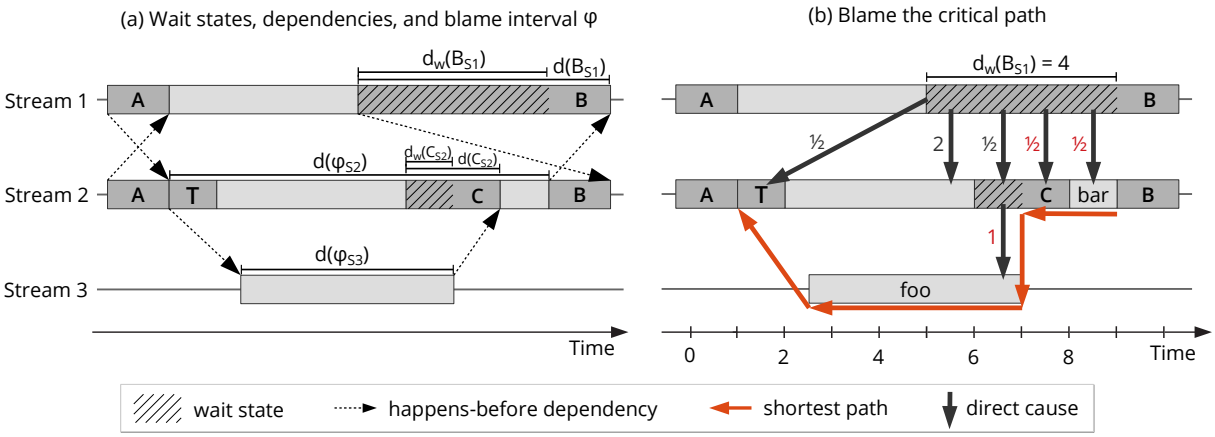


Figure 5.5: In simplified blame shifting, wait states do not propagate blame and balancing of program regions between streams is ignored. The exclusive runtime of synchronizing activities, the duration d_w an activity is a wait state, and the duration d of synchronization intervals φ_i are labeled in timeline (a). Timeline (b) shows absolute numbers for blame according to Equation 5.4 and highlights blame on the critical path with red color.

Blame-Weighted Critical Path

This thesis proposes to combine critical-path detection and root-cause analysis. Both analyses aim to detect the cause of imbalances and overlap up to the detection of wait states. In case of the proposed wait-state-based critical-path detection, each program region or task on the critical path inevitably causes a wait state and, hence, will also receive blame from the root-cause analysis presented in Section 5.3. The critical path focuses on wait states with impact on the overall program runtime, whereas the root-cause analysis detects the cause of all wait states and quantifies the imbalances. The combination of both analyses weights the critical path according to the amount of waiting time caused.

Simplified Blame-Shifting

Blame shifting can be a time-consuming and complex task. Therefore, a trade-off can be reasonable in terms of analysis accuracy. As blame on the critical path by definition has an impact on the total program runtime, it is obsolete to expose global blame (see Section 5.3.3). To further reduce the time and implementation effort required for the root cause analysis, the balancing of regions between streams can be disregarded. Moreover, a flat blame shifting without blame propagation can significantly reduce the analysis time.

Figure 5.5 illustrates the simplified blame shifting for two parallelization layers. Blame is shifted from a wait state to the event on the stream that is connected with a dependency edge and distributed along the critical path according to the runtime share of activities or tasks in the synchronization interval $d(\varphi)$. The amount of blame to be distributed to an activity a equals the duration of all directly caused wait states $\{w_1, ..., w_k\}$ with k being the number of wait states. $d(a)$ represents the exclusive time of the activity a and $d_w(a)$ the duration for which activity a is a wait state itself. Blame b for an activity a is then computed as shown in Equation 5.4.

$$b(a) = \sum_{i=1}^{k} d(w_i) \cdot \frac{d(a) - d_w(a)}{d(\varphi)}, \; k \text{ ... } \textit{number of directly caused wait states} \tag{5.4}$$

In practice, the share of the wait operation that is not a wait state at the same time is most often very short. Consequently, the accuracy of the analysis is hardly reduced when the critical path changes to another stream at the end of wait operations. The simplification eliminates $d_w(a)$ in Equation 5.4.

Figure 5.5 (b) shows the simplified blame shifting using absolute numbers. *foo* is assigned with 1 and *bar* with ½ blame on the critical path. All other activities are not on the critical path and therefore their blame is ignored. Blamed wait states such as C do not propagate blame. The trace analyzer CASITA (see Section 6.3), which has been implemented in the context of this thesis, uses this simplified blame shifting to reduce the analysis time. Blame propagation can be used optionally.

Runtime Reduction Potential

Activities on the critical path can also be weighted according to their potential to reduce the overall program runtime. A main difference to the approach discussed beforehand is that the runtime reduction potential is determined per wait operation and not distributed according to the runtime share of activities. Obviously, the upper limit for runtime reduction is the runtime of the activity itself.

A simple approach considers the closest near-critical path for each wait operation and synchronization interval respectively. Hence, an activity or task on the critical path only needs to be optimized until the former closest near-critical path manifests as the critical path. The respective runtime reduction of an activity is derived from the duration of the shortest wait state. Equation 5.3 reflects this metric, which in detail specifies the maximum runtime gain by optimizing an activity on a specific stream in a given interval. This derived runtime reduction potential is a reasonable marker for program regions that are executed only on the critical path. Region execution that is balanced across streams should be rather weighted with the rebalancing potential as defined in Equation 5.2.

However, the proposed weighting metrics are only a heuristic approach that does not replace a performance prediction that considers the optimization of program regions in a global program context and not only for a synchronization interval. A graph-based performance prediction is proposed in [SDJ16].

Deceleration Potential

In contrast to the runtime reduction potential, the deceleration potential highlights program activities that are not on the critical path and could therefore be decelerated without increasing the duration of the program. Such activities are weighted according to the waiting time that follows on their path. Thus, the activity with the longest subsequent waiting time, which is also most distant from the critical path, can be delayed the most. Applied to execution streams, this metric highlights those that have the smallest share of the critical path (primary criterion) and the most waiting time (secondary criterion).

The deceleration potential might be used for improving energy efficiency. An approach to control the CPU frequency for program regions has been presented in [SM14]. However, this might be more complex for SPMD-parallel execution of program regions, where the critical path is not always on the same execution stream.

6 A Framework for Systematic Performance Analysis

This chapter presents a performance analysis framework, which detects inefficiency patterns based on a set of extensible rules and performs a combination of critical-path and root-cause analysis as proposed in Section 5.4.2. As a prerequisite, the Score-P tools infrastructure had to be extended to enable a comprehensive analysis of scalable heterogeneous programs. The analysis workflow and the toolchain are presented in Section 6.1. Section 6.2 introduces the enhancements regarding data collection. The critical path analysis tool for heterogeneous applications (CASITA) [SSD14] has been developed to verify the applicability of the proposed analysis. It is described in Section 6.3. The presentation of the analysis results is depicted in Section 6.4. Section 6.5 describes the validation, scalability, and practicability of the analysis by CASITA.

6.1 Workflow and Toolset Architecture

According to the analysis layers that have been proposed in Section 3.2, the entire analysis workflow can be divided into three stages: program instrumentation, program execution with logging, and data inspection. Figure 6.1 summarizes instrumentation and program execution in the upper part under measurement, while tools for data inspection and data containers are shown in the lower part.

In the first stage, the program is instrumented with Score-P . The choice of instrumentation options (see Section 3.2.1 – Instrumentation, Tool Interfaces) determines the possibilities for subsequent analysis and thus their quality and significance. As a matter of principle, only events that have been recorded can be analyzed. Events that are relevant for the analysis have been described in Section 5.1.1. The extensions to Score-P are explained in Section 6.2.

In the second stage, the program has to be executed. It is important that the runtime environment is configured appropriately. Score-P allows several settings to be made using environment variables. For a useful analysis, a representative run is necessary. The result of the execution run is an OTF2 trace.

In the third stage, the program trace is inspected. To extract relevant information and highlight important properties of the program run, CASITA examines the trace. The integration of CASITA into the Score-P tools infrastructure has been proposed in [SDS15]. The analysis process with CASITA is unattended and generates an enriched OTF2 trace, a pattern summary, and a sorted region profile. To validate the analysis results or identify further inefficiencies, the OTF2 trace can be manually investigated with Vampir.

6.2 Extended Performance Data Collection

To enable a more comprehensive analysis of heterogeneous programs, the measurement infrastructure Score-P has been extended with focus on the adapters for computation offloading. Analysis requirements in terms of necessary events and additional information to rebuild inter-stream dependencies have been discussed in Section 5.1.1.

Offloading Dependencies

In order to associate tasks with their trigger operations, additional information must be recorded. In case of a task-trigger or stream-wait operation, the referenced stream (Score-P location) is stored as an OTF2 attribute attached to the respective region. For device synchronization (wait for all device streams) no

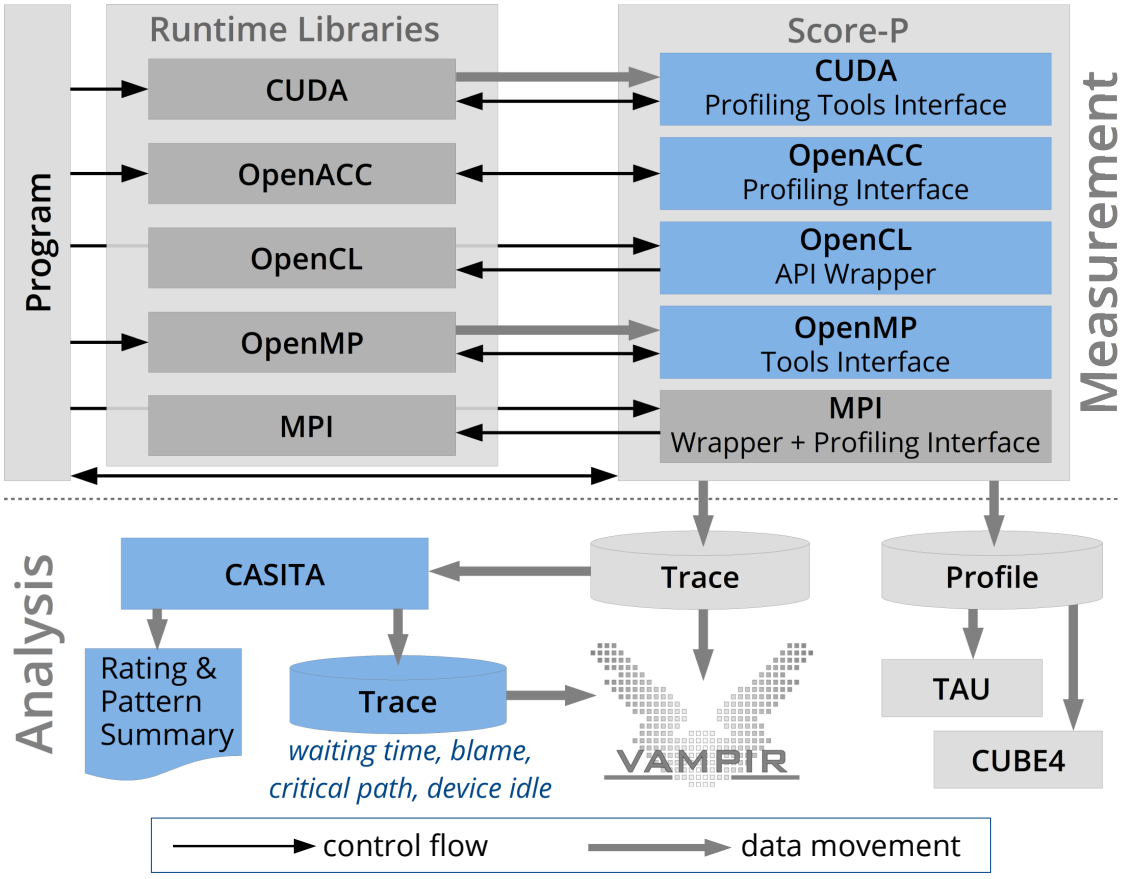


Figure 6.1: Interaction of the program, runtime libraries, the measurement tool Score-P, and the trace analyzer CASITA. Components that have been developed or extended as part of this work are highlighted (blue).

additional attributes have to recorded, because the streams that are associated with an offloading device are already stored in the OTF2 trace definitions.

CUDA provides an additional synchronization option with CUDA events. To support it, the target stream and an event identifier are added to the trigger operations of CUDA events. Event-synchronization and event-query operations are stored together with the event identifier, which indirectly provides the target stream through the associated event-trigger operation. For non-blocking synchronization (see Section 4.1.1) with event queries, the result of the query operation is also stored, as it enables the start and end of the synchronization to be determined.

Explicit task dependencies, e.g. between tasks on different device streams, require the operation that describes the dependency to be stored with additional OTF2 attributes, which identify tasks or events to wait for. Implicit dependencies between offloaded tasks are not identified during data collection. However, some can be detected during data analysis (see Section 6.3.3). For this purpose, the CUDA default stream is marked separately.

Score-P Offloading Adapters

Score-P supports three computation offloading APIs since version 3.0. In the context of this thesis, the CUDA adapter has been extended to enable recording of offloading dependencies. The OpenCL adapter has been implemented from scratch [DT15]. All API calls into the OpenCL library are intercepted and respective OTF2 events recorded. OpenCL events are attached to each offload task, to collect the begin and end time of OpenCL kernels and OpenCL data transfers.

The OpenACC adapter has been implemented based on the profiling interface that has been co-developed in the OpenACC consortium. It enables the execution begin and end of OpenACC directives as well as a few additional events to be recorded. Except for device memory allocations, only host-side events are captured (see Table 4.4 in Section 4.2.1). Offloaded tasks have to be collected via Score-P's CUDA or OpenCL adapter.

Score-P OMPT Adapter

As a potential replacement for the OPARI2 source-to-source instrumenter and to ensure the usability of the OMPT interface, an experimental OMPT adapter has been implemented in Score-P. As it is a part of the specification, the OMPT interface covers the full scope of OpenMP (since version 5.0), including the OpenMP target directives. This is an important advantage over OPARI2, even though an extension of OPARI2 by OpenMP target directives has been disucssed in [DSGS14].

OMPT events reflect the behavior of the OpenMP runtime. The Score-P adapter records these events based on the respective runtime callbacks. To enable the reconstruction of dependencies, OpenMP parallel and implicit-task regions are annotated via OTF2 attributes. OpenMP barriers are perfectly nested inside of parallel regions and therefore do not require any annotation. As with the other offloading APIs, target regions are annotated with a device identifier, which is used to describe a dependency to an offloading stream. Hence, target regions are handled similar to trigger operations for offloaded tasks.

6.3 A Scalable Trace Analyzer for Hybrid Programs

The parallel trace analysis has been implemented in CASITA, an MPI and OpenMP parallel C++ program, which is available as open source [DSS19]. It supports MPI, OpenMP threads, CUDA, OpenCL, and OpenMP offloading. The limitation to these programming APIs is mainly due to the OTF2 trace format, which does not provide a sufficiently generic data representation as described in Section 5.1.1. Internally, CUDA and OpenCL calls are abstracted into a generic offloading representation. OpenACC is indirectly supported based on the CUDA and OpenCL support.

The parallelization is implemented as proposed in Section 5.2, which allows the analysis to be performed directly after the measurement run with the same resources. CASITA implements a critical-path analysis in combination with a simplified root-cause analysis as depicted in Section 5.4.2. Furthermore, it detects the inefficiency patterns that have been described in Section 4.1 as well as common wait patterns in MPI and OpenMP parallelization. CASITA can be easily extended by adding new analysis rules. Currently implemented rules are described in Section 6.3.2.

6.3.1 Trace Analysis Procedure

The parallel trace analysis is performed in four stages, each with several subtasks. Technically, it was possible to perform stage one and two together, since the events are read in chronological order and the analysis could be implemented without a look-ahead. By detecting wait states, parts of the critical-path and root-cause analysis overlap. Each MPI process executes the following core part of the analysis:

1. Read the input trace
    - Read all events from associated streams (stream group)
    - Create a local EDG that contains only analysis-relevant events
2. Apply the analysis rules
    - Iterate over events (vertices)
    - Add dependency edges (also remote edges to other stream groups)
    - Detect wait states and determine waiting time
    - Distribute blame (via graph backtracing)

3. Critical-path detection
   - Check if the local stream group has the globally last event (to determine the end of the critical path and the initial master)
   - Track the critical path within the stream group until a remote edge (master)
   - Wait for a signal to continue as new master or to stop critical-path tracking (slave)
4. Generate analysis output
   - Read all events from associated streams (again) and move along the events of the EDG
   - Generate the analysis metrics from waiting time, blame, and the critical path
   - Determine the analysis metrics for previously unobserved regions (such as CPU functions)
   - Summarize the analysis metrics for each program region
   - Write the events back to the output trace and add new records/metrics

Finally, the root rank collects the metrics from each other rank to create the pattern summary and the optimization guidance profile.

CASITA uses two methods to reduce the main memory requirements of the analysis. The first uses the property that the analysis can start at global collectives (see Section 5.2). Hence, the complete analysis can be performed whenever a global collective is encountered, the results stored, and the EDG then discarded. The second method takes advantage of the fact that only certain program regions are needed for the analysis. Therefore, the trace is read twice. During the first reading, only required events are included in the EDG and analyzed. This saves memory, especially for programs with many small functions. When the trace is written out, it is read again and the analysis results are applied to all regions.

6.3.2 Analysis Rules

CASITA uses analysis rules to reconstruct event dependencies and detect execution patterns in the EDG. Rules are categorized into programming models. Each rule defines the event that triggers it. A rule is not necessarily assigned to an inefficiency pattern and, thus, it can be used in the detection of multiple patterns. Adding rules allows the detection of new patterns without the need to change the core analyses. Thus, they are an extension interface for CASITA, which detects the following inefficiencies so far:

MPI	• late sender/receiver • (unbalanced) blocking collective
OpenMP	• (unbalanced) barrier
Offloading	• early offload synchronization (event-, stream-, and device-based) • (compute) idle device • consecutive transfers

Many inefficiencies are based on wait states. To detect them, a respective rule (depending on the programming model) is triggered whenever the leave event of a wait operation is processed. By applying the rule, the wait operation is either marked as wait state by adding a blocking edge, or it receives blame, which is distributed via a graph walkback. Depending on the wait's programming model, it is a simple stream walkback, or requires shortest path backtracking. Wait states are assigned with waiting time, whereas synchronization operations on offloaded tasks that are already completed are assigned with blame. For the pattern summary, each occurrence of an inefficiency is counted and its severity (caused waiting time) added up.

Eight rules are implemented to cover MPI collectives and two-sided communication. Three rules are used to expose imbalances in OpenMP barriers. Finally, there are five generic offloading rules, which cover early blocking device synchronization, device idle, task dependencies, and consecutive transfers, as well as four additional rules, which handle synchronization via CUDA events. The rules can be applied in different phases of the trace analysis: during graph construction (while reading the input trace), in an extra analysis phase, or during trace writing. Currently, most rules are applied in an extra analysis phase.

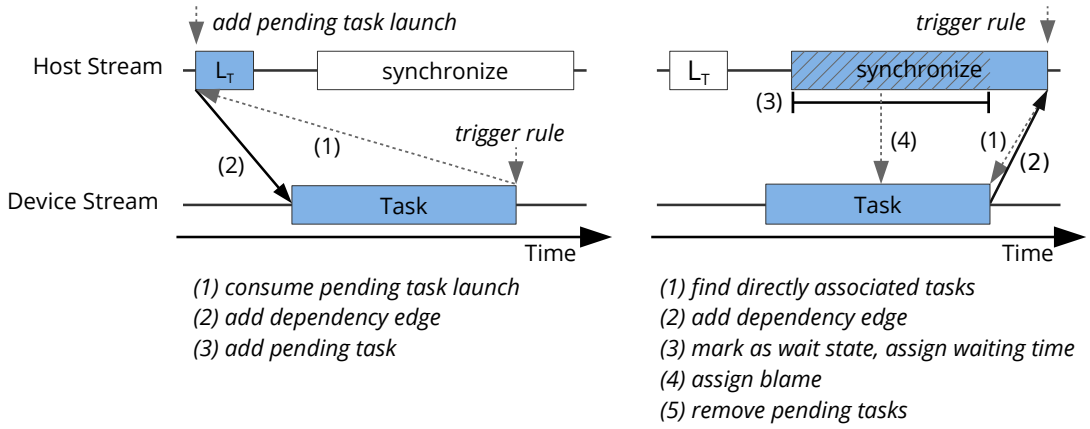


Figure 6.2: A pending task launch is the prerequisite for the offload execution rule (left). It represents a task launch begin event that has not yet been connected with the associated task begin event. A pending offloaded task has not yet been synchronized. It connects both rules. Steps (2) to (4) of the synchronization rule (right) are only performed, if at least one associated task was found. Otherwise, the synchronization operation is blamed for being late.

The offload execution rule is the basis for the detection of all offloading inefficiencies. A prominent inefficiency in computation offloading is detected by the synchronization rule. Figure 6.2 describes both rules. The synchronization rule is illustrated in the usual form, where the offloaded task is blamed. It is further simplified by considering only one task on one device stream. Blame distribution in case multiple tasks are synchronized is depicted in Section 5.3.2. Event-based and non-blocking synchronization are more complex to detect and therefore covered in extra rules.

6.3.3 Implicit Dependencies between Offloaded Tasks

CASITA implements a rule to identify implicit task dependencies on an offloading device. Dependencies between offloaded tasks are implicit if the tasks are executed on the same device stream. Further dependencies emerge from the resources available on the device, e.g. if a compute task cannot execute because another compute task on the same device (but a different stream) occupies required resources. In the latter case, a potential delaying task is compared with the trigger of the considered task and the temporally closer one is considered a compelling dependency.

CASITA implements an additional rule to detect dependencies between partly overlapping tasks. Such dependencies can occur, for example, when a compute task no longer needs all resources at the end of its execution and thus another task can already start. Their detection is a prerequisite for the correct detection of the critical path. However, CASITA uses reverse edges (the end of the edge is temporally before its beginning) to connect overlapping tasks, which overestimates the length of the critical path.

A dependency between two partially overlapping tasks is assumed if no task is executed completely parallel to the other, and the task that starts and ends later is delayed. A task is definitely delayed, if its trigger operation starts before the delaying task. A task is also assumed to be delayed, if it starts long after its trigger, where “long” can be a system-specific fixed time span or application-specific, e.g. ten times longer than the shortest time between trigger and task start.

CUDA’s default-stream semantic introduces additional dependencies. The default stream is an implicit device stream that is used when a task is launched without specifying the target stream. In the legacy mode, which is also the default up to CUDA 9.2, the default stream waits on all other blocking streams in the same CUDA context before it executes a task. Furthermore, it does not allow any other task to be executed in parallel. Hence, any task that is executed in the default stream acts as a barrier, which creates dependencies to all previously and subsequently triggered device tasks in the same CUDA context.

6.4 Presentation of the Analysis Results

CASITA presents the analysis results in the form of a tabular output, the pattern summary and the guidance profile, as well as an OTF2 program trace with additional records. The trace stores waiting times, blame, device idle, and the critical-path metric.

6.4.1 Pattern Summary and Optimization Guidance Profile

The pattern summary gives an overview of waiting times and detected inefficiencies. It lists the total waiting time, waiting times by programming model, and the severity of individual inefficiencies classified by programming model. It can thus be used to estimate the time in which computing resources are unused or wasted, and exposes the most severe inefficiencies. Finally, it allows an estimation of the optimization potential of the program. Listing 1 shows a pattern summary for an execution of the LSMS code, which is discussed in Section 6.5.3.

The optimization guidance profile provides a starting point for optimization. It rates regions according to the assigned blame on the critical path. If this metric is equal for two program regions, further criteria are used in the following order: (1) critical-path time, (2) total blame, and (3) exclusive runtime. The secondary criteria are also reported in the summary output to enable an analysis according to another rating order. Additionally, blame is divided into the underlying inefficiency patterns, e.g. to distinguish the extent to which a region is responsible for device idle or a late send operation. An example of CASITA's guidance profile is given in the appendix in Listing 2. The critical-path time of a region is also the upper limit of the total runtime gain, which can be achieved by optimizing this region.

The rating has the known weaknesses of a program profile, as it provides only aggregated values over the program execution and cannot show dynamic runtime effects. Hence, artifacts in the execution might dominate the rating, e.g. a single call-path execution might have caused all blame on the critical path. However, such as scenario could also be detected automatically. The timeline visualization (see Section 6.4.2) can be used to validate the rating.

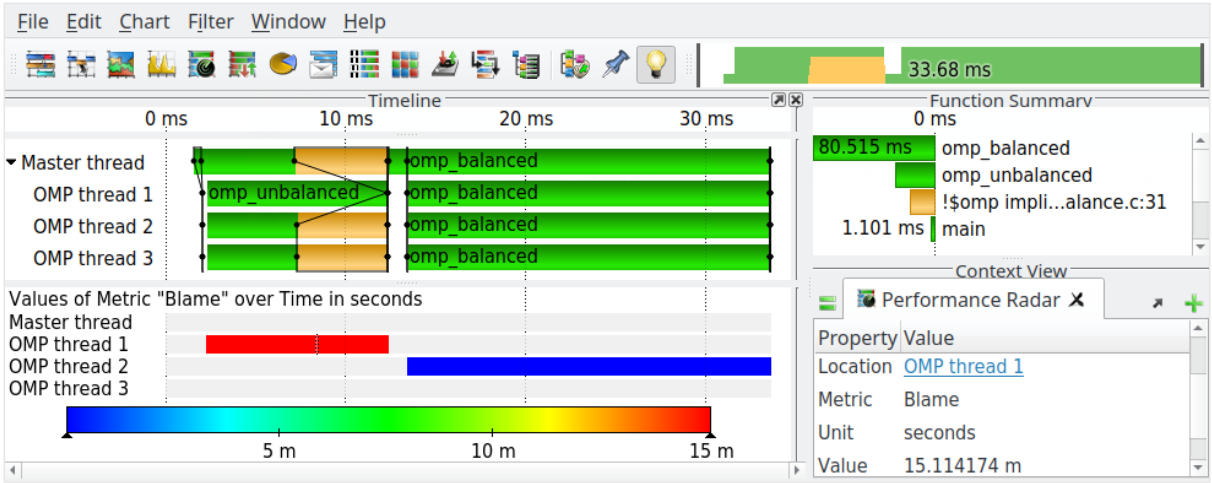
Currently, neither the pattern summary nor the guidance profile provide stream-aware analysis results. Consequently, it is not possible to highlight streams with particularly long waiting times or streams that dominate the critical path, which could expose stream-level balancing issues or hardware irregularities.

6.4.2 Timeline Visualization

The manual analysis of a timeline representation is a convenient approach for investigating details about the individual occurrences of inefficiencies and their manifestation over the time of a program. It also enables the recognition of patterns that have not been covered by automatic analysis so far.

OTF2 traces can be visualized with the Vampir trace browser, which provides several timeline displays with a powerful zooming capability. CASITA enhances the OTF2 input trace with additional OTF2 records to benefit from this timeline representation. It stores the blame metric and the critical path as OTF2 counters, the waiting time as an OTF2 attribute for each affected region, and device idle using OTF2 event records. Accordingly, blame and the critical path only become visible in counter displays, while device idle is directly visible in the *Master Timeline*. The waiting time is displayed in the *Context View* and can also be visualized as a counter using a derived metric. Figures 6.3, 6.4, and 6.5 show possible visualizations of the new metrics in Vampir.

To better integrate the analysis, a new display could be created for the critical path, which represents the activities on the critical path as a single horizontal bar. Compared to Vampir's *Master Timeline* and *Performance Radar*, such a display is inherently scalable with the number of execution streams. Additionally, blame on the critical path could be added to the *Function Summary* as a metric, making CASITA's guidance profile also available for sections of a trace.



Wait OpenMP barrier: 15.1 ms (3.8 ms on avg. per host stream, 6 occurrences)

Region	Runtime	Time on CP	CP Ratio	Blame	Blame Ratio	Blame on CP
omp_unbalanced	25.3 ms	10.1 ms	31.34%	15.1 ms	99.85%	15.1 ms
omp_balanced	80.5 ms	20.1 ms	62.67%	0	0	0

Figure 6.3: Blame an unbalanced OpenMP barrier. In the *Master Timeline* in the top left, the imbalance can be identified by the color coding of the program regions. The *Performance Radar* at the bottom left highlights the assigned blame. The *Context View* at the bottom right shows the blame assigned to `omp_unbalanced`. The *Function Summary* in the top right lists the exclusive runtimes of the regions. Relevant CASITA analysis results are shown at the bottom.

6.5 Application of the Analysis by CASITA

This section demonstrates the application of the presented performance analysis framework. Synthetic mini-programs are used to validate its fundamental features, such as the detection of inefficiency patterns as well as the distribution of blame and the detection of the critical path. HPL and its CUDA version HPL-CUDA are used to examine the scalability of the analysis. The significance of the analysis as well as the applicability to a complex heterogeneous program is shown with the real-world application LSMS.

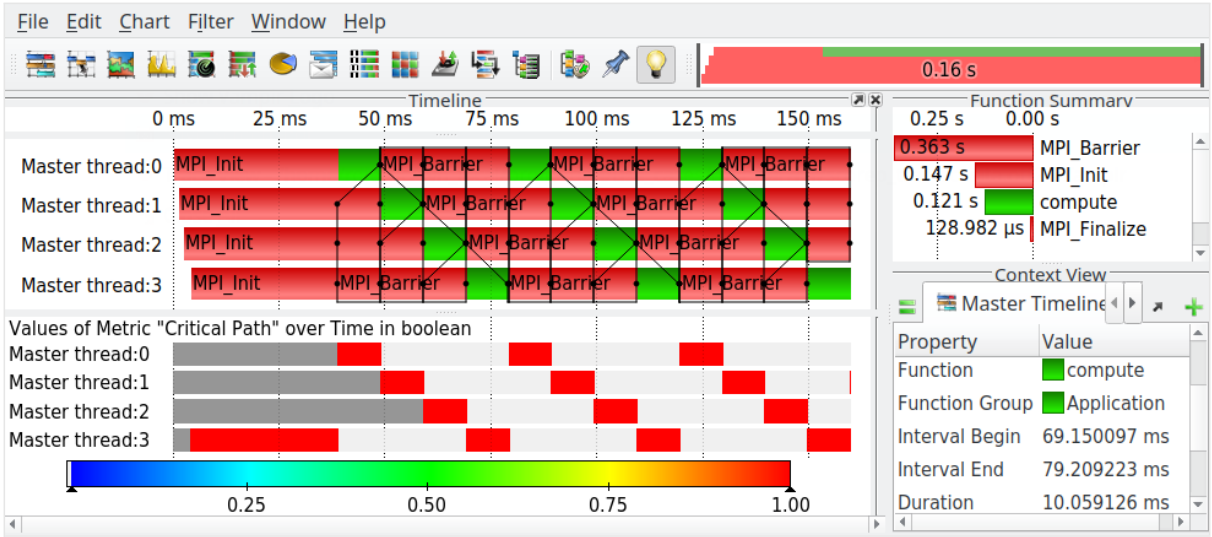
6.5.1 Synthetic Programs

Three program codes with OpenMP, MPI, and CUDA show the detection of unbalanced execution, regions on the critical path (in the tables abbreviated with CP), and the two main offloading inefficiencies host wait and device idle. The `sleep` function[1] was used to achieve predictable region execution times and, thus, to generate specific inefficiencies. The analysis results from the pattern summary and the optimization guidance profile are additionally validated by visualizing CASITA's result trace with Vampir.

Blame Unbalanced Execution

The detection of imbalances is one of the essential aspects of the proposed analysis and its implementation in CASITA. For illustration purposes, we compare an unbalanced and a balanced execution of an OpenMP-parallel region with four threads. Figure 6.3 shows the respective Vampir visualization. The unbalanced OpenMP-parallel region runs 10 ms, where one thread takes twice as long as the others. As a result, three threads are waiting for about 5 ms in the implicit OpenMP barrier at the end of the parallel region. In practice, such an imbalance occurs, for example, when loop iterations are not distributed evenly across the threads.

[1] sleep waits at least the given time



Wait in MPI collective: 371.16ms (92.79ms on avg. per rank; 42 overall occurrences)

Region	Runtime	Time on CP	CP Ratio	Blame	Blame Ratio	Blame on CP
compute	120.7 ms	120.7 ms	77.63%	362.0 ms	99.96%	362.0 ms
MPI_Init	147.5 ms	34.6 ms	22.26%	0	0	0
MPI_Barrier	362.5 ms	0.1 ms	0.06%	0	0	0

Figure 6.4: Critical-path visualization with Vampir. The *Master Timeline* in the top left and the *Function Summary* in the top right present compute regions in green and MPI collectives in red. The *Context View* shows details on a compute region. The *Performance Radar* visualizes the critical path in red. CASITA's pattern and profile output are shown below.

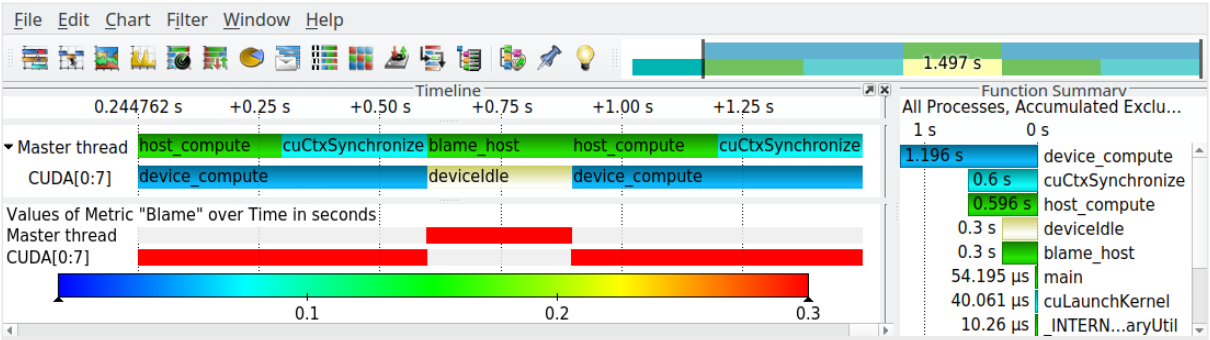
CASITA detects the imbalance and blames *OpenMP thread 1* with about 15 ms. The values from Vampir's *Function Summary*, *Performance Radar*, and the associated *Context View* are consistent with CASITA's output profile in the table below and show the expected result. The pattern summary also quantifies the only inefficiency pattern in this example with the expected value of 15 ms.

This example also shows a characteristic of CASITA's optimization guidance profile, which rates the program regions by *blame on the critical path*. In contrast to other profiles, it highlights the unbalanced region as most valuable optimization candidate, although the duration and even the critical-path share of the balanced region is much higher. The full blame of a barrier is assigned to the previous regions on the thread that last enters the barrier. In this execution scenario, these regions are on the critical path, which is why *blame* and *blame on the critical path* have the same value. The blue bar in the *Performance Radar* (zero values are in light grey) also shows that the second OpenMP-parallel region is not perfectly balanced. However, the imbalance is negligible.

Detect Regions on the Critical Path

The correct detection of the critical path is shown by the example of a round robin scheduling with four MPI processes. Starting with MPI rank 0 only one process computes, while the others are waiting in an MPI barrier. Accordingly, the critical path should always be on the computing process. In the example, each process is granted three times 10 ms computing time, which results in a total computing time of about 120 ms.

Figure 6.4 illustrates the described execution with Vampir. As expected, the *Master Timeline* shows three times four consecutive steps. The *Performance Radar* below shows the same pattern for the critical-path counter generated by CASITA. The critical path is obviously detected correctly and contains besides the compute regions only `MPI_Init`, `MPI_Finalize`, and the `main` function. The latter two, however, are negligible in terms of critical-path and runtime. The critical path ends in the `main` function before `MPI_Init` of process 3.



0.30 s idle device, 0.60 s early blocking device synchronization

Region	Runtime	Time on CP	CP Ratio	Blame	Blame Ratio	Blame on CP
device_compute	1.20 s	1.20 s	69.73%	0.60 s	66.6%	0.60 s
blame_host	0.30 s	0.30 s	17.43%	0.30 s	33.4%	0.10 s

Figure 6.5: Analysis of device idle and early blocking device synchronization. The *Master Timeline* in the top left shows the temporal program flow on host and device. The *Performance Radar* below visualizes the blame metric. The *Function Summary* in the top right lists the exclusive region runtimes. The main results of the CASITA analysis are displayed at the bottom.

The CASITA optimization guidance profile confirms the Vampir visualization. The region `compute` is for its entire runtime of 120.7 ms on the critical path. The assigned blame corresponds roughly to the runtime of the MPI barriers, which is a little longer, because the remaining execution time of the barrier, after the last process entered, is not accounted as blame. The regions in the profile are sorted primarily according to the blame on the critical path, which is why `compute` is listed despite the shortest exclusive runtime before the MPI regions. The second sorting criteria is the *time on the critical path*, which ranks `MPI_Init` before `MPI_Barrier`.

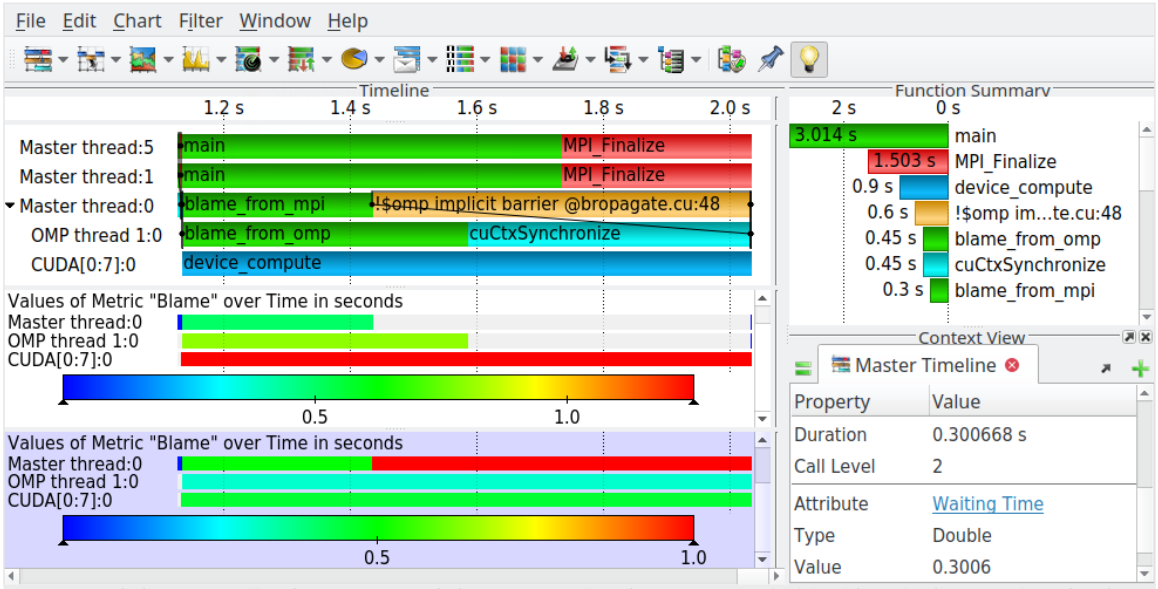
The only detected type of inefficiency pattern is *wait in MPI collective*. Since with four processes each process has to wait three times per loop, the waiting time in this example must be at least 90ms per process. CASITA determines a pattern severity of 371.16 ms in total and 92.8 ms on average per rank. Besides the waiting time in `MPI_Barrier`, it also includes the waiting times from `MPI_Init` and `MPI_Finalize`.

Device Idle and Host Wait

Device idle and blocking host synchronization are prominent offloading inefficiencies. Figure 6.5 shows a respective Vampir visualization. The program executes a single host thread, which triggers a CUDA kernel with a duration of 0.6 s, sleeps 0.3 s, and then waits 0.3 s for the completion of the device kernel. Subsequently, the device is idle for 0.3 s, while the host thread executes the function `blame_host`. To end the device idle, the host thread then starts another CUDA kernel instance, sleeps 0.3 s, and waits for the kernel to complete.

Figure 6.5 visualizes the course of the critical path and the assigned blame. As expected, the CUDA kernels are blamed for the waiting time during the device synchronization, which is 0.3 s for each kernel. The host thread is blamed for the device idle time between the kernel launches. Since the affected functions `host_compute`, `cuCtxSynchronize`, and `blame_host` have roughly the same duration, each is assigned with one third of the total blame (about 0.1 s). Since the device synchronizations generate wait states, the critical path passes through both kernel instances. Between the launch of the second kernel and the synchronization of the first kernel, the critical path is on the host.

According to CASITA's profile output, the CUDA kernel `device_compute` is the most valuable optimization candidate. It is for its entire runtime on the critical path and receives the most blame, the latter completely for keeping the host waiting. The second optimization candidate is the function



1.53 s wait in MPI collective, 0.60 s wait in OpenMP barrier, 0.45 s early blocking device synchronization

Pos.	Region	Runtime	Time on CP	Blame	Blame on CP
1	device_compute	0.90 s	0.90 s	0.85 s (48.5%)	0.85 s
blamed for: 52.9% (0.45 s) host wait, 35.3% (0.30 s) OpenMP barrier, 11.8% (0.10 s) MPI collective					
37	blame_from_omp	0.45 s	0	0.80 s (31.1%)	0
blamed for: 62.5% (0.50 s) MPI collective, 37.5% (0.30 s) OpenMP barrier					
38	blame_from_mpi	0.30 s	0	0.50 s (19.45%)	0
blamed for: 100% (0.50 s) MPI collective					
1	device_compute	0.90 s	0.90 s	0.45 s (17.5%)	0.45 s
37	blame_from_mpi	0.30 s	0	0.50 s (19.4%)	0
38	blame_from_omp	0.45 s	0	0.30 s (11.7%)	0

Figure 6.6: Blame propagation across programming APIs. The *Master Timeline* in the top left shows an imbalance which propagates from CUDA over OpenMP to MPI. The *Performance Radar* below highlights the regions causing it. MPI rank two to four are filtered in the timeline visualization. The *Function Summary* (top right) shows the exclusive region runtimes. The *Context View* (bottom right) shows the duration and waiting time of the selected `MPI_Finalize`. An excerpt from CASITA's analysis summary is shown at the bottom. The lavender background highlights blame shifting without blame propagation.

`blame_host`, which receives 0.1 s blame while being on the critical path. CASITA has identified both types of inefficiencies: idle device and early blocking device synchronization. The latter causes about 0.6 s waiting time on the host and turns up again as a blame for `device_compute`.

Blame Propagation

Propagating blame across wait states is an essential feature to identify the root cause of imbalances, especially in heterogeneous programs. However, it can significantly increase the analysis time, which is why it is optional in CASITA. The Vampir visualization in Figure 6.6 shows an example program that illustrates the effect of blame propagation in comparison to flat blame shifting. The synthetic test case executes a device kernel with a runtime of about 0.9 s. The MPI ranks 1 to 5 sleep 0.6 s before they run into `MPI_Finalize`. MPI rank 0 executes an OpenMP parallel region with two threads, where the master thread sleeps for 0.3 s before it runs the implicit barrier of the region. The worker thread sleeps 0.45 s before it starts to wait for the device and runs into the barrier afterwards. The most severe inefficiency pattern is the imbalance in the MPI collective `MPI_Finalize`.

Software Environment	Hardware Environment (per node)
CUDA 9.2.88, Intel Parallel Studio XE 2018.1.163, OpenMPI 2.1.2	4 x NVIDIA K80 GPUs (each with 12 GB GDDR5 RAM), 2 x Intel Xeon E5-2680v3 CPUs @ 2.5 GHz, 64 GB RAM, Infiniband (FDR)

Table 6.1: Hardware and software environment for HPL-CUDA experiments

Figure 6.6 also shows the result of blame shifting from an MPI imbalance via an OpenMP barrier and a device synchronization to a CUDA kernel. The latter receives direct blame of $0.45\,\mathrm{s}$ for the blocking device synchronization. It is additionally blamed for half of the waiting time from the OpenMP barrier ($0.6\,\mathrm{s}/2 = 0.3\,\mathrm{s}$), as the device synchronization takes half of the execution time of *OMP thread 1:0*. The OpenMP barrier itself is blamed with two thirds of the waiting time in `MPI_Finalize` ($0.2\,\mathrm{s}$), half of which is propagated to the device synchronization and further to the CUDA kernel. The latter is assigned in total with $0.45\,\mathrm{s} + 0.3\,\mathrm{s} + 0.1\,\mathrm{s} = 0.85\,\mathrm{s}$ blame.

The CUDA kernel is on the critical path for its entire runtime and thus also receives $0.85\,\mathrm{s}$ blame on the critical path. With about 12%, only a small fraction of its total blame is propagated from the MPI imbalance. The host regions *blame_from_mpi* and *blame_from_omp* have no share in the critical path and are therefore to be found at the end of the ranking. Their optimization would only increase the waiting time in the OpenMP barrier and the device synchronization.

In the case of flat blame shifting (highlighted with purple background in Figure 6.6), the waiting time in `MPI_Finalize` is shifted as blame to rank 0. The implicit barrier region receives with $1\,\mathrm{s}$ the most blame, followed by the host region *blame_from_mpi* with $0.50\,\mathrm{s}$ of blame. The device kernel receives its $0.3\,\mathrm{s}$ blame for the device synchronization, while the host region *blame_from_omp* receives $0.3\,\mathrm{s}$ blame for the OpenMP barrier imbalance.

6.5.2 High-Performance Linpack

For a trace-based analysis, it is expected that its performance will depend on the number of processed events. In general, more events should increase the analysis time, wherein events which are part of an inefficiency pattern have a much greater influence as they require additional processing. With regard to the scalability of CASITA, the MPI communication pattern of the original program is crucial as the MPI communication is reenacted. Since each MPI communication is additionally performed in reverse direction, CASITA executes about twice as many MPI transfers as the original program and requires further communication to track the critical path. Therefore, in the worst case the analysis may exceed the runtime of the original program. For compute-bound programs, though, the analysis should be significantly faster. Finally, the scalability also depends on the distribution of the events over streams, whereby an even distribution enhances load balancing.

HPL-CUDA: Analysis Scalability

To prove these assumptions, the CASITA analysis has been applied for HPL-CUDA [Fat09], a publicly available CUDA version of the High-Performance Linpack (HPL) [DLP03] benchmark, which maps processes to GPUs one-to-one. The experiments have been run on the GPU island (phase 2) of TU Dresden's Taurus cluster (see Table 6.1). Score-P 4.1 was used to generate the trace files. To keep the size of the trace files small, the compiler instrumentation was switched off and some regions were instrumented manually instead.

The input data file *HPL.dat* (see Appendix, Listing 3) changes only in three parameters between the experiments: the order of the coefficient matrix (problem size), the number of process rows (P), and the number of process columns (Q). By factorizing the number of processes, P and Q are assigned values that are as close as possible to each other. For factors of different size, P is assigned the smaller value.

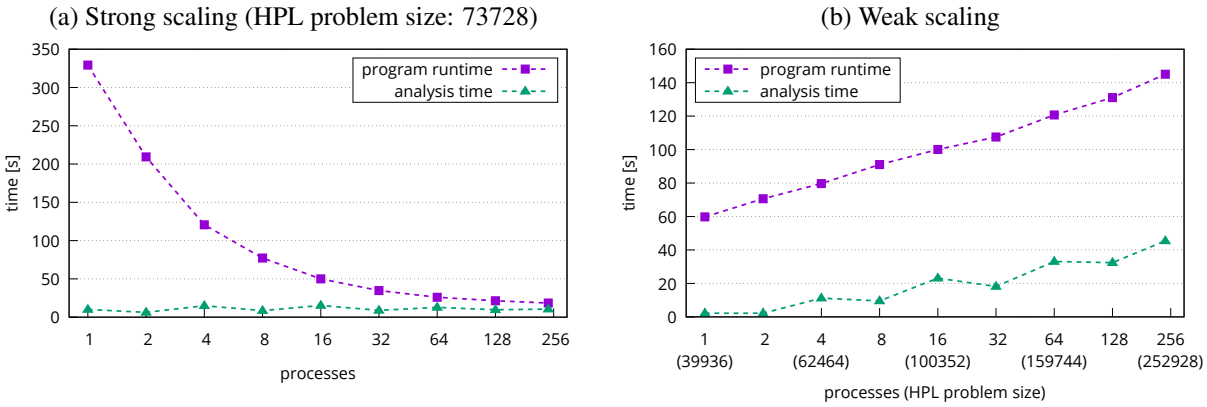


Figure 6.7: HPL-CUDA – Benchmark time vs analysis time

A blocking factor of 1024 turned out to provide decent performance on NVIDIA K80 GPUs for most matrix sizes. As the perfect scaling of HPL is not the focus of this work, the MPI implementation has not been optimized for different problem sizes either. It can therefore be expected that the HPL benchmark will not achieve optimal performance for all runs, but still acceptable scaling.

To investigate the scalability of the analysis, strong scaling and weak scaling measurements were performed on 1 to 240 GPUs (60 nodes). The processes were distributed evenly over the compute nodes and pinned so that a GPU local to the CPU socket is used. As the size of the problem increases, so does the number of generated trace events. For strong scaling, the problem size has been set to $72 * 1024 = 73728$, which requires almost the entire memory of a single node. For weak scaling, the base problem size (for one process) is $39 * 1024 = 39936$.

Since HPL basically performs matrix multiplication, a computational complexity $O(n^3)$ is assumed. In order to keep the workload per process approximately the same, the problem size $n(p)$ is determined based on the number of processes p and the base problem size n_1: $n(p) = n_1 * \sqrt[3]{p}$. To avoid short iterations at the end of the execution, the result was rounded down to an integer multiple by the blocking factor. The exact determination of the workload for HPL is more complex, which, however, has no influence on the results of the experiments regarding the analysis. Especially with HPL-CUDA the load balancing between CPU and GPU has to be taken into account (see [RdCL16]). Besides, the optimal benchmark configuration (via HPL.dat and the MPI implementation) can change with the problem size and the degree of parallelization.

The graphs in Figure 6.7 compare the HPL-CUDA benchmark runtime (time to solve the linear system) with the trace analysis time. As expected, the runtime gain of HPL-CUDA saturates with an increase in computing resources for strong scaling. The weak scaling scenario shows an increase in both benchmark runtime and analysis time, which is mainly due to the idealized assumption of the computational effort and the increasing parallelization overhead at scale. From the strong scaling experiments, it can be seen that the analysis time does not depend on the program runtime.

The graphs in Figure 6.8 visualize the correlation between analysis time and the number of trace events. Figure 6.8a confirms the assumption that the analysis time correlates with the number of trace events. In both strong scaling and weak scaling there are large variations in the number of trace events, which is mainly due to the HPL configuration. In the case of strong scaling, only the process grid is varied with the parameters P and Q. According to the measurement results, identical values of P and Q lead to more communication (compare [PWD+16] – FAQs) and thus more events that have to be analyzed. For strong scaling, the analysis time varies between 7 s and 17 s with a tendency to increase at higher scale, which is due to the increasing communication overhead. For weak scaling, the number of events per process tends to increase when more processes (and GPUs respectively) are used, which also increases the analysis time.

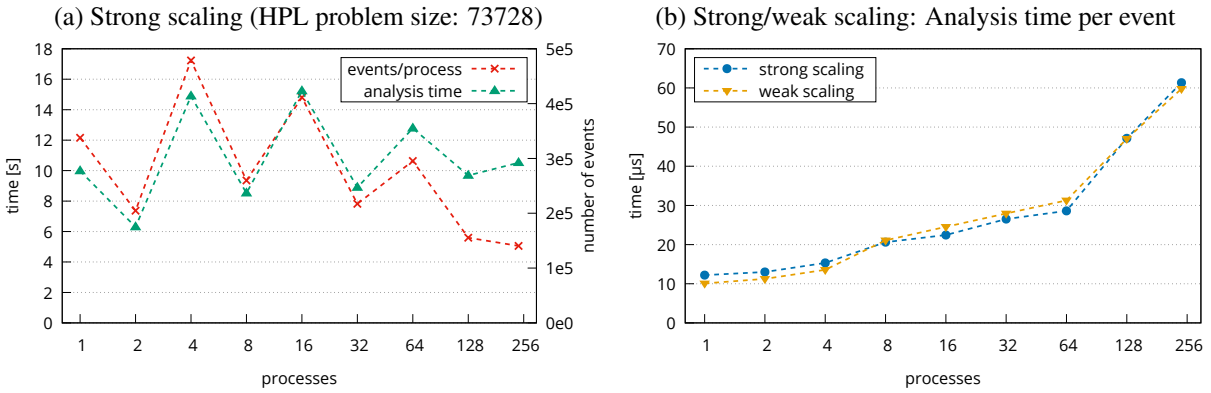


Figure 6.8: HPL-CUDA – Visual correlation between analysis time and number of trace events

Figure 6.8b shows the average analysis time required per event on each process (the analysis time multiplied by the number of processes and divided by the number of events). To exclude the influence of trace reading, which is performed twice during the analysis, it has been stripped from the values that are shown in this chart. Although in most cases, there is a significant difference between strong and weak scaling in the total analysis time, the analysis time per event is almost identical for the same number of processes. While this is in line with the expectations, it demonstrates a certain stability of the analysis time per event. For eight processes, the analysis times per event is almost identical, because the workloads have a similar size. With more processes, there is an increasing share of MPI events, whose processing takes, due to the inter-process communication, longer than for OpenMP and offloading events. From 64 to 128 processes, there is a significant increase of analysis time per event, as the share of applying the analysis rules suddenly increases by 10%. The reason for the latter, is a growing share of MPI events and a strong increase in the share of MPI collectives, which are executed at the beginning of the benchmark. To further investigate the analysis scalability over processes, measurements with HPL on up to 12288 processes were performed.

HPL-CUDA: Inefficiencies

Three inefficiencies provide a significant optimization potential: device idle, early device synchronization, and MPI late sender. Figure 6.9a and 6.9c show how the critical-path share and the severity of inefficiencies changes with strong scaling from one to 240 processes (and GPUs respectively). Due to the decreasing workload per GPU, the device idle time increases. This also shortens the synchronization time by the host. Correspondingly, the device's critical-path share decreases, which in turn causes an increase in the host's critical-path share. The MPI communication and its share of the total runtime increases with more processes. As it is not performed concurrently to device tasks, this causes an additional increase of the device idle time.

As shown in Figure 6.9b and 6.9d, a similar behavior of the inefficiencies can be observed for weak scaling, although the workload per GPU should remains constant. However, the increases and decreases are much smaller. The decreasing use of the device is primarily due to the increasing share of MPI communication, which includes waiting and communication time in late sender, late receiver, and collectives. Other inefficiencies such as OpenMP barriers and MPI collectives are negligible.

Host-device communication is mostly executed concurrently to device computation. Exclusive communication is only between 2.4% and 5.2% of the total runtime for strong scaling and between 2.2% and 4.7% for weak scaling. Relative to the total communication time, the share of copy operations that overlap with compute operations decreases with decreasing workload per GPU. From 1 to 240 processes, it decreases for strong scaling from about 96% to 62% and for weak scaling from 92% to 86%.

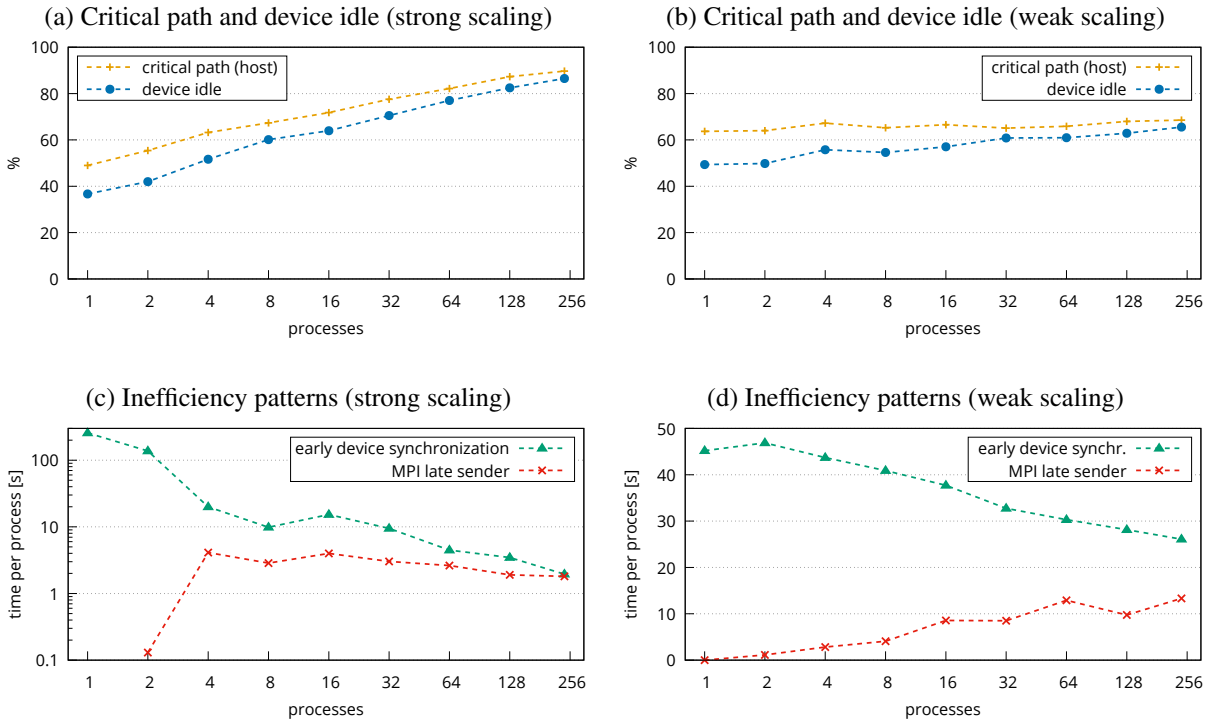


Figure 6.9: HPL-CUDA – Execution inefficiencies and critical path

HPL: Analysis Scalability

To test the analysis scalability over a larger number of processes than with HPL-CUDA, benchmarks of HPL version 2.2 [PWD+16] have been executed on the Haswell island of TU Dresden's Taurus cluster. The software and hardware environment are as specified in Table 6.1, except that the nodes do not contain GPUs. The blocking factor for HPL has been set to 256, the problem size for strong scaling to $128 * 256 = 32768$, and the base problem size for weak scaling to $96 * 256 = 24576$. Except for the problem size and the number of process rows and columns, no parameters of the input file *HPL.dat* (see appendix Listing 4) are changed between the experiments.

Figure 6.10 shows the scalability of the CASITA analysis from 2 to 12288 processes (on 512 nodes) by the example of HPL. For the measured values with 2^n processes, 16 processes were executed per node (from 16 processes onwards). In the measurements with multiples of 24 processes, 24 processes per node were executed and thus all cores were used. The measurement results confirm the expectations and also the findings from the experiments with HPL-CUDA. Although there is no direct correlation between analysis time and program runtime (see Figure 6.10a), the generated trace events build a link between the two. The analysis time closely correlates with the amount of analysis-relevant trace events, which are predominantly from MPI point-to-point (P2P) and MPI wait operations.

Figure 6.10b shows the benchmark, waiting, and analysis time for weak scaling of HPL. The strong increase in waiting time per process explains the strong increase in the HPL runtime from 6144 to 8192 processes (from 256 to 512 nodes). The analysis time increases significantly as of 3072 processes, which is, however, due to the proportionally increasing number of MPI events.

As of 768 processes with strong scaling, the analysis time is longer than the benchmark time. The latter settles at about four seconds, while the analysis time continues to increase slightly up to the maximum measured value of 8.7s. This is basically in line with the expectations, as the communication dominates the benchmark time from a certain scaling and is reenacted twice during the analysis. The critical-path detection uses P2P communication along the path, which took at most 1.6 s during the analysis of the strong scaling runs.

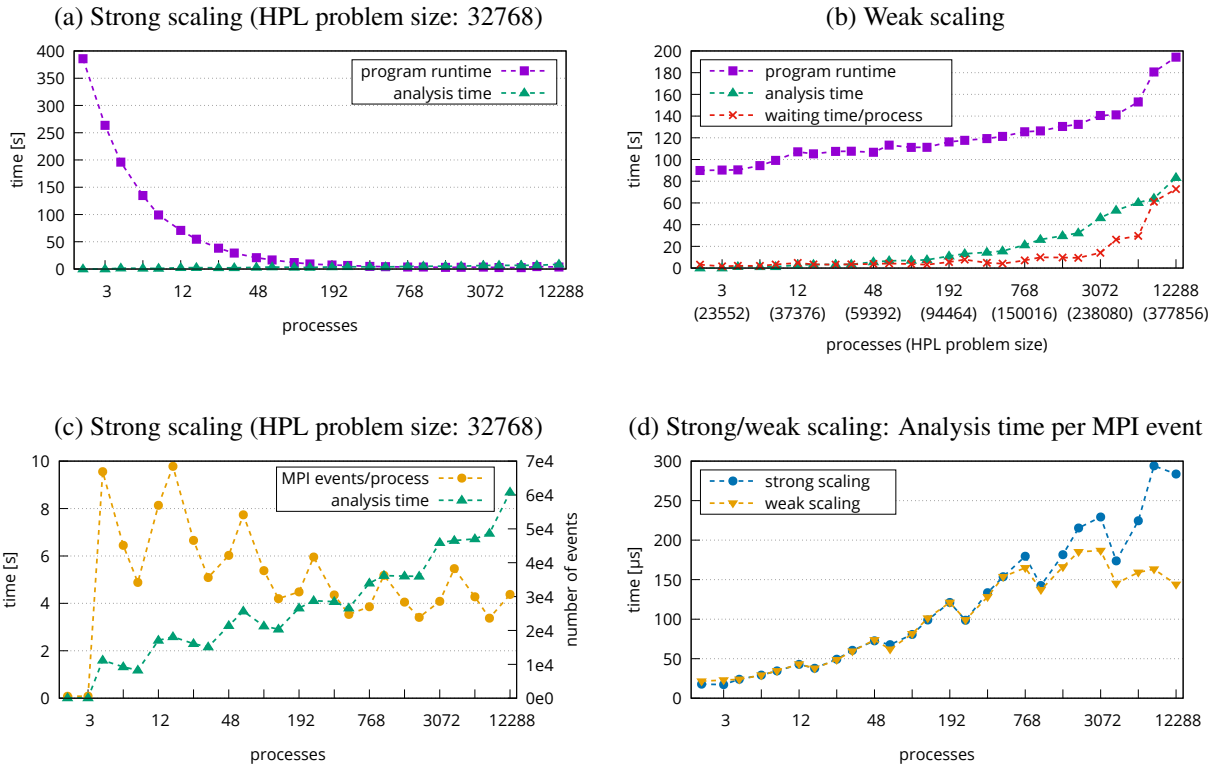


Figure 6.10: HPL – Relation of analysis time with benchmark time (upper graphs) and number of MPI events (lower graphs)

As in the HPL-CUDA experiments, the MPI events per process vary strongly with the number of processes (see Figure 6.10c). For strong scaling from 768 processes upwards, the number of events remains constantly between about 23,000 and 39,000, with a slight tendency to grow with increasing number of processes. Although the analysis time correlates with the number of MPI events, no saturation is visible until 12288 processes. With higher scaling, the analysis time no longer decreases to the same extent as the events per process. Starting with 2048 processes, it increases with the number of processes. Weak scaling generates similar results. However, the number of MPI events also grows with the size of the problem, which in turn increases the analysis time.

Figure 6.10d shows the average analysis time per MPI event (analysis time multiplied by the number of processes and divided by the number of MPI events). As with HPL-CUDA, trace reading has been stripped from the analysis time. Up to 1024 processes, the values for strong and weak scaling are close together. For weak scaling over 1024 processes, the values begin to saturate, while they continue to increase with strong scaling. There is a significant difference in the values around 3072 and from 6144 processes upwards, which indicates a divergent mix of MPI operations or a different communication pattern for strong and weak scaling. In fact, the share of blocking MPI operations is slightly higher for these process counts with strong scaling.

HPL: Inefficiencies

In the HPL experiments, the dominating inefficiency pattern is the MPI late sender with a significant increase in the weak scaling experiments on 512 nodes (8192 and 12288 processes). HPL does not use collectives except for `MPI_Init` and `MPI_Finalize`. The influence of the late receiver pattern is negligible. For weak scaling, the average waiting time per process increases with the number of processes. For strong scaling, it settles at about 1 s per process.

Summary of the Analysis Scalability

The results are generally in line with expectations, although the analysis scales slightly worse than the HPL benchmark itself. The latter is due to the forward and backward replay of MPI communication and the additional critical path detection. The analysis time correlates with the number of events. However, there are several additional influencing factors, such as the communication pattern and peculiarities of the MPI implementation, which can result in longer analysis times.

6.5.3 LSMS – Analyzing a Complex Scientific Code

This section demonstrates the analysis of a complex scientific application code with CASITA. "The Locally Self-consistent Multiple Scattering (LSMS) code solves the first principles Density Functional theory Kohn-Sham equation for a wide range of materials with a special focus on metals, alloys and metallic nanostructures. It has traditionally exhibited near perfect scalability on massively parallel high performance computer architectures."[ELL+17] The LSMS code supports CUDA to efficiently run on GPU-accelerated systems. It implements process-level parallelization via MPI, where each process can optionally use a CUDA-capable GPU for acceleration. Multiple OpenMP threads can also be used for parallelization. In contrast to MPI parallelization, a single GPU is shared within the thread group, which can increase the GPU utilization. Although LSMS supports multithreading and offloading, it does not perform load balancing between host and device.

In the following, different approaches to accelerate the code execution are examined and thus the plausibility of the analysis results is demonstrated. The tuning parameters are the number of host threads, the compiler, code changes, and the migration to other hardware. Table 6.2 lists the used setups, while Table 6.3 shows some key metrics that have been determined by CASITA for the baseline in the first row and each tuning step in the following rows.

Score-P has been used to generate an OTF2 trace for each execution run. To keep the tracing overhead low, the compiler instrumentation was switched off. Instead, the execution steps of the main program loop and the matrix generation routine were manually instrumented. Thus, only a small call stack with very little measuring overhead is recorded and the top-level runtime characteristics of the program are retained. The execution environment was the K80 GPU island of TU Dresden's Taurus cluster (see Table 6.1) and an IBM Power system AC922 with six V100 GPUs und two POWER9 CPUs per node.

Initial Setup

For the first run, the code was compiled with GCC 7.3.0 and executed on a single node with four MPI processes, each using one GPU. The first row in Table 6.3 shows the CASITA metrics for the single-threaded execution of a bulk iron simulation with 128 atoms.

The initial scenario is the execution with only one thread per process, where the critical path and thus the runtime is dominated by host regions with almost 71%. Consequently, the acceleration of the host regions has priority. About 71% device compute idle supports this proposition and indicates that more load should be put on the device. It is the most severe inefficiency, wasting device-compute resources of about $3536\,\mathrm{s}$ in total and $884\,\mathrm{s}$ per device. The second most severe inefficiency is the early blocking device synchronization with about $1456\,\mathrm{s}$ of wasted time on the host in total and $346\,\mathrm{s}$ per device.

The most promising optimization candidate is the manually instrumented host region *buildKKMatrix* with almost 50% of the critical-path time and about $635\,\mathrm{s}$ blame on the critical path. It received the blame almost completely due to not keeping the device busy. The second most promising optimization candidate is a ZGEMM device kernel with about 29% of the critical-path time and $358\,\mathrm{s}$ blame on the critical path. It received all of its blame for letting the host wait. The top 10 ranking and the pattern summary are shown in the appendix in Listing 2.

Setup	CPU	GPU	Compiler	build matrix
I	2 x Intel Xeon E5-2680v3	4 x NVIDIA K80	GCC 7.3.0	on host
II	2 x Intel Xeon E5-2680v3	4 x NVIDIA K80	Intel 2018.1.163	on host
III	2 x Intel Xeon E5-2680v3	4 x NVIDIA K80	Intel 2018.1.163	on device
IV	2 x POWER9	4 x NVIDIA V100	GCC 6.4.0	on device

Table 6.2: LSMS build and execution environments

		Host		Device			
Setup - Threads	Runtime [s]	Critical Path [s] ([%])	Blame on CP [s]	Critical Path [s] ([%])	Blame on CP [s]	Idle [%]	Compute overlap per device [s]
I - 1	1253.5	883.9 (70.5)	807.1	358.7 (28.6)	358.1	71.3	0
II - 1	1047.1	678.6 (64.8)	605.2	357.7 (34.2)	356.7	65.4	0
II - 2	703.3	341.4 (48.5)	300.5	339.7 (48.3)	338.8	49.3	28.1
II - 4	563.8	196.1 (34.8)	157.7	349.2 (61.9)	349.0	37.4	47.2
II - 6	533.7	144.6 (27.1)	128.6	371.0 (69.5)	370.9	33.1	50.4
III - 1	557.6	161.7 (29.0)	78.9	384.8 (69.0)	383.6	30.2	0
III - 6	407.0	27.3 (6.7)	24.7	375.6 (92.3)	375.4	4.9	52.3
IV - 6	81.5	46.7 (57.3)	36.6	32.5 (39.9)	31.9s	39.7	8.3
IV - 8	72.0	35.7 (49.5)	28.2	34.8 (48.3)	34.4	32.7	10.3
IV - 16	65.1	26.6 (40.8)	39.7	37.1 (57.0)	37.3	25.5	11.8

Table 6.3: CASITA results of LSMS simulation with 128 atoms for different execution scenarios

From GCC to Intel Compiler

To accelerate exclusively the host code, the LSMS source code was compiled with the Intel compiler (see Table 6.1), which usually generates faster code for Intel CPUs than GCC [ALGE12, Res17]. Without load balancing between host and device, only the times of the host regions should decrease and the device times should remain approximately the same. The second row of Table 6.3 shows CASITA's analysis results for the Intel-compiled, single-threaded code execution.

In comparison to the GCC version, the code execution is about $206\,\mathrm{s}$ faster, which is an improvement of about 20%. For the host regions, a similar improvement can be seen in the critical-path time and for blame on the critical path, whereas these metrics remain almost constant for the device regions. Hence, neither the application behavior nor the inefficiencies have changed fundamentally. Only the shares of the critical path have changed.

Increasing the Number of Host Threads

Increasing the number of host threads has two effects on the execution of the LSMS code. The host code execution is accelerated by utilizing more CPU cores and the device usage is increased by enabling overlap of device tasks. The second part of Table 6.3 (from row two to five) compares a bulk iron simulation of 128 atoms with one, two, four, and six threads.

The improvement in execution balancing can be seen in CASITA's rating metric *blame on the critical path*. For the host regions, its value decreases by increasing the number of threads, since the host regions are executed faster and thus cause less device idle time.

Using two threads instead of one, the host execution can at best speedup by a factor of two, if no superlinear speedup occurs. Consequently, the runtime gain cannot be greater than half of the host's critical-path time, if the critical-path time on the device remains constant. Based on the values from Table 6.3 (rows two and three), the runtime with two threads cannot be less than $1047.1\,\mathrm{s} - 678.6\,\mathrm{s}/2 = 707.8\,\mathrm{s}$. However, the compute overlap reduces the critical-path time on the device by $18\,\mathrm{s}$, which means that in the best case the program runtime can be $689.8\,\mathrm{s}$. With additional consideration of non-parallelizable regions, parallelization overhead, and resource contention, the measured runtime of $703.3\,\mathrm{s}$ is close to the optimum. Similar calculations can also be made for more threads, whereby the efficiency of parallelization decreases as the number of threads increases. The latter is mainly due to the critical path moving from the host to the device, which can execute compute tasks only to a limited extent concurrently.

The critical-path share of the host regions matches roughly to GPU idle, which basically means that the device tasks are mostly on the critical path while being executed. Due to the absence of load balancing between host and device, device tasks are almost immediately synchronized after their triggers and blamed for letting the host wait. Hence, blame on the critical path and critical-path time are almost identical for device tasks.

The overlap of compute tasks per device (last column in Table 6.3) is about $28\,\mathrm{s}$ (8.3% of the device's critical path time) for two threads and doubles with two additional threads. Using more than four threads hardly increases the computation overlap on the K80 GPU. However, with six threads, it reaches 13.6% of the device's critical path time, which is 9.4% of the program runtime.

The increase in the number of host threads also changes the order in CASITA's optimization guidance profile. With two threads or more, the ZGEMM GPU-kernel takes over the top of the rating, despite a shorter exclusive runtime than *buildKKMatrix*. However, all host regions together still contribute about 55% to the critical path.

While two and four threads show an average OpenMP barrier imbalance of less than $0.5\,\mathrm{s}$ per thread, using six threads increases the average barrier waiting time per thread to more than $33\,\mathrm{s}$. The imbalance is due to the fact that the decomposition into subproblems is restricted. Optimal balancing requires the number of atoms divided by the number of overall host threads to be a positive integer value. Figure 6.11 visualizes the OpenMP imbalance with Vampir. Additionally, it shows the blame metric, which CASITA added to the trace, color coded in the lower timeline. Regions on the *OMP thread 1:0* receive blame from the barrier imbalance, as the thread enters the barrier last. Due to device idle, blame is assigned to regions on all host threads, including OpenMP barrier and *buildKKMatrix*. The master thread receives extra blame for a rather small MPI imbalance, which occurs after the shown section. Finally, the ZGEMM kernels on the CUDA streams are blamed due to causing waiting time on the host.

Build Matrix on the Device

The LSMS code provides the compile-time option to perform the matrix generation on the GPU and thus to move load from host to device. Since the equivalent host region *buildKKMatrix* was on the critical path and did not overlap with device tasks, the critical-path share of the host and device idle time are reduced. Another advantage is that some data transfers are no longer necessary. Therefore, the execution on the GPU is reasonable, if the region cannot be further accelerated on the host and the GPU version is faster overall.

Comparing the two Intel-compiled, single-threaded executions, the matrix generation on the GPU results in a total runtime reduction of about $490\,\mathrm{s}$ (47%) and 35% less GPU idle time (compare Table 6.3). The critical-path share on the host decreases from 64.8% to 29%, while it increases on the device from 34.2% to 69%. As a result, the host has to wait a few more seconds for the GPU, which causes more blame on the device. However, the overall blame on the critical path decreases significantly from about $962\,\mathrm{s}$ to $463\,\mathrm{s}$.

Considering the execution with six threads (and matrix generation on the host), the host region *buildKKMatrix* is with about 18% critical-path share second in the ranking (after the ZGEMM kernel). Con-

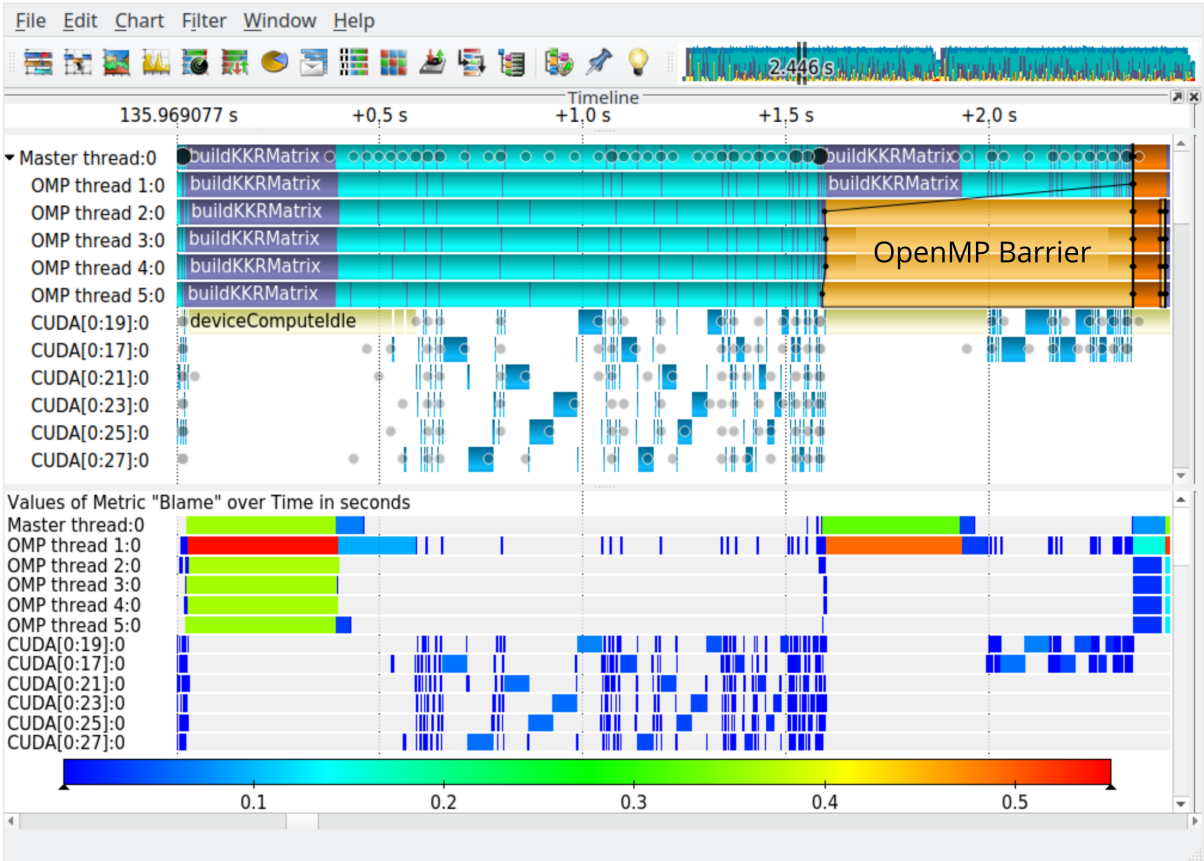


Figure 6.11: OpenMP imbalance in LSMS simulation of 128 atoms with six threads per process. In the *Master Timeline* on the top, the implicit OpenMP barrier is highlighted in orange. While the ZGEMM kernels (blue) are running on the CUDA streams, the host threads are mainly waiting (cyan). The *Performance Radar* on the bottom highlights the blame metric color-coded. Warmer colors indicate higher values of blame for the respective region.

sequently, a smaller runtime gain can be expected from moving the matrix generation to the GPU. In fact, the runtime is about 24% shorter, mainly due to the omission of data transfers. The critical path is over 92% on the device, and thus the optimization focus now clearly on the side of the GPU.

From Haswell+K80 to Power9+V100

Primarily to speed up the GPU execution, the code was ported to TU Dresden's IBM Power system AC922, which pairs two POWER9 CPUs with six NVIDIA Tesla V100 GPUs. Due to comparability with the other runs, only four V100 GPUs are used.

Compared to the fastest execution on the K80 GPUs (Intel-compiled version with six threads and matrix build on the GPU), the runtime has dropped significantly from 407 s to 81.5 s (80% less). Host regions provide with over 57% critical-path share and 36.6 s blame on the critical path again more potential for runtime improvement than device kernels.

Using eight threads instead of six should, in the best case, reduce the critical path on the host by 25% from 46.7 s to 35 s. CASITA determines a critical-path time of 35.7 s. The overlap of the compute tasks per device also increases by about 25% from 8.3 s to 10.3 s. The nearly 10 s shorter overall runtime with eight threads results in a balanced distribution of the critical path and also achieves less overall blame on the critical path (62.6 s).

Changing to 16 host threads should further reduce the execution time on the host and shorten the critical-path time on the device due to even more overlap of compute tasks. A runtime gain of 1.5 s are due to

```
Stream summary: 240 MPI rank(s), 1440 host stream(s), 240 device(s)
Total program runtime        :    135.50s
Total waiting time (host)    : 106005.58s (441.69s / rank, 73.61s (54.33%) / host stream)
                                           384.79s on rank 184 (min) - taurusi2092
                                           587.08s on rank  38 (max) - taurusi2054
Pattern summary:
 MPI wait patterns           :  5615.75s (23.40s / rank; 20026 overall occurrences)
  Late sender                :   377.27s (1.57s / rank; 2805 overall occurrences)
  Late receiver              :     0.01s (0.000047s / rank; 13 overall occurrences)
  Wait in MPI collective     :  5238.47s (21.83s / rank; 17208 overall occurrences)
 OpenMP
  Wait in OpenMP barrier     :   708.96s (0.49s / host stream, 220032 overall occurrences)
 Offloading
  Idle device                :  6812.46s (21.96% -> 28.39s / device)
  Compute idle device        :  6869.02s (22.14% -> 28.62s / device)
  Early blocking wait        : 99667.19s (415.28s / device, 2883564 overall occurrences)
  ... on compute kernels     : 27236.76s (113.49s / device)
  Total communication time   :   379.50s (1.22% of offload time)
  -Exclusive communication   :    56.56s (14.90% of total communication time)
  -Copy-compute overlap      :   322.95s (85.10% of total communication time)
  -Blocking communication    :     0.03s (1440 occurrences), exclusive: 0.03s
  -Consecutive communication :    74.93s (1462950 occ.), same direct. 19.74s (599534 occ.)
  Compute overlap            :  3106.91s (12.952s / device)
  Kernel startup delay       :     4.46ms/kernel (626897 occ.), total delay: 2799.06s
```

Listing 1: CASITA pattern summary for LSMS with 1024 atoms on 240 GPUs

more overlap of device compute tasks, whereas the average waiting time in OpenMP barriers per thread increases by $1.2\,\mathrm{s}$. The latter significantly increases blame on the critical path. The critical-path time decreases by $9.1\,\mathrm{s}$ on the host, while it increases only by $2.3\,\mathrm{s}$ on the device. The gap in between equals almost exactly the runtime gain of $6.9\,\mathrm{s}$.

Problem Size and Number of Processes

LSMS implements a static load balancing, which results in MPI imbalances depending on the size of the problem and the number of processes. Previously, the analysis was limited to the node-level performance for a fixed number of 128 atoms on four processes, which results in negligible MPI imbalances. Valid numbers of atoms are $2 * n^3$ for positive integers n and the number of processes must be greater than or equal to the number of atoms.

Increasing the problem size for a fixed number of processes also increases the MPI imbalances. The dominating MPI imbalance manifests itself in `MPI_Allgather` collectives. For experiments with four processes from 128 to 1024 atoms, the average MPI waiting time per process remains between 1% and 2% of the program runtime.

Moving to 240 processes for a constant number of 1024 atoms increases the average MPI waiting time per process to about 17% ($23.4\,\mathrm{s}/135.5\,\mathrm{s}$) of the program runtime. Thus, the work is not evenly distributed among the individual processes. According to the measurements there is no relation between the MPI imbalance and the integer divisibility of the number of atoms by the number of processes. However, an increase in processes tends to result in higher MPI waiting times.

Listing 1 shows the plain CASITA pattern summary output. The global imbalance can already be seen in the variation of the total waiting time per process group (process and its threads and device streams). On average, each host thread waits for more than half of its execution time, most of the time for a device stream to finish execution ($415.28\,\mathrm{s}/6$ threads$/135.5\,\mathrm{s} = 51\%$), while the GPUs are on average for about 22% idle. As there are no OpenMP parallel regions executed during MPI communication, the host threads are idle or waiting for $17\% + 51\% = 68\%$ of the program runtime. It should be noted that the idle time of OpenMP threads is not accounted in CASITA's *Total waiting time (host)* output. The OpenMP barrier imbalance has with about $0.5\,\mathrm{s}$ on average per thread a negligible impact on the program runtime.

7 Conclusion, Future Work, and Outlook

This dissertation presents a framework for systematic performance analysis of parallel programs. With performance analysis being a powerful tool for optimizing parallel programs, it is particularly useful for hybrid parallelization, where inefficiencies can propagate over the boundaries of a programming model. A prominent inefficiency are load imbalances with wait states or idle times as symptoms. On the basis of patterns, inefficient parallel execution can be identified and quantified, to finally determine the cause of imbalances and their relevance for the overall program execution.

To enable a comprehensive performance analysis of heterogeneous applications, potential inefficiencies in the use of accelerators have been examined in Chapter 4. One of the main contributions of this thesis is the specification of inefficiency patterns that can occur during computation offloading. Together with the inefficiency patterns in MPI and OpenMP, which have already been specified in previous work, the resulting wait and idle states on host and device are the prerequisite for the proposed holistic analysis. To also enable a detailed analysis of programs with OpenACC and OpenMP directives, this work contributed significantly to the specification of respective tool interfaces for performance data acquisition.

A holistic performance analysis across process- and thread-level parallelization as well as computation offloading has been presented in Chapter 5. Based on wait states as inherent part of several inefficiency patterns, a generic application of root-cause and critical-path analysis has been developed. While the root-cause analysis identifies the actual cause of imbalances, the critical-path analysis highlights regions that have an impact on the overall program runtime. In combination, regions on the critical path are weighted according to the imbalances caused. On the other hand, program regions that are not on the critical path are excluded as potential optimization candidates. Since both analyses overlap to a large extent, the critical-path detection has been simplified to tracking of a single path.

The implementation of the analysis framework and its prerequisites has been described in Chapter 6. It covers all layers in the performance analysis process. Besides the integration of the tool interfaces for OpenACC and OpenMP, the data acquisition with Score-P has been extended so that dependencies between host operations and offloaded tasks can be reconstructed. The trace-based inefficiency analysis has been implemented in CASITA, which generates an optimization guidance profile, a pattern summary, and an enhanced output trace. The highest rated program regions and the listed inefficiencies serve as starting points for optimization. Vampir's generic visualization concept allows the waiting times, their cause, and the critical path to be displayed in counter-enabled timelines. The correctness, scalability, and applicability of the analysis was demonstrated with synthetic programs, the HPL benchmark, and a complex scientific application.

In summary, this thesis extends the scope of previous performance analysis and respective software tools. This includes the specification of potential inefficiencies in computation offloading and the proposed generic analysis, which enables the accurate identification of the root causes of inefficiencies. In combination with the critical path, more expressive metrics can be provided than with separate analyses or the subsequent combination of their results. Eventually, this allows a more accurate guidance towards program regions that are relevant for program optimization and thus a systematic analysis of heterogeneous programs with hybrid parallelization.

The results of this work have already been incorporated into various components of performance analysis. The tool interfaces for the directive-based programming APIs OpenACC and OpenMP are part of the corresponding specifications. Respective implementations are available with PGI's OpenACC runtime and LLVM's OpenMP runtime. The performance measurement infrastructure Score-P was extended by the described interfaces for OpenACC and OpenMP as well as the ability to capture host-device dependencies. Based on the results of this work, NVIDIA implemented the OpenACC tool interface and the critical-path analysis for offloading with the CUDA API in their profiling tools.

As future work, the presented performance analysis framework can be refined and extended in many ways. An appropriate extension would be the integration of the system's hardware topology in the generic analysis. Moreover, the analysis does not yet include the general concept of tasking, which could cover computation offloading as a specialized tasking model. This work has abstracted inefficiency patterns for offloading APIs, although many inefficiency patterns specified for traditional programming APIs such as MPI and OpenMP are also applicable in abstract form to the underlying programming models. Some patterns can be further abstracted, since they are existent in several programming models, e.g. collective waiting in a barrier.

The proposed analysis approach also enables more metrics to be determined, such as the parallel efficiency and the pure communication time in the individual programming models. The latter is determined in this work only for computation offloading. A further step could be the enhancement of the analysis capability, e.g. by a near-critical-path analysis, to estimate the runtime gain through the optimization of a program region. In general, the proposed graph-based approach also allows a performance prediction, if corresponding candidates are known beforehand, e.g. from a profile.

Considering the implementation of the trace analysis, CASITA can be tuned primarily in terms of parallelization and more efficient data structures to reduce the analysis time and the memory consumption. Moreover, the input trace could be supplemented with the additional information, instead of being completely rewritten. Eventually, CASITA is a useful tool to further enrich the Score-P software stack.

In the future, it is to be expected that the already complex parallelism of today's computing systems and software will increase. There is a good chance that the heterogeneity grows in order to be able to solve various special problems more efficiently. Hence, the relevance and usefulness of the presented performance analysis framework are likely to increase.

Bibliography

[AAB+17] Emmanuel Agullo, Olivier Aumage, Bérenger Bramas, Olivier Coulaud, and Samuel Pitoiset. Bridging the Gap Between OpenMP and Task-Based Runtime Systems for the Fast Multipole Method. *IEEE Transactions on Parallel and Distributed Systems*, 28(10):2794–2807, 2017. DOI:10.1109/TPDS.2017.2697857.

[ABB+13] Thomas Alrutz, Jan Backhaus, Thomas Brandes, Vanessa End, Thomas Gerhold, Alfred Geiger, Daniel Grünewald, Vincent Heuveline, Jens Jägersküpper, Andreas Knüpfer, Olaf Krzikalla, Edmund Kügeler, Carsten Lojewski, Guy Lonsdale, Ralph Müller-Pfefferkorn, Wolfgang Nagel, Lena Oden, Franz-Josef Pfreundt, Mirko Rahn, Michael Sattler, Mareike Schmidtobreick, Annika Schiller, Christian Simmendinger, Thomas Soddemann, Godehard Sutmann, Henning Weber, and Jan-Philipp Weiss. *GASPI – A Partitioned Global Address Space Programming Interface*, pages 135–136. Springer Berlin Heidelberg, 2013. DOI:10.1007/978-3-642-35893-7_18.

[ABF+10] Laksono Adhianto, Sinchan Banerjee, Mike Fagan, Mark Krentel, Gabriel Marin, John Mellor-Crummey, and Nathan R. Tallent. HPCTOOLKIT: Tools for Performance Analysis of Optimized Parallel Programs. *Concurrency and Computation: Practice and Experience*, 22(6):685–701, 2010. DOI:10.1002/cpe.1553.

[AHLT09] Eric Anderson, Christopher Hoover, Xiaozhou Li, and Joseph Tucek. Efficient Tracing and Performance Analysis for Large Distributed Systems. In *International Symposium on Modeling, Analysis Simulation of Computer and Telecommunication Systems*, pages 1–10. IEEE, 2009. DOI:10.1109/MASCOT.2009.5366158.

[ALGE12] Sergio Aldea, Diego R. Llanos, and Arturo González-Escribano. Using SPEC CPU2006 to evaluate the sequential and parallel code generated by commercial and open-source compilers. *The Journal of Supercomputing*, 59(1):486–498, 2012. DOI:10.1007/s11227-010-0449-4.

[Amd67] 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.

[ARH94] Cedell A. Alexander, Donna S. Reese, and James C. Harden. Near-Critical Path Analysis of Program Activity Graphs. In *Proceedings of the Second International Workshop on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems*, MASCOTS, pages 308–317, 1994. DOI:10.1109/MASCOT.1994.284406.

[ATNW11] Cédric Augonnet, Samuel Thibault, Raymond Namyst, and Pierre-André Wacrenier. StarPU: A Unified Platform for Task Scheduling on Heterogeneous Multicore Architectures. *Concurrency and Computation: Practice & Experience*, 23(2):187–198, 2011. DOI:10.1002/cpe.1631.

[Bau14] Michael Edward Bauer. *Legion: Programming Distributed Heterogeneous Architectures with Logical Regions*. PhD thesis, STANFORD UNIVERSITY, 2014. online at legion.stanford.edu, accessed 10 Feb 2019.

[BB14] Enes Bajrovic and Siegfried Benkner. Automatic Performance Tuning of Pipeline Patterns for Heterogeneous Parallel Architectures. In *Proceedings of International Conference on Parallel and Distributed Processing, Techniques and Applications (PDPTA)*, 2014. online at worldcomp-proceedings.com, accessed 10 Feb 2019.

[BBD+13] George Bosilca, Aurelien Bouteiller, Anthony Danalis, Mathieu Faverge, Thomas Herault, and Jack J. Dongarra. PaRSEC: Exploiting Heterogeneity to Enhance Scalability. *Computing in Science Engineering*, 15(6):36–45, 2013. DOI:10.1109/MCSE.2013.98.

[BDG+16] G. Ballard, J. Demmel, A. Gearhart, B. Lipshitz, Y. Oltchik, O. Schwartz, and S. Toledo. Network Topologies and Inevitable Contention. In *First International Workshop on Communication Optimizations in HPC (COMHPC)*, pages 39–52. IEEE, 2016. DOI:10.1109/COMHPC.2016.010.

[BGWA10] David Böhme, Markus Geimer, Felix Wolf, and Lukas Arnold. Identifying the Root Causes of Wait States in Large-Scale Parallel Applications. In *39th International Conference on Parallel Processing*, ICPP, pages 90–100. IEEE, 2010. DOI:10.1109/ICPP.2010.18.

[BH00] Bryan Buck and Jeffrey K. Hollingsworth. An API for Runtime Code Patching. *International Journal of High Performance Computing Applications*, 14(4):317–329, November 2000. DOI:10.1177/109434200001400404.

[BMS03] Robert Bell, Allen D. Malony, and Sameer Shende. ParaProf: A Portable, Extensible, and Scalable Tool for Parallel Performance Profile Analysis. In Harald Kosch, László Böszörményi, and Hermann Hellwagner, editors, *Euro-Par 2003 Parallel Processing*, pages 17–26. Springer Berlin Heidelberg, 2003. DOI:10.1007/978-3-540-45209-6_7.

[Böh13] David Böhme. *Characterizing Load and Communication Imbalance in Parallel Applications*. PhD thesis, RWTH Aachen, 2013. online at publications.rwth-aachen.de, accessed 10 Feb 2019.

[BP13] Natalie Bates and Michael Patterson. Achieving the 20MW Target: Mobilizing the HPC Community to Accelerate Energy Efficient Computing. In *Transition of HPC Towards Exascale Computing*, volume 24 of *Advances in Parallel Computing*, pages 37–45. IOS press, 2013. DOI:10.3233/978-1-61499-324-7-37.

[BPG10] Shajulin Benedict, Ventsislav Petkov, and Michael Gerndt. PERISCOPE: An Online-Based Distributed Performance Analysis Tool. In *Tools for High Performance Computing 2009*, pages 1–16. Springer Berlin Heidelberg, 2010. DOI:10.1007/978-3-642-11261-4_1.

[BWdS+12] David Böhme, Felix Wolf, Bronis R. de Supinski, Martin Schulz, and Markus Geimer. Scalable Critical-Path Based Performance Analysis. In *26th International Parallel Distributed Processing Symposium*, IPDPS, pages 1330–1340. IEEE, 2012. DOI:10.1109/IPDPS.2012.120.

[CBL10] Marc Casas, Rosa Badia, and Jesús Labarta. Automatic Phase Detection and Structure Extraction of MPI Applications. *International Journal of High Performance Computing Applications*, 2010. DOI:10.1177/1094342009360039.

[CDT+15] Tim Cramer, Robert Dietrich, Christian Terboven, Matthias S. Müller, and Wolfgang E. Nagel. Performance Analysis for Target Devices with the OpenMP Tools Interface. In *29th International Parallel and Distributed Processing Symposium Workshop*, IPDPSW. IEEE, 2015. DOI:10.1109/IPDPSW.2015.27.

[cFC07] Wu chun Feng and Kirk Cameron. The Green500 List: Encouraging Sustainable Supercomputing. *Computer*, 40(12):50–55, 2007. DOI:10.1109/MC.2007.445.

[CH01] Edmond Chow and David Hysom. Assessing Performance of Hybrid MPI/OpenMP Programs on SMP Clusters. Technical report, Lawrence Livermore National Laboratory, 2001.

[CMFMC13] Milind Chabbi, Karthik Murthy, Michael Fagan, and John Mellor-Crummey. Effective Sampling-driven Performance Tools for GPU-accelerated Supercomputers. In *International Conference on High Performance Computing, Networking, Storage and Analysis*, SC, pages 43:1–43:12. ACM, 2013. DOI:10.1145/2503210.2503299.

[CUG15] Isaías A. Comprés Ureña and Michael Gerndt. Tuning Plugin Development for the Periscope Tuning Framework. In *Tools for High Performance Computing 2014*, pages 81–121. Springer International Publishing, 2015. DOI:10.1007/978-3-319-16012-2_5.

[CVKG10] Long Chen, Oreste Villa, Sriram Krishnamoorthy, and Guang R. Gao. Dynamic Load Balancing on Single- and Multi-GPU Systems. In *International Symposium on Parallel Distributed Processing (IPDPS)*, pages 1–12. IEEE, 2010. DOI:10.1109/IPDPS.2010.5470413.

[DBB07] Romain Dolbeau, Stéphane Bihan, and François Bodin. HMPP™: A Hybrid Multi-core Parallel Programming Environment. In *Proceedings of the 1st Workshop on General Purpose Processing on Graphics Processing Units*, 2007.

[DBH+12] James Dinan, Pavan Balaji, Jeff R. Hammond, Sriram Krishnamoorthy, and Vinod Tipparaju. Supporting the Global Arrays PGAS Model Using MPI One-Sided Communication. In *Proceedings of 26th International Parallel and Distributed Processing Symposium*, IPDPS, pages 739–750. IEEE, 2012. DOI:10.1109/IPDPS.2012.72.

[DHJ07] Luiz DeRose, Bill Homer, and Dean Johnson. Detecting Application Load Imbalance on High End Massively Parallel Systems. In *Euro-Par 2007 Parallel Processing*, volume 4641 of *Lecture Notes in Computer Science*, pages 150–159. Springer Berlin Heidelberg, 2007. DOI:10.1007/978-3-540-74466-5_17.

[DHJ+08] Luiz DeRose, Bill Homer, Dean Johnson, Steve Kaufmann, and Heidi Poxon. Cray Performance Analysis Tools. In *Tools for High Performance Computing*, pages 191–199. Springer Berlin Heidelberg, 2008. DOI:10.1007/978-3-540-68564-7_12.

[Dij59] Edsger W. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1(1):269–271, 1959. DOI:10.1007/BF01386390.

[DIJ10] Robert Dietrich, Thomas Ilsche, and Guido Juckeland. Non-Intrusive Performance Analysis of Parallel Hardware Accelerated Applications on Hybrid Architectures. In *39th International Conference on Parallel Processing Workshops*, ICPPW, pages 135–143. IEEE, 2010. DOI:10.1109/ICPPW.2010.30.

[DJW15] Robert Dietrich, Guido Juckeland, and Michael Wolfe. OpenACC Programs Examined: A Performance Analysis Approach. In *44th International Conference on Parallel Processing*, ICPP. IEEE, 2015. DOI:10.1109/ICPP.2015.40.

[DK04] N. Drosinos and N. Koziris. Performance comparison of pure MPI vs hybrid MPI-OpenMP parallelization models on SMP clusters. In *Proceedings of 18th International Parallel and Distributed Processing Symposium*, 2004. DOI:10.1109/IPDPS.2004.1302919.

[DKMN08] Jens Doleschal, Andreas Knüpfer, Matthias S. Müller, and Wolfgang E. Nagel. Internal timer synchronization for parallel event tracing. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, volume 5205 of *Lecture Notes in Computer Science*, pages 202–209. Springer Berlin Heidelberg, 2008. DOI:10.1007/978-3-540-87475-1_29.

[DLP03] Jack J. Dongarra, Piotr Luszczek, and Antoine Petitet. The LINPACK Benchmark: past, present and future. *Concurrency and Computation: Practice and Experience*, 15(9):803–820, 2003. DOI:10.1002/cpe.728.

[Dol16] Romain Dolbeau. Theoretical Peak FLOPS per instruction set on less conventional hardware, 2016. DOI:10.13140/RG.2.2.13018.75209. Working Paper.

[dOSdK13] Benhur de Oliveira Stein and Jacques Chassin de Kergommeaux. Pajé trace file format. Technical report, Departamento de Eletrônica e Computação – Universida de Federal de Santa Maria, Laboratoire Logiciel Systèmes et Réseaux, 2013. online at paje.sourceforge.net, accessed 10 Feb 2019.

[DS17] Robert Dietrich and Sameer Shende. *OpenACC for Programmers: Concepts and Strategies*, chapter 3 – Programming Tools for OpenACC. Addison-Wesley Professional, 1st edition, 2017.

[DSGS14] Robert Dietrich, Felix Schmitt, Alexander Grund, and Dirk Schmidl. Performance Measurement for the OpenMP 4.0 Offloading Model. In *Euro-Par 2014: Parallel Processing Workshops*, volume 8806 of *Lecture Notes in Computer Science*, pages 291–301. Springer International Publishing, 2014. DOI:10.1007/978-3-319-14313-2_25.

[DSGS16] Robert Dietrich, Felix Schmitt, Alexander Grund, and Jonas Stolle. Critical-Blame Analysis for OpenMP 4.0 Offloading on Intel Xeon Phi. *Journal of Systems and Software*, 125:381 – 388, 2016. DOI:10.1016/j.jss.2015.12.050.

[DSS19] Robert Dietrich, Felix Schmitt, and Jonas Stolle. Critical path analysis tool for heterogeneous applications (CASITA). In *Open Access Repository and Archive*. Technische Universität Dresden, 2019. DOI:10.25532/OPARA-28.

[DSWB12] Robert Dietrich, Felix Schmitt, René Widera, and Michael Bussmann. Phase-Based Profiling in GPGPU Kernels. In *41st International Conference on Parallel Processing Workshops*, ICPPW, pages 414–423. IEEE, 2012. DOI:10.1109/ICPPW.2012.59.

[DT15] Robert Dietrich and Ronny Tschüter. A Generic Infrastructure for OpenCL Performance Analysis. In *8th International Conference on Intelligent Data Acquisition and Advanced Computing Systems*, volume 1 of *Technology and Applications*, pages 334–341. IEEE, 2015. DOI:10.1109/IDAACS.2015.7340754.

[DTC+16] Robert Dietrich, Ronny Tschüter, Tim Cramer, Guido Juckeland, and Andreas Knüpfer. Evaluation of Tool Interface Standards for Performance Analysis of OpenACC and OpenMP Programs. In *Tools for High Performance Computing 2015*, pages 67–83. Springer International Publishing, 2016. DOI:10.1007/978-3-319-39589-0_6.

[DTJK17] Robert Dietrich, Ronny Tschüter, Guido Juckeland, and Andreas Knüpfer. *Analyzing Offloading Inefficiencies in Scalable Heterogeneous Applications*, pages 457–476. Springer International Publishing, 2017. DOI:10.1007/978-3-319-67630-2_34.

[DWK+17] Matthias Diener, Sam White, Laxmikant V. Kale, Michael Campbell, Daniel J. Bodony, and Jonathan B. Freund. Improving the Memory Access Locality of Hybrid MPI Applications. In *Proceedings of the 24th European MPI Users' Group Meeting*, EuroMPI, pages 11:1–11:10. ACM, 2017. DOI:10.1145/3127024.3127038.

[DWMDF+15] Mattias De Wael, Stefan Marr, Bruno De Fraine, Tom Van Cutsem, and Wolfgang De Meuter. Partitioned Global Address Space Languages. *ACM Computing Surveys*, 47(4):62:1–62:27, 2015. DOI:10.1145/2716320.

[DWW+13] Robert Dietrich, Frank Winkler, Thomas William, Jonas Stolle, Robert Henschel, and Donald K. Berry. *A Case Study: Holistic Performance Analysis on Heterogeneous Architectures using the Vampir Toolchain*, pages 793–802. Advances in Parallel Computing. IOS Press, 2013. DOI:10.3233/978-1-61499-381-0-793.

[EBW11] Dominic Eschweiler, Daniel Becker, and Felix Wolf. Patterns of Inefficient Performance Behavior in GPU Applications. In *19th International Euromicro Conference on Parallel, Distributed and Network-Based Processing*, PDP, pages 262–266. IEEE, 2011. DOI:10.1109/PDP.2011.84.

[ELL+17] Markus Eisenbach, Jeff Larkin, Justin Lutjens, Steven Rennich, and James H. Rogers. GPU acceleration of the Locally Selfconsistent Multiple Scattering code for first principles calculation of the ground state and statistical physics of materials. *Computer Physics Communications*, 211:2–7, 2017. DOI:10.1016/j.cpc.2016.07.013.

[EMCS+13] Alexandre E. Eichenberger, John Mellor-Crummey, Martin Schulz, Michael Wong, Nawal Copty, Robert Dietrich, Xu Liu, Eugene Loh, and Daniel Lorenz. OMPT: An OpenMP Tools Application Programming Interface for Performance Analysis. In *OpenMP in the Era of Low Power Devices and Accelerators*, volume 8122 of *Lecture Notes in Computer Science*, pages 171–185. Springer Berlin Heidelberg, 2013. DOI:10.1007/978-3-642-40698-0_13.

[EMCS+14] Alexandre Eichenberger, John Mellor-Crummey, Martin Schulz, Nawal Copty, Jim Cownie, Robert Dietrich, Xu Liu, Eugene Loh, and Daniel Lorenz. OpenMP Technical Report 2 on the OMPT Interface. Technical report, OpenMP Architecture Review Board, 2014. online at openmp.org, accessed 9 Feb 2019.

[ENC04] "wait state". A Dictionary of Computing. *Encyclopedia.com*, 2004. online at encyclopedia.com, accessed 4 Mar 2019.

[ETS14] H. Carter Edwards, Christian R. Trott, and Daniel Sunderland. Kokkos: Enabling manycore performance portability through polymorphic memory access patterns. *Journal of Parallel and Distributed Computing*, 74(12):3202–3216, 2014. DOI:10.1016/j.jpdc.2014.07.003.

[EWG+12] Dominic Eschweiler, Michael Wagner, Markus Geimer, Andreas Knüpfer, Wolfgang E. Nagel, and Felix Wolf. Open Trace Format 2 - The next generation of scalable trace formats and support libraries. In *Applications, Tools and Techniques on the Road to Exascale Computing*, volume 22 of *Advances in Parallel Computing*, pages 481–490. IOS Press, 2012. DOI:10.3233/978-1-61499-041-3-481.

[EZL89] Derek L. Eager, John Zahorjan, and Edward D. Lazowska. Speedup versus efficiency in parallel systems. *IEEE Transactions on Computers*, 38(3):408–423, 1989. DOI:10.1109/12.21127.

[Fat09] Massimiliano Fatica. Accelerating Linpack with CUDA on Heterogenous Clusters. In *Proceedings of 2nd Workshop on General Purpose Processing on Graphics Processing Units*, GPGPU-2, pages 46–51. ACM, 2009. DOI:10.1145/1513895.1513901.

[FBM+14] Alejandro Fernández, Vicenç Beltran, Xavier Martorell, Rosa M. Badia, Eduard Ayguadé, and Jesús Labarta. *Task-Based Programming with OmpSs and Its Application*, pages 601–612. Springer International Publishing, 2014. DOI:10.1007/978-3-319-14313-2_51.

[Fou18] HSA Foundation. HSA Platform System Architecture Specification 1.2, 2018. online at hsafoundation.com, accessed 10 Feb 2019.

[FS18] Wu Feng and Tom Scogland. Green500 List, November 2018. online at https://www.top500.org/green500, accessed 10 Feb 2019.

[GGL09] Juan González, Judit Giménez, and Jesús Labarta. Automatic Detection of Parallel Applications Computation Phases. In *23rd International Parallel and Distributed Processing Symposium*, IPDPS, pages 1–11. IEEE, 2009. DOI:10.1109/IPDPS.2009.5161027.

[GKM82] Susan L. Graham, Peter B. Kessler, and Marshall K. Mckusick. Gprof: A Call Graph Execution Profiler. *SIGPLAN Notices*, 17(6):120–126, 1982. DOI:10.1145/872726.806987.

[GS13] William Gropp and Marc Snir. Programming for Exascale Computers. *Computing in Science Engineering*, 15(6):27–35, Nov 2013. DOI:10.1109/MCSE.2013.96.

[Gus88] John L. Gustafson. Reevaluating Amdahl's Law. *Communications of the ACM*, 31(5):532–533, 1988. DOI:10.1145/42411.42415.

[GWW+10] Markus Geimer, Felix Wolf, Brian J. N. Wylie, Daniel Becker Erika Abraham, and Bernd Mohr. The Scalasca Performance Toolset Architecture. *Concurrency and Computation: Practice and Experience*, 22(6):702–719, 2010. DOI:10.1002/cpe.1556.

[GWWM09] Markus Geimer, Felix Wolf, Brian J.N. Wylie, and Bernd Mohr. A Scalable Tool Architecture for Diagnosing Wait States in Massively Parallel Applications. *Parallel Computing*, 35(7):375–388, 2009. DOI:10.1016/j.parco.2009.02.003.

[Hal92] Robert J. Hall. Call Path Profiling. In *Proceedings of the 14th International Conference on Software Engineering*, ICSE, pages 296–306. ACM, 1992. DOI:10.1145/143062.143147.

[Har12] Mark Harris. *How to Optimize Data Transfers in CUDA C/C++*. NVIDIA, 2012. online at devblogs.nvidia.com, accessed 24 Feb 2019.

[Her09] Marc-André Hermanns. Trace-based performance simulation of large-scale applications. Master's thesis, University of Hagen, 2009. online at juser.fz-juelich.de, accessed 11 Feb 2019.

[HGC14] Jeff R. Hammond, Sayan Ghosh, and Barbara M. Chapman. Implementing OpenSHMEM Using MPI-3 One-Sided Communication. In *OpenSHMEM and Related Technologies. Experiences, Implementations, and Tools*, volume 8356 of *Lecture Notes in Computer Science*, pages 44–58. Springer International Publishing, 2014. DOI:10.1007/978-3-319-05215-1_4.

[HHdSSB11] Charles J. Horowitz, Joseph Hughto, Andre da Silva Schneider, and Donald K. Berry. Neutron star crust and molecular dynamics simulation. *Neutron Star Crust*, 2011, arXiv:1109.5095.

[Hil90] Mark D. Hill. What is scalability? *SIGARCH Computer Architecture News*, 18(4):18–21, December 1990. DOI:10.1145/121973.121975.

[HK14] Richard D. Hornung and Jeffrey A. Keasler. The RAJA Portability Layer: Overview and Status, LLNL-TR-661403. Technical report, Lawrence Livermore National Laboratory, 2014. DOI:10.2172/1169830.

[HMBM05] Kevin A. Huck, Allen D. Malony, Robert Bell, and Alan Morris. Design and Implementation of a Parallel Performance Data Management Framework. In *International Conference on Parallel Processing*, ICPP, pages 473–482. IEEE, June 2005. DOI:10.1109/ICPP.2005.29.

[HMBW13] Marc-André Hermanns, Manfred Miklosch, David Böhme, and Felix Wolf. Understanding the Formation of Wait States in Applications with One-sided Communication. In *Proceedings of the 20th European MPI Users' Group Meeting*, EuroMPI, pages 73–78. ACM, 2013. DOI:10.1145/2488551.2488569.

[Hol96] Jeffrey K. Hollingsworth. An Online Computation of Critical Path Profiling. In *Proceedings of the SIGMETRICS Symposium on Parallel and Distributed Tools*, SPDT, pages 11–20. ACM, 1996. DOI:10.1145/238020.238024.

[HV12] Martijn Hendriks and Frits W. Vaandrager. Reconstructing Critical Paths from Execution Traces. In *15th International Conference on Computational Science and Engineering*, CSE, pages 524–531. IEEE, 2012. DOI:10.1109/ICCSE.2012.78.

[Int16a] Intel Corporation. *Intel® Xeon Phi™ Processor x200, Offload Over Fabric, User's Guide, Revision 1.0*, 2016. online at intel.com, accessed 11 Feb 2019.

[Int16b] *Intel® Xeon Phi™ Processor 7290*, 2016. Specifications, online at ark.intel.com, accessed 9 Feb 2019.

[Int16c] *Intel® Xeon® Processor E5-2699 v4*, 2016. Specifications, online at ark.intel.com, accessed 9 Feb 2019.

[Jai91] Raj Jain. *The Art of Computer Systems Performance Analysis*, volume 182. John Wiley & Sons New York, 1991.

[JD17] Guido Juckeland and Robert Dietrich. *Parallel Programming with OpenACC*, chapter 3 – Profiling performance of hybrid applications with Score-P and Vampir, pages 55–68. Morgan Kaufmann, 2017. DOI:10.1016/B978-0-12-410397-9.00003-2.

[JR13] James Jeffers and James Reinders. *Intel Xeon Phi Coprocessor High Performance Programming*. Morgan Kaufmann Publishers Inc., 1st edition, 2013. DOI:10.1016/C2011-0-06997-1.

[Juc12] Guido Juckeland. *Trace-based Performance Analysis for Hardware Accelerators*. PhD thesis, Technische Universität Dresden, 2012. online at qucosa.de, accessed 9 Feb 2019.

[KBB+06] Andreas Knüpfer, Ronny Brendel, Holger Brunst, Hartmut Mix, and Wolfgang E. Nagel. Introducing the Open Trace Format (OTF). In *Computational Science – ICCS 2006*, volume 3992, pages 526–533. Springer Berlin Heidelberg, 2006. DOI:10.1007/11758525_71.

[KBD+08] Andreas Knüpfer, Holger Brunst, Jens Doleschal, Matthias Jurenz, Matthias Lieber, Holger Mickler, Matthias S. Müller, and Wolfgang E. Nagel. The Vampir Performance Analysis Tool-Set. In *Tools for High Performance Computing*, pages 139–155. Springer Berlin Heidelberg, 2008. DOI:10.1007/978-3-540-68564-7_9.

[KBH+08] Rob Knauerhase, Paul Brett, Barbara Hohlt, Tong Li, and Scott Hahn. Using OS Observations to Improve Performance in Multicore Systems. *IEEE Micro*, 28(3):54–66, 2008. DOI:10.1109/MM.2008.48.

[KDD+13] Andreas Knüpfer, Robert Dietrich, Jens Doleschal, Markus Geimer, Marc-André Hermanns, Christian Rössel, Ronny Tschüter, Bert Wesarg, and Felix Wolf. Generic Support for Remote Memory Access Operations in Score-P and OTF2. In *Tools for High Performance Computing 2012*, pages 57–74. Springer Berlin Heidelberg, 2013. DOI:10.1007/978-3-642-37349-7_5.

[KHMW06] Andrej Kühnal, Marc-André Hermanns, Bernd Mohr, and Felix Wolf. Specification of Inefficiency Patterns for MPI-2 One-Sided Communication. In *Euro-Par 2006 Parallel Processing*, volume 4128 of *Lecture Notes in Computer Science*, pages 47–62. Springer Berlin Heidelberg, 2006. DOI:10.1007/11823285_6.

[Khr19] Khronos OpenCL Working Group. *The OpenCL Specification, Version v2.2-10*, 2019. online at khronos.org, accessed 8 Feb 2019.

[KKN13] Ian Karlin, Jeff Keasler, and Rob Neely. Lulesh 2.0 updates and changes. Technical Report LLNL-TR-641973, Lawrence Livermore National Laboratory, 2013. online at computation.llnl.gov, accessed 11 Feb 2019.

[KLLB13] Karthik Kumar, Jibang Liu, Yung-Hsiang Lu, and Bharat Bhargava. A Survey of Computation Offloading for Mobile Systems. *Mobile Networks and Applications*, 18(1):129–140, 2013. DOI:10.1007/s11036-012-0368-0.

[Kra14] Jiri Kraus. *CUDA Pro Tip: Profiling MPI Applications*. NVIDIA, 2014. online at devblogs.nvidia.com, accessed 11 Feb 2019.

[KW59] James E. Kelley, Jr and Morgan R. Walker. Critical-path Planning and Scheduling. In *Papers Presented at the Eastern Joint IRE-AIEE-ACM Computer Conference*, IRE-AIEE-ACM '59 (Eastern), pages 160–173. ACM, 1959. DOI:10.1145/1460299.1460318.

[LCM+00] Kathleen A. Lindlan, Janice Cuny, Allen D. Malony, Sameer. Shende, Bernd Mohr, Reid Rivenburgh, and Craig Rasmussen. A Tool Framework for Static and Dynamic Analysis of Object-Oriented Software with Templates. In *Proceedings of the 2000 ACM/IEEE Conference on Supercomputing*, pages 49–49. ACM/IEEE, 2000. DOI:10.1109/SC.2000.10052.

[LDTW14] Daniel Lorenz, Robert Dietrich, Ronny Tschüter, and Felix Wolf. A Comparison between OPARI2 and the OpenMP Tools Interface in the Context of Score-P. In *Using and Improving OpenMP for Devices, Tasks, and More*, volume 8766 of *Lecture Notes in Computer Science*, pages 161–172. Springer International Publishing, 2014. DOI:10.1007/978-3-319-11454-5_12.

[LGP+96] Jesús Labarta, Sergi Girona, Vincent Pillet, Toni Cortes, and Luis Gregoris. DiP: A Parallel Program Development Environment. In Luc Bougé, Pierre Fraigniaud, Anne Mignotte, and Yves Robert, editors, *Proceedings of the Second International Euro-Par Conference on Parallel Processing-Volume II*, Euro-Par, pages 665–674. Springer Berlin Heidelberg, 1996. DOI:10.1007/BFb0024763.

[LMCF13] Xu Liu, John Mellor-Crummey, and Michael Fagan. A New Approach for Performance Analysis of OpenMP Programs. In *27th International Conference on Supercomputing*, ICS, pages 69–80. ACM, 2013. DOI:10.1145/2464996.2465433.

[LQPdS10] Chunhua Liao, Daniel J. Quinlan, Thomas Panas, and Bronis R de Supinski. A ROSE-based OpenMP 3.0 research compiler supporting multiple runtime libraries. In *Beyond Loop Level Parallelism in OpenMP: Accelerators, Tasking and More*, pages 15–28. Springer Berlin Heidelberg, 2010. DOI:10.1007/978-3-642-13217-9_2.

[LSW15] Daniel Lorenz, Sergei Shudler, and Felix Wolf. Preventing the Explosion of Exascale Profile Data with Smart Thread-level Aggregation. In *4th Workshop on Extreme Scale Programming Tools*, ESPT, pages 1:1–1:10. ACM, 2015. DOI:10.1145/2832106.2832107.

[LTCS10] Michael Laurenzano, Mustafa M. Tikir, Laura Carrington, and Allan Snavely. PEBIL: Efficient static binary instrumentation for Linux. In *International Symposium on Performance Analysis of Systems Software*, ISPASS, pages 175–183. IEEE, 2010. DOI:10.1109/ISPASS.2010.5452024.

[MBB+12] Dieter an Mey, Scott Biersdorf, Christian Bischof, Kai Diethelm, Dominic Eschweiler, Michael Gerndt, Andreas Knüpfer, Daniel Lorenz, Allen Malony, Wolfgang E. Nagel, Yury Oleynik, Christian Rössel, Pavel Saviankou, Dirk Schmidl, Sameer Shende, Michael Wagner, Bert Wesarg, and Felix Wolf. Score-P: A Unified Performance Measurement System for Petascale Applications. In Christian Bischof, Heinz-Gerd Hegering, Wolfgang E. Nagel, and Gabriel Wittum, editors, *Competence in High Performance Computing 2010*, pages 85–97. Springer Berlin Heidelberg, 2012. DOI:10.1007/978-3-642-24025-6_8.

[MBH+14] Guoyong Mao, David Böhme, Marc-André Hermanns, Markus Geimer, Daniel Lorenz, and Felix Wolf. Catching Idlers with Ease: A Lightweight Wait-State Profiler for MPI Programs. In *Proceedings of the 21st European MPI Users' Group Meeting*, EuroMPI/ASIA, pages 103–108. ACM, 2014. DOI:10.1145/2642769.2642783.

[Mei97] Wagner Meira, Jr. Understanding Parallel Program Performance Using Cause-Effect Analysis. Technical report, University of Rochester, 1997.

[Mes15] Message Passing Interface Forum. *MPI: A Message-Passing Interface Standard, Version 3.1*, 2015. online at mpi-forum.org/docs, accessed 10 Feb 2019.

[MK12] Kathryn Mohror and Karen L. Karavanic. Trace profiling: Scalable event tracing on high-end parallel systems. *Parallel Computing*, 38(4–5):194–225, 2012. DOI:10.1016/j.parco.2011.12.003.

[MLA98] Wagner Meira, Jr., Thomas J. LeBlanc, and Virgílio A. F. Almeida. Using Cause-effect Analysis to Understand the Performance of Distributed Programs. In *Proceedings of the SIGMETRICS Symposium on Parallel and Distributed Tools*, SPDT, pages 101–111. ACM, 1998. DOI:10.1145/281035.281046.

[MLG05] Edu Metz, Raimondas Lencevicius, and Teofilo F. Gonzalez. Performance Data Collection Using a Hybrid Approach. *SIGSOFT Software Engineering Notes*, 30(5):126–135, 2005. DOI:10.1145/1095430.1081729.

[MLP+17] Suejb Memeti, Lu Li, Sabri Pllana, Joanna Kołodziej, and Christoph Kessler. Benchmarking OpenCL, OpenACC, OpenMP, and CUDA: Programming Productivity, Performance, and Energy Consumption. In *Proceedings of the 2017 Workshop on Adaptive Resource Management and Scheduling for Cloud Computing*, ARMS-CC '17, pages 1–6. ACM, 2017. DOI:10.1145/3110355.3110356.

[MMBS16] Deepak Majeti, Kuldeep S. Meel, Rajkishore Barik, and Vivek Sarkar. Automatic Data Layout Generation and Kernel Mapping for CPU+GPU Architectures. In *Proceedings of the 25th International Conference on Compiler Construction*, CC, pages 240–250. ACM, 2016. DOI:10.1145/2892208.2892210.

[MMSW02] Bernd Mohr, Allen D. Malony, Sameer S. Shende, and Felix Wolf. Design and Prototype of a Performance Tool Interface for OpenMP. *The Journal of Supercomputing*, 23(1), 2002. DOI:10.1023/A:1015741304337.

[MP14] Chougule Meenal and Gutte Prashant. Parallel Programming Models: A Systematic Survey. *International Journal of Computer Science and Information Technologies*, 25(4), 2014.

[MSD+] Hans W. Meuer, Erich Strohmaier, Jack J. Dongarra, Host D. Simon, and Martin Meuer. TOP 500 Supercomputer Sites. online at top500.org, accessed 10 Feb 2019.

[MSM05] Allen D. Malony, Sameer S. Shende, and Alan Morris. Phase-Based Parallel Performance Profiling. In *In Proceedings of the International Conference ParCo*, volume 33 of *NIC Series*, pages 203–210. John von Neumann Institute for Computing, 2005. online at pdfs.semanticscholar.org, accessed 11 Feb 2019.

[MW03] Bernd Mohr and Felix Wolf. KOJAK - A Tool Set for Automatic Performance Analysis of Parallel Programs. In *Euro-Par 2003 Parallel Processing*, Lecture Notes in Computer Science, pages 1301–1304. Springer Berlin Heidelberg, 2003. DOI:10.1007/978-3-540-45209-6_177.

[NVI16] *NVIDIA Tesla P100*, 2016. Whitepaper, online at nvidia.com, accessed 9 Feb 2019.

[NVI17] *NVIDIA Tesla V100 GPU Architecture*, 2017. Whitepaper, online at nvidia.com, accessed 9 Feb 2019.

[NVI18a] NVIDIA. *CUDA C Programming Guide, CUDA Toolkit v10.0.130*, 2018. online at docs.nvidia.com, accessed 8 Feb 2019.

[NVI18b] NVIDIA. *CUPTI User's Guide, CUDA Toolkit v10.0.130*, 2018. online at docs.nvidia.com, accessed 8 Feb 2019.

[NVI18c] NVIDIA. *NVIDIA Nsight Compute User Manual, v1.0*, 2018. online at docs.nvidia.com, accessed 18 Feb 2019.

[NVI18d] NVIDIA. *NVML Reference Manual, vR410*, 2018. online at docs.nvidia.com, accessed 25 Feb 2019.

[NVI18e] NVIDIA. *Profiler User's Guide, CUDA Toolkit v10.0.130*, 2018. online at docs.nvidia.com, accessed 8 Feb 2019.

[NVI19a] NVIDIA. *Multi-Process Service, vR418*, 2019. online at docs.nvidia.com, accessed 4 Apr 2019.

[NVI19b] NVIDIA. *NVIDIA Nsight Systems User Guide, Rev. 2019.1.190125*, 2019. online at docs.nvidia.com, accessed 18 Feb 2019.

[OAC15] OpenACC-Standard Organization. *The OpenACC Application Programming Interface, Version 2.5*, 2015. online at openacc.org, accessed 11 Feb 2019.

[OAC18] OpenACC-Standard Organization. *The OpenACC Application Programming Interface, Version 2.7*, 2018. online at openacc.org, accessed 11 Feb 2019.

[OMP15] OpenMP Architecture Review Board. *OpenMP Application Programming Interface, Version 4.5*, 2015. online at openmp.org, accessed 11 Feb 2019.

[OMP18] OpenMP Architecture Review Board. *OpenMP Application Programming Interface, Version 5.0*, 2018. online at openmp.org, accessed 11 Feb 2019).

[PBAL09] Judit Planas, Rosa M. Badia, Eduard Ayguadé, and Jesús Labarta. Hierarchical Task-Based Programming With StarSs. *The International Journal of High Performance Computing Applications*, 23(3):284–299, 2009. DOI:10.1177/1094342009106195.

[PGAB+07] Jelena Pješivac-Grbović, Thara Angskun, George Bosilca, Graham E. Fagg, Edgar Gabriel, and Jack J. Dongarra. Performance analysis of MPI collective operations. *Cluster Computing*, 10(2):127–143, 2007. DOI:10.1007/s10586-007-0012-0.

[PMM+15] Kiran Pamnany, Sanchit Misra, Vasimuddin Md., Xing Liu, Edmond Chow, and Srinivas Aluru. Dtree: Dynamic task scheduling at petascale. In Julian M. Kunkel and Thomas Ludwig, editors, *High Performance Computing*, pages 122–138. Springer International Publishing, 2015.

[PPL+95] Vincent Pillet, Vincent Pillet, Jesús Labarta, Toni Cortes, Toni Cortes, Sergi Girona, and Sergi Girona. PARAVER: A Tool to Visualize and Analyze Parallel Code. Technical report, Universitat Politècnica de Catalunya, 1995. online at pdfs.semanticscholar.org, accessed 11 Feb 2019.

[PWD+16] Antoine Petitet, Clint Whaley, Jack Dongarra, Andy Cleary, and Piotr Luszczek. HPL - A Portable Implementation of the High-Performance Linpack Benchmark for Distributed-Memory Computers, Version 2.2. 2016. online at netlib.org/benchmark/hpl/, accessed 10 Feb 2019.

[RBK16] Michael P. Robson, Ronak Buch, and Laxmikant V. Kale. Runtime Coordinated Heterogeneous Tasks in Charm++. In *Proceedings of the Second Internationsl Workshop on Extreme Scale Programming Models and Middleware*, ESPM2, pages 40–43. IEEE, 2016. DOI:10.1109/ESPM2.2016.011.

[RdCL16] David Rohr, Jan de Cuveland, and Volker Lindenstruth. A Model for Weak Scaling to Many GPUs at the Basis of the Linpack Benchmark. In *2016 IEEE International Conference on Cluster Computing (CLUSTER)*, pages 192–202. IEEE, 2016. DOI:10.1109/CLUSTER.2016.15.

[Res17] Colfax Research. A Performance-Based Comparison of C/C++ Compilers, 2017. online at colfaxresearch.com, accessed 18 Mar 2019.

[RGL14] Claudia Rosas, Judit Giménez, and Jesús Labarta. Scalability Prediction for Fundamental Performance Factors. *Supercomputing Frontiers and Innovations*, 1(2):4–19, 2014. DOI:10.14529/jsfi140201.

[SAF12] Jia Song and Jim Alves-Foss. Performance Review of Zero Copy Techniques. *International Journal of Computer Science and Security (IJCSS)*, 6(4), 2012. cscjournals.org, accessed 11 Feb 2019.

[Sch05] Martin Schulz. Extracting Critical Path Graphs from MPI Applications. In *International Cluster Computing*, pages 1–10. IEEE, 2005. DOI:10.1109/CLUSTR.2005.347035.

[Sch13] Felix Schmitt. Critical-Path Analysis and Optimization Guidance for CUDA Programs. Diplomarbeit. Technische Universität Dresden, 2013.

[Sco18] Score-P developer community. *Score-P User Manual 4.1 (revision 13848)*, 2018. DOI:10.5281/zenodo.1456612.

[SDJ16] Felix Schmitt, Robert Dietrich, and Guido Juckeland. Scalable critical-path analysis and optimization guidance for hybrid MPI-CUDA applications. *The International Journal of High Performance Computing Applications*, 2016. DOI:10.1177/1094342016661865.

[SDS15] Felix Schmitt, Robert Dietrich, and Jonas Stolle. Integrating Critical-Blame Analysis for Heterogeneous Applications into the Score-P Workflow. In *Tools for High Performance Computing 2014*, pages 161–173. Springer International Publishing, 2015. DOI:10.1007/978-3-319-16012-2_8.

[SKVM15] Pavel Saviankou, Michael Knobloch, Anke Visser, and Bernd Mohr. Cube v4: From Performance Report Explorer to Performance Analysis Tool. *Procedia Computer Science*, 51:1343–1352, 2015. DOI:10.1016/j.procs.2015.05.320.

[SLG+14] Harald Servat, Germán Llort, Juan González, Judit Giménez, and Jesús Labarta. Identifying Code Phases Using Piece-Wise Linear Regressions. In *28th International Parallel and Distributed Processing Symposium*, pages 941–951. IEEE, 2014. DOI:10.1109/IPDPS.2014.100.

[SLH+13] Harald Servat, Germán Llort, Kevin Huck, Judit Giménez, and Jesús Labarta. Framework for a Productive Performance Optimization. *Parallel Computing*, 39(8):336–353, 2013. DOI:10.1016/j.parco.2013.05.004.

[SM06] Sameer S. Shende and Allen D. Malony. The Tau Parallel Performance System. *The International Journal of High Performance Computing Applications*, 20(2):287–311, 2006. DOI:10.1177/1094342006064482.

[SM14] Robert Schöne and Daniel Molka. Integrating Performance Analysis and Energy Efficiency Optimizations in a Unified Environment. *Computer Science - Research and Development*, 29(3–4):231–239, 2014. DOI:10.1007/s00450-013-0243-7.

[SPP19] SCALASCA performance properties. Online Help, 2019. apps.fz-juelich.de, accessed 11 Feb 2019.

[SRC17] N. V. Sunitha, K. Raju, and Niranjan N. Chiplunkar. Performance improvement of CUDA applications by reducing CPU-GPU data transfer overhead. In *International Conference on Inventive Communication and Computational Technologies (ICICCT)*, pages 211–215. IEEE, 2017. DOI:10.1109/ICICCT.2017.7975190.

[SSD14] Felix Schmitt, Jonas Stolle, and Robert Dietrich. CASITA: A Tool for Identifying Critical Optimization Targets in Distributed Heterogeneous Applications. In *43rd International Conference on Parallel Processing Workshops*, ICPPW, pages 186–195. IEEE, 2014. DOI:10.1109/ICPPW.2014.35.

[SSR95] Gautam Shah, Aman Singla, and Umakishore Ramachandran. The Quest for a Zero Overhead Shared Memory Parallel Machine. In *Proceedings of the 1995 International Conference on Parallel Processing*, pages 194–201. CRC Press, 1995.

[SWW09] Zoltán Szebenyi, Felix Wolf, and Brian J. N. Wylie. Space-efficient Time-series Call-path Profiling of Parallel Applications. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, SC, pages 37:1–37:12. ACM, 2009. DOI:10.1145/1654059.1654097.

[TAMC10] Nathan R. Tallent, Laksono Adhianto, and John M. Mellor-Crummey. Scalable Identification of Load Imbalance in Parallel Executions Using Call Path Profiles. In *International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–11. IEEE, 2010. DOI:10.1109/SC.2010.47.

[Tho99] Mikkel Thorup. Undirected Single-source Shortest Paths with Positive Integer Weights in Linear Time. *Journal of the ACM*, 46(3):362–394, May 1999. DOI:10.1145/316542.316548.

[THW10] Jan Treibig, Georg Hager, and Gerhard Wellein. LIKWID: A Lightweight Performance-Oriented Tool Suite for x86 Multicore Environments. In *39th International Conference on Parallel Processing Workshops*, pages 207–216. IEEE, 2010. DOI:10.1109/ICPPW.2010.38.

[TJYD10] Dan Terpstra, Heike Jagode, Haihang You, and Jack Dongarra. Collecting Performance Data with PAPI-C. In *Tools for High Performance Computing 2009*, pages 157–173. Springer Berlin Heidelberg, 2010. DOI:10.1007/978-3-642-11261-4_11.

[TMCP10] Nathan R. Tallent, John M. Mellor-Crummey, and Allan Porterfield. Analyzing Lock Contention in Multithreaded Applications. *SIGPLAN Notices*, 45(5):269–280, 2010. DOI:10.1145/1837853.1693489.

[Vet00] Jeffrey Vetter. Performance Analysis of Distributed Applications Using Automatic Classification of Communication Inefficiencies. In *Proceedings of the 14th International Conference on Supercomputing*, ICS, pages 245–254. ACM, 2000. DOI:10.1145/335231.335255.

[WBB12] Matthias Weber, Ronny Brendel, and Holger Brunst. Trace File Comparison with a Hierarchical Sequence Alignment Algorithm. In *10th International Symposium on Parallel and Distributed Processing with Applications*, ISPA, pages 247–254. IEEE, 2012. DOI:10.1109/ISPA.2012.40.

[WBW+17] Matthias Weber, Ronny Brendel, Michael Wagner, Robert Dietrich, Ronny Tschüter, and Holger Brunst. Visual Comparison of Trace Files in Vampir. In *6th Workshop on Extreme-Scale Programming Tools*, pages 334–341, 2017. DOI:10.1109/IDAACS.2015.7340754.

[WDK15] Michael Wagner, Jens Doleschal, and Andreas Knüpfer. MPI-focused Tracing with OTFX: An MPI-aware In-memory Event Tracing Extension to the Open Trace Format 2. In *Proceedings of the 22nd European MPI Users' Group Meeting*, EuroMPI, pages 7:1–7:8. ACM, 2015. DOI:10.1145/2802658.2802664.

[WF13] Brian J. N. Wylie and Wolfgang Frings. Scalasca Support for MPI+OpenMP Parallel Applications on Large-scale HPC Systems Based on Intel Xeon Phi. In *Proceedings of the Conference on Extreme Science and Engineering Discovery Environment: Gateway to Discovery*, XSEDE, pages 37:1–37:8. ACM, 2013. DOI:10.1145/2484762.2484777.

[WM03] Felix Wolf and Bernd Mohr. Automatic performance analysis of hybrid MPI/OpenMP applications. *Journal of Systems Architecture*, 49(10–11):421–439, 2003. DOI:10.1016/S1383-7621(03)00102-4.

[WMDM07] Felix Wolf, Bernd Mohr, Jack Dongarra, and Shirley Moore. Automatic analysis of inefficiency patterns in parallel applications. *Concurrency and Computation: Practice and Experience*, 19(11):1481–1496, 2007. DOI:10.1002/cpe.1128.

[WMS+13] Matthias Weber, Kathryn Mohror, Martin Schulz, Bronis R. de Supinski, Holger Brunst, and Wolfgang E. Nagel. Alignment-Based Metrics for Trace Comparison. In *19th International Conference on Parallel Processing*, Euro-Par, pages 29–40. Springer, Berlin, Heidelberg, 2013. DOI:10.1007/978-3-642-40047-6_6.

[WPB+14] Hao Wang, S. Potluri, D. Bureddy, C. Rosales, and D.K. Panda. GPU-Aware MPI on RDMA-Enabled Clusters: Design, Implementation and Evaluation. *Transactions on Parallel and Distributed Systems*, 25(10):2595–2605, 2014. DOI:10.1109/TPDS.2013.222.

[YM88] Cui-Qing Yang and Barton P. Miller. Critical Path Analysis for the Execution of Parallel and Distributed Programs. In *8th International Conference on Distributed Computing Systems*, pages 366–373. IEEE, 1988. DOI:10.1109/DCS.1988.12538.

[ZWW+16] Erik Zenker, Benjamin Worpitz, René Widera, Axel Huebl, Guido Juckeland, Andreas Knüpfer, Wolfgang E. Nagel, and Michael Bussmann. Alpaka – An Abstraction Library for Parallel Kernel Acceleration. In *2016 IEEE International Parallel and Distributed Processing Symposium Workshops*, IPDPSW, pages 631–640. IEEE, 2016. DOI:10.1109/IPDPSW.2016.50.

List of Figures

List of Tables

Appendix

```
                                    Region      Calls     Time[s]  Time on CP  CP[%]    Blame[s] Blame[%] Blame on CP
----------------------------------------
                            buildKKRMatrix       8192 2455.257773  619.716602  49.44 2468.267054  50.8300  634.839855
 -> 99.36% (2452.413393 s) blame for device idle
            zgemm_sm35_ldg_nn_32x8x64x8x16     466944 1426.074705  358.702875  28.62 1422.916685  29.3027  358.056730
 -> 100.00% (1422.916685 s) blame for host wait
                            zblock_lu_cuda       8192  493.225056  123.082706   9.82  232.430286   4.7865   60.667441
 -> 98.64% (229.269034 s) blame for device idle
 -> 1.36% (3.161252 s) blame for MPI collective
                      cuStreamSynchronize     180224 1678.477553   54.807941   4.37  502.490095  10.3480   56.139691
 -> 97.86% (491.719654 s) blame for device idle
 -> 2.14% (10.770441 s) blame for MPI collective
!$omp for @energyContourIntegration.cpp:3
                                         57        256  109.547484   27.388507   2.18  106.775116   2.1989   27.223993
 -> 99.34% (106.074481 s) blame for device idle
          calculateSingleScattererSolution       8192  188.362740   46.872688   3.74  111.123326   2.2884   24.030177
 -> 100.00% (111.123326 s) blame for device idle
                                  lsms_init          4   10.174010    2.929630   0.23    1.560249   0.0321    1.560249
 -> 31.25% (0.487642 s) blame for MPI collective
 -> 68.75% (1.072606 s) blame for MPI late sender
                                   MPI_Send        960    2.792859    0.933992   0.07    2.200557   0.0453    0.738311
 -> 100.00% (2.200557 s) blame for device idle
                     cuMemcpyHtoDAsync_v2     180224    2.801090    0.737798   0.06    1.437335   0.0296    0.388270
 -> 98.69% (1.418459 s) blame for device idle
 -> 1.31% (0.018876 s) blame for MPI collective
                    recalculateCoreStates          8    1.951704    0.492233   0.04    1.200750   0.0247    0.341057
 -> 94.44% (1.134016 s) blame for device idle
 -> 5.56% (0.066734 s) blame for MPI collective
----------------------------------------
                             Sum of Top 10     853196 6368.664975 1235.664972  98.58 4850.401453  99.8862 1163.985773
                               on host:  9     386252 4942.590270  876.962097  69.96 3427.484767  70.5836  805.929043
                             on device:  1     466944 1426.074705  358.702875  28.62 1422.916685  29.3027  358.056730

Stream summary: 4 MPI rank(s), 4 host stream(s), 4 device(s)
Total program runtime          : 1253.505795 s
Total waiting time (host)      : 1710.790897 s (427.697724 s per rank, 427.697724 s (34.12%) per host stream)
                                               421.636662 s on rank 2 (min)
                                               431.544662 s on rank 0 (max)

Pattern summary:
 MPI wait patterns             :   32.313162 s (8.078290 s per rank; 321 overall occurrences)
  Late sender                  :    1.082156 s (0.270539 s per rank; 101 overall occurrences)
  Late receiver                :    0.029210 s (0.007303 s per rank; 94 overall occurrences)
  Wait in MPI collective       :   31.201795 s (7.800449 s per rank; 126 overall occurrences)
 Offloading
  Idle device                  : 3292.222629 s (66.35% -> 823.055657 s per device)
  Compute idle device          : 3535.520363 s (71.26% -> 883.880091 s per device)
  Early blocking wait          : 1456.267500 s (364.066875 s per device, 171943 overall occurrences)
  -on compute kernels          : 1422.916685 s (355.729171 s per device)
  Total communication time     :  243.297734 s (4.90% of offload time)
  -Exclusive communication     :  243.297734 s (100.00% of total communication time)
  -Blocking communication      :    0.000182 s (4 occurrences), exclusive: 0.000182 s
  -Consecutive communication   :  232.601919 s (188408 occurrences)
  No overlap between copy and compute tasks!
  Kernel startup delay         :    0.927433 ms/kernel (448712 occurrences), total delay: 416.150494 s
  No overlap between compute tasks!
```

Listing 2: CASITA summary for GCC-compiled LSMS with 128 atoms on 4 GPUs (single-threaded)

```
HPLinpack benchmark input file
Innovative Computing Laboratory, University of Tennessee
HPL.out      output file name (if any)
6            device out (6=stdout,7=stderr,file)
1            # of problems sizes (N)
__N__        Ns
1            # of NBs
1024         NBs
0            PMAP process mapping (0=Row-,1=Column-major)
1            # of process grids (P x Q)
__P__        Ps
__Q__        Qs
16.0         threshold
1            # of panel fact
2            PFACTs (0=left, 1=Crout, 2=Right)
1            # of recursive stopping criterium
1            NBMINs (>= 1)
1            # of panels in recursion
4            NDIVs
1            # of recursive panel fact.
2            RFACTs (0=left, 1=Crout, 2=Right)
1            # of broadcast
3            BCASTs (0=1rg,1=1rM,2=2rg,3=2rM,4=Lng,5=LnM)
1            # of lookahead depth
0            DEPTHs (>=0)
2            SWAP (0=bin-exch,1=long,2=mix)
256          swapping threshold
1            L1 in (0=transposed,1=no-transposed) form
0            U  in (0=transposed,1=no-transposed) form
1            Equilibration (0=no,1=yes)
8            memory alignment in double (> 0)
```

Listing 3: HPL.dat template for HPL-CUDA experiments

```
HPLinpack benchmark input file
Innovative Computing Laboratory, University of Tennessee
HPL.out      output file name (if any)
6            device out (6=stdout,7=stderr,file)
1            # of problems sizes (N)
__N__        Ns
1            # of NBs
256          NBs
0            PMAP process mapping (0=Row-,1=Column-major)
1            # of process grids (P x Q)
__P__        Ps
__Q__        Qs
16.0         threshold
1            # of panel fact
0            PFACTs (0=left, 1=Crout, 2=Right)
1            # of recursive stopping criterium
2            NBMINs (>= 1)
1            # of panels in recursion
2            NDIVs
1            # of recursive panel fact.
0            RFACTs (0=left, 1=Crout, 2=Right)
1            # of broadcast
0            BCASTs (0=1rg,1=1rM,2=2rg,3=2rM,4=Lng,5=LnM)
1            # of lookahead depth
0            DEPTHs (>=0)
2            SWAP (0=bin-exch,1=long,2=mix)
128          swapping threshold
0            L1 in (0=transposed,1=no-transposed) form
0            U  in (0=transposed,1=no-transposed) form
1            Equilibration (0=no,1=yes)
8            memory alignment in double (> 0)
```

Listing 4: HPL.dat template for HPL 2.2 experiments

Abbreviations

API An **Application Programming Interface** is an interface between software components, which defines routines, data structures, variables, compiler directives, and environment variables to abstract from an underlying implementation.

CASITA The **Critical path AnalySIs Tool for heterogeneous Applications** is a trace analyzer, which has been developed as part of this thesis. (See Section 6.3.)

CPU The **Central Processing Unit** is the main processor within a computer. It can control special purpose processors such as GPUs.

CUDA **Compute Unified Device Architecture** refers to "a general purpose parallel computing platform and programming model"[NVI18a] developed by NVIDIA for programming of GPUs. This work refers mostly to CUDA's offloading API. (See Section 2.3.2)

CUPTI The **CUDA Profiling Tools Interface** is an interface which enables performance tools to acquire runtime information from CUDA-related operations on the host (CPU) and CUDA-capable compute devices (NVIDIA GPUs).

EDG An **Event Dependency Graph** is a weighted, directed, acyclic graph, where vertices represent events and edges represent dependencies. (See Definition 3 in Section 5.1.1)

GPGPU **General-Purpose Computing on Graphics Processing Units** refers to the use of graphics processors for general computations, which have traditionally been executed on CPUs.

GPU A **Graphics Processing Unit** is a many-core processor that is traditionally optimized for graphics processing. Modern GPUs are also capable of general purpose computing and can efficiently process massively data-parallel workloads. (See Section 2.2.1)

HPC **High-Performance Computing** refers to parallel computing on distributed resources, which exceeds the capabilities of desktop computers or workstations.

MPI The **Message Passing Interface** is an API for message passing in parallel applications. It is the defacto standard for communication between processes in HPC. (See Section 2.3.5)

MPMD **Multiple Programs, Multiple Data** is a style of parallel programming, where different programs are executed in parallel on different input data.

OMPT **OpenMP Tools** is "an interface that helps a first-party tool monitor the execution of an OpenMP program."[OMP18] It covers all aspects in the OpenMP specification that are related to performance analysis.

OpenACC **Open Accelerators** is a directive-based, portable programming interface for heterogeneous platforms. It is mainly used to offload massively-parallel workloads to accelerators such as GPUs. (See Section 2.3.3.)

OpenCL The **Open Compute Language** is a portable programming interface for heterogeneous platforms. This work refers mostly to its offloading API. (See Section 2.3.2)

OpenMP **Open Multi-Processing** is an API for multithreading and the defacto standard for shared-memory programming in HPC. It includes an offloading API since version 4.0 of its specification. (See Section 2.3.3)

OTF2 **Open Trace Format 2** refers to a file format for logging of event traces. The software package consists of a library and several tools.

SPMD **Single Program, Multiple Data** is a common parallelization style, where the same program is executed in parallel on different input data. It is usually used in conjunction with message passing.

Acknowledgments

First of all, I would like to express my sincere gratitude to my doctoral supervisor Prof. Dr. Wolfgang Nagel for the continuous support of my PhD study, his immense knowledge, and the excellent research environment at the ZIH. I would also like to thank Prof. Dr. Matthias Müller for serving as the second reviewer.

My sincere thanks goes to my advisor Dr. Matthias Lieber. Your constructive criticism and suggestions significantly influenced the content of this thesis and improved its quality. Besides my advisor, I would also like to thank Dr. Andreas Knüpfer for his encouragement, insightful comments, and advises. A very special gratitude goes out to my friend and former colleague Felix Schmitt. Many thanks for proof-reading most of the dissertation, your suggestions, and your keen eye for details.

Furthermore, I would like to thank all my colleagues for the stimulating discussions and all the fun we have had in the past years. Special thanks are due to my friends and longtime office mates Ronny Tschüter and Frank Winkler for the relaxed but also productive atmosphere at work. Thanks also for fruitful discussions and proof-reading parts of the thesis. I would also like to thank my former mentor and colleague Guido Juckeland for his motivation and advises, Holger Brunst for valuable feedback through his extensive knowledge about performance analysis, and Jens Lukaschkowitz for improving equations and definitions and for ensuring the constant availability of coffee.

Last but not the least, I would like to thank my family for their understanding and support during this exhausting time, and of course my two little boys for being as charming and cheerful as their mother.