

Analyzing Memory Accesses for Performance and Correctness of Parallel Programs

Von der Fakultät für Mathematik, Informatik und Naturwissenschaften der
RWTH Aachen University zur Erlangung des akademischen Grades
eines Doktors der Naturwissenschaften genehmigte Dissertation

vorgelegt von

Diplom-Ingenieur

Tim Cramer

aus Werdohl

Berichter: Univ.-Prof. Dr. rer. nat. Matthias S. Müller
Univ.-Prof. Dr. ir. Joost-Pieter Katoen

Tag der mündlichen Prüfung: 5. Juli 2017

Diese Dissertation ist auf den Internetseiten der Universitätsbibliothek online verfügbar.

Kurzfassung

Der stetig wachsende Bedarf an Rechenleistung im wissenschaftlichen Umfeld hat im laufenden Jahrzehnt sowohl zu einer weiten Verbreitung als auch hohen Akzeptanz von hochparallelen Computerarchitekturen geführt. Dieser Trend ist auch in der TOP500-Liste der leistungsfähigsten Supercomputer der Welt manifestiert, in welcher über 40 % der Gesamt-Performance aus Akzelerator-basierten Systemen resultiert. Die Programmierung dieser Systeme erforderte in der Vergangenheit häufig zeitaufwändige Anpassungen der rechenintensiven Programmteile, bevor produktivere Ansätze wie OpenACC oder die Offloading-Direktiven in OpenMP aufkamen. Jedoch bleibt auch mit diesen nutzerfreundlicheren Ansätzen die Programmierung für heterogene Architekturen komplex und fehleranfällig und stellt viele Anforderungen an den Programmierer, der eine hohe Performance für seine Anwendung erreichen will.

Eine Schlüsselrolle für das Verständnis der Performance und der Korrektheit eines parallelen Programms spiegelt sich in der Analyse der Speicherzugriffe wieder. Diese Arbeit verfolgt einen ganzheitlichen Ansatz unter Berücksichtigung der Hardware-Eigenschaften, des Programmierparadigmas, der zugrundeliegende Implementierung und der Schnittstelle für eine adäquate Tool-Unterstützung in Bezug auf beide Aspekte. Die Verbesserung der Performance und die Validierung einer Anwendung erfordert hierbei ein tiefgehendes Verständnis des dynamischen Laufzeitverhaltens. Hierbei ist das adäquate Platzieren der Daten und Threads essentiell für die Performance, und die Zugriffsreihenfolge essentiell für das deterministische Verhalten bzw. die Korrektheit einer Anwendung. Aus diesem Grund wird diese Arbeit zunächst eine systematische Methodik zur Bewertung von OpenMP Target-Devices, Muster für die effiziente Task-parallele Programmierung von *Non-Uniform Memory Access* (NUMA) Architekturen, sowie Verbesserungen für eine standardkonforme Tool-Unterstützung präsentieren. Basierend auf den gewonnenen Erkenntnissen, wird im Anschluss ein OpenMP Epochen-Modell für die Korrektheitsanalyse definiert, welches die Semantik inklusive des Laufzeit- und Speichermodells von OpenMP berücksichtigt. Die Evaluierung der entwickelten Konzepte erfolgt an Hand von relevanten Tools zur Performance- und Korrektheitsanalyse.

Stichwörter: HPC, OpenMP, Tools, Performance-Analyse, Korrektheitsanalyse, parallele Programmierung

Abstract

The demand for large compute capabilities in scientific computing led to wide use and acceptance of highly-parallel computer architectures during the last decade. This trend is manifested in the TOP500, listing the fastest supercomputer of the world, in which about 40 % of the performance share results from accelerator-based systems. Programming for these architectures in the past often required a time-consuming rewrite of the compute-intensive application parts, until more productive approaches like *Open Accelerators* (OpenACC) or the target offloading features of *Open Multi-Processing* (OpenMP) came to existence. However, parallel programming for heterogeneous architectures is still a complex and error-prone task, posing several challenges to the programmer who wants to achieve high application performance.

One key factor for the understanding of the performance and the correctness of a parallel program is reflected in the analysis of the memory accesses. This work takes a holistic view on the hardware properties, the programming paradigm, its particular implementation and the interfaces for an adequate tool support with respect to both aspects. The improvement of the performance and the validation of an application requires a deep comprehension of the dynamic runtime behavior. Here, the appropriate data and thread placement is essential for the performance, and the order of the memory accesses is essential for the deterministic behavior or rather the correctness of the application. Therefore, this work will first present a systematic methodology for the assessment of OpenMP for target devices, patterns for the efficient usage of task-based programming on *Non-Uniform Memory Access* (NUMA) architectures, and the improvement of standard-compliant tool support. Based on the gathered insights, an OpenMP epoch model for correctness checking is defined, which respects the OpenMP semantics including the runtime and memory model. The evaluation of the developed concepts is shown by application to real-world performance analysis and correctness checking tools.

Keywords: HPC, OpenMP, Tools, Performance Analysis, Correctness Checking, Parallel Programming

Contents

1. Introduction	1
1.1. Motivation and Goals	2
1.2. User Application Studies	3
1.3. Contributions	6
1.4. Thesis Structure	7
2. Efficient Memory Access in OpenMP	9
2.1. Overview of OpenMP	9
2.2. System Architectures	12
2.3. Benchmarks	15
2.4. Systematic Assessment of OpenMP for Target Devices	21
2.5. Patterns for Task-parallel Programming on NUMA Architectures	35
2.6. Summary and Conclusion	44
3. Improving Standard-compliant Tool Support	45
3.1. Interface Designs	47
3.2. Overview about OMPT	47
3.3. Extension for OpenMP Target Devices	52
3.4. Evaluation of the OMPT Extension	61
3.5. Summary and Conclusion	67
4. Epoch Model for OpenMP	69
4.1. Terminology	69
4.2. Thread-local Epoch Generation	74
4.3. Epoch Merging	80
4.4. Evaluation of the Concept	82
4.5. Extension for Task-based Programs	88
4.6. Epoch Concept for Further Parallel Programming Paradigms	95
4.7. Summary and Conclusion	100
5. Tool Supported Analysis	103
5.1. Correctness Checking	103
5.1.1. Related Work	104
5.1.2. Instrumentation-based Memory Access Tracing	106
5.1.3. OpenMP-specific Error Detection: Methods and Algorithms	108
5.1.4. Epoch-based Analysis of the OpenMP Specification	119
5.2. Performance Analysis	122

Contents

6. Summary and Conclusion	125
Statement of Originality	129
List of Figures	133
List of Tables	135
Listings	137
Bibliography	139
A. Appendix: OMPT Record Types	149

1. Introduction

Since Seymour Cray developed the first supercomputers back in the 1960s, *High Performance Computing* (HPC) evolved as powerful tool for many scientific and engineering research areas. Mechanical and electrical engineering, molecular dynamics, energy, climate and weather predictions, health and bioinformatics, computational chemistry, geophysics, deep learning or financial services are only some of the fields where supercomputers are adopted nowadays. Although the costs for the resources for such computer-aided simulations are very high because of the expensive hardware components and the high energy consumption, they are still cost efficient. Many technical systems or scientific interrelationships are too complex for an analytical consideration and thus simplified approximations can be modeled and computed in order to gain scientific insights. In engineering applications the use of supercomputers often avoids time-consuming and cost-intensive prototype constructions, because different parameters of mechanical or electrical devices can be simulated first.

This ongoing demand for large compute capabilities in many scientific simulations led to a wide use and acceptance of new parallel computer architectures during the last decade. As formulated in the famous Moore's Law in 1965, the number of transistors per silicon chip still doubles roughly every 18 month. However, a performance increase is not reachable anymore by increasing clock speeds, because of the thermal dissipation and parasitic effects of the components. One of the major challenges on the road to Exascale is the excessive power consumption of such a system. Thus, it is still a strong requirement to improve the energy efficiency of HPC systems in terms of the ratio of performance and power consumption. One possible solution for this challenge is the usage of highly-parallel many-core architectures, which already manifested in existence of more and more accelerator-based supercomputers. The fact that these systems have a performance share of about 40 % in the latest TOP 500¹ list of the fastest supercomputers of the world substantiates this observation. However, the existence of innovative, energy efficient and highly-parallel architectures also requires powerful programming paradigms in order to enable the simulation developers to express parallelism for a broad range of algorithms. In a popular article published in 2005, Herb Sutter formulated this fundamental turn toward concurrency in software as "The Free Lunch is over" [86]. The application of parallel programming paradigms often requires a time-consuming rewrite of parts of application in order to achieve a good performance. Directive-based approaches like OpenMP [71] or OpenACC [70] promise to shift some burden of the effort from the developer to the compiler and runtime system.

¹<https://www.top500.org>, List from November 2016

1.1. Motivation and Goals

The portability and user-friendliness of new programming approaches also require methods for the analysis of applications. This includes methodologies for the performance assessment of parallel architectures and scalable tools for the performance analysis of applications, but also tools which enable the correctness validation of parallel programs. The reason is that with an increasing amount of features in the programming paradigm and increasing parallelism of the architecture the programming complexity grows and thus applications tend to be more error-prone. Since many of those applications need to process huge amounts of data, a deep analysis of the memory accesses is required in order to enable an efficient usage of the available hardware. This analysis can either be done by a systematic methodology which includes the evaluation and analysis of the performance characteristics of a given architecture or by a tool-driven approach. For the latter one the major challenge is the early availability of powerful performance analysis and correctness checking tools, which is often not given for new micro-architectures. The reason is that the development time for such powerful and scalable tools might take a long period of time, because of vendor-specific constraints like proprietary interfaces or communication protocols. In order to decrease the period of time between the appearance of a new architecture and the availability of a corresponding analysis tool, vendor- and platform-independent (or rather standard-compliant) approaches are required. This allows tool developers to rely on a set of basic data acquiring methods for a given parallel programming paradigm.

In this thesis it will be shown that the acquired data about the memory accesses of an application can be used as solid foundation for performance evaluations and for correctness checking. This work develops a general and holistic approach, which is reliable, scalable, portable and vendor-independent. Since the single node performance of an application is important for any kind of HPC applications the focus of this work is on OpenMP, which is the de-facto standard for parallel shared memory programming. Here, the complete set of features of the specification is taken into account in order to support the complete parallel expressiveness of the paradigm. This includes the applicability of the tasking concept on large NUMA systems, the programming for heterogeneous architectures as well as the hierarchical nesting of parallelism. Although the focus is on OpenMP, many of the developed concepts are not limited to this paradigm. The transferability to other parallel programming languages will be discussed as well.

1.2. User Application Studies

This thesis is motivated by the analysis of two real-world simulation codes from RWTH Aachen University: the *innovative Modern Object Oriented Solver Environment* (iMOOSE) suite, developed by the Institute for Electrical Machines (IEM), and a *Finite Element Method* (FEM) solver for the efficient computation of the gear tooth contact, developed by the Laboratory for Machine Tools and Production Engineering (WZL). It will stress the importance of interdisciplinary solutions, which combine the expertise of domain specialists and experts in HPC.

Hybrid Parallelization of a Simulation Package for the Construction of Electrical Machines

For the construction and optimization of electrical machines a high degree of effectiveness is a mandatory requirement. The different mechanisms leading to a decrease of this effectiveness can be analyzed analytically, numerically or with measurement techniques. However, many geometries are too complex for an analytical analysis and the construction of a corresponding prototype is time-consuming and cost-intensive. Thus, the Institute for Electrical Machines (IEM)² of RWTH Aachen University developed the in-house FEM package iMOOSE [90] in order to have a tool for the computation of the complex energy loss without the requirement for experimental measurements. Here, not only the optimization of single components of the machine with respect to their efficiency and power density is of certain interest, but also the holistic optimization of the complete system. This results in an increasing demand for more compute capabilities. Since also the complexity and the degree of parallelism of modern supercomputers increase, the iMOOSE package was parallelized in an interdisciplinary work between domain experts and computer scientists [12, 11]. Nowadays, many supercomputers consist of clustered compute nodes, where the same memory is only shared within a single node. In order to respect this system architecture, the parallelization combines two parallel programming paradigms: *Message Passing Interface* (MPI) [63] for the communication and parallel execution between the nodes and OpenMP for utilization of each shared memory node.

For the intra node parallelization the most compute-intensive parts like the assembly of the system matrix, the add of the Jacobian matrix, the solving of the system of linear equations and the computation of the curl of the magnetic vector potential for the determination of the magnetic flux density have been parallelized on loop-level with OpenMP. Here, the specific challenges were the creation of a thread-safe code basis of the huge C++ legacy code, the guarantee of an appropriate data encapsulation and the architecture-specific performance optimizations. Furthermore, one of the most important design objectives was to keep the maximum flexibility

²<http://www.iem.rwth-aachen.de>

1. Introduction

for the approximation of the solution of the linear equation system, because the efficiency and the convergence behavior strongly depends on the current use case of the FEM package. One of used algorithms for this approximation is a *Conjugate Gradient* (CG) method, which also will be used in this thesis as a representative for a real-world compute kernel (refer to Section 2.3).

The inter node parallelization uses a domain decomposition of the *Finite Element* (FE) mesh, where each process is working on a particular sub mesh exclusively. Here, the relative motion between stator and rotor in electrical machines requires a flexible representation of the elements. This is important, because the position of each FE changes with every time step and thus the neighbor relation of the elements have to be adopted correspondingly. In a domain decomposition this will lead to a higher communication overhead if the initial mesh distribution of the elements is not accurate. In order to avoid the expensive communication across the mesh domains, the elements of the stator and the rotor are assigned to different domains, where the air-gap elements in between are assigned either to the stator or the rotor. This results in an adequate load balance during the complete simulation time.

On the one hand the insights of this user study stress the importance for an efficient data access. On the other hand they show the challenges domain experts are faced to with respect the correctness of a real-world hybrid parallel application, because the complete simulation package was never designed to ensure the required thread-safety. This motivates the development of convenient methodologies a programmer can rely on to ensure both: performance and correctness.

Library Support for the Simulation of Gear Tooth Contact on highly-Parallel Systems

For the efficient computation of the bearing behavior and noise excitation of the gear tooth contact, the Laboratory for Machine Tools and Production Engineering (WZL)³ of RWTH Aachen University developed the FEM based application package ZaKo3D [13]. In an iterative workflow the package is used for the compute-aided optimization step of the construction of the 3D gear contact characteristics in order to improve the gear geometry. Based on the geometric data of the flank and a corresponding FE model of the gear section, the simulation determines the contact distances, loads and deflections on the teeth. For the determination of the load a corresponding weight is put on each node of the FE model for the x-, y- and z-direction. This results in a linear equation system for each of the load cases, which can be solved by different direct or iterative solving algorithms. In an interdisciplinary work [21] practice-oriented approaches for the approximation of all load cases were analyzed in detail.

³<http://www.wzl.rwth-aachen.de>

Numerical analysis offers a huge bandwidth of algorithms and methods for the approximation of a linear equation system. The choice for an appropriate approach depends on the numerical characteristics of the system, the size, the amount of right-hand sides, the density of the matrix and the available computer architecture. Direct solving algorithm have the advantage that the linear equations system only has to be solved once for the general case and a solution for all load cases (i.e., for each right-hand side) can be determined by back substitution. In order to reduce the memory overhead, ZaKo3D used the in-house frontal solving algorithm [47] FINEL. However, this solver has the disadvantage that it is pure sequential and thus does not fit on modern parallel architectures. In [21] first the use of an iterative method (the CG algorithm presented in Section 2.3) was evaluated. Compared to the direct approach, the solving of a single equations system is much faster. However, since a complete set of equation systems which only differ in the right-hand side has to be solved, this iterative method was not an appropriate solution. Since the effort for the development and maintenance of an own parallel direct solver was too high, the decision was made in favor to the *Parallel Direct Sparse Solver* (PARDISO) package [79]. The package provides an interface for the solving of sparse symmetric and asymmetric linear equations systems. Here, the advantage is that it possible to hand over any number of right-hand sides, which enables the library to optimize the solving process by applying an adequate decomposition. This led to a speedup of about 45 on 8-core machine compared to the sequential solving with FINEL. In order to prepare the simulation code for future parallel architectures, the memory and thread placement has been analyzed on huge NUMA system. For the performance on such a systems it is essential to avoid remote data accesses across the NUMA domains, as will be shown in Chapter 2 of this thesis. However, in this case a third-party library is used, where the access patten for the solving algorithm is unknown. Thus, also the optimal distribution of the data across the domains is unknown. Nevertheless, by analyzing the distribution of the memory an optimization was possible, which increased the performance by another factor of 3.5 on 128-core machine compared to the unoptimized version on the same machine.

These results form another example which shows the importance of interdisciplinary solutions, without describing all the details. The knowledge of the domain experts about the algorithm requirements and the knowledge of the computer scientists led to a significant improvement of the performance. On the one hand, this enables deeper scientific insights, because more complex simulations became possible. On the other hand, the parallel resources of the RWTH Compute Cluster can be used more efficiently. However, the effort for the optimization of the domain-specific simulations could be further improved by the availability of appropriate performance analysis tools, which allow a more convenient analysis (even for third-party libraries) of the memory access and data distribution in modern supercomputers. This motivates the further investigations which will be demonstrated in this thesis.

1.3. Contributions

My work contributes the following additions to the state of the art analysis of memory access in parallel programs.

First, the performance of features introduced with the OpenMP 4.5 standard has been analyzed and its application has been improved with respect to the *efficiency of the memory access on different architectures*. On the one hand this includes a systematic methodology for the assessment of OpenMP target devices by determining the basic performance characteristics of a new micro-architecture, assessing the paradigm-specific overheads, analyzing the scalability, predicting the performance by using an appropriate model and evaluating the performance with selected standard benchmark suites. On the other hand this includes the analysis of tasking patterns on large NUMA-based systems with respect to the data and thread affinity as well as the runtime-specific behavior.

Second, extensions for *standard-compliant tool support for future versions of the OpenMP standard* have been developed and proposed to the OpenMP Language Committee. These extensions with focus on OpenMP 4.x features aim to complete the current standard-compliant tools proposal in order to cover the complete range of current OpenMP constructs. Parts of the proposal are reflected in the technical report 4 which is the first basis for a standard-compliant OpenMP tools interface and aims to be part of the specification in OpenMP 5.0.

Third, the first formal *epoch model for OpenMP* based on the *OpenMP Tools* (OMPT) interface has been defined. This model covers the complete range of OpenMP features and enables the determination of happens-before relation with respect to the OpenMP runtime and memory model [14]. Since OMPT itself has a low overhead, it intends to be the foundation for scalable correctness checking tools.

Finally, *methods and algorithms* have been developed, which enable the *detection of new classes of errors*. These methods are based on the definition of the developed OpenMP epoch model, where a memory access tracer applies the theoretical approaches as part of the real-world correctness checking tool MUST [43]. In order to ensure the applicability of the binary instrumentation-based memory access tracer, a specific compression algorithm is used in order to reduce the massive memory overhead.

This work as a whole shows how information based on the memory accesses of parallel applications can be used in order to ensure both: the correctness and the performance of the application. The sustainability of the developed methods is guaranteed by the fact that parts of this work have been proposed to the OpenMP specification, which is the de-facto standard for shared memory programming, and the general transferability of the concepts to other programming paradigms.

1.4. Thesis Structure

Chapter 2 covers the efficient memory access in OpenMP programs. First, a brief overview of the programming paradigm, the used and developed benchmarks and the system architectures is given. Second, a methodology for the systematic assessment of OpenMP for target devices is presented. Last, the task-parallel programming with OpenMP on NUMA architectures is analyzed in detail by applying different tasking patterns and memory distribution strategies.

In order to ensure tool portability for a certain programming paradigm, reliable methods for the acquiring of runtime information are required. Chapter 3 discusses the different approaches for standard-compliant interfaces with a focus on programming paradigms using target device offloading. A detailed overview of the design objects for OMPT – the interface for OpenMP – is given. Based on this, the extension for target devices is presented and evaluated.

Chapter 4 defines the formal and generic epoch model for the determination of happens-before relations, which are required for the validation of parallel programs. It shows how the synchronization information from OMPT can be used on a per-thread level, and how to merge this information for the determination of global epochs. The model is evaluated with a nested parallel application and extended for OpenMP tasking. Furthermore, the portability of the concept to other parallel programming paradigms is discussed.

Chapter 5 evaluates the methods developed in the previous chapter for the performance and correctness analysis. For the latter purpose an instrumentation-based memory access tracer is presented and evaluated. For the performance analysis it is shown that the designed OMPT extension works and can be used in the real-world performance measurement infrastructure *Scalable Performance Measurement Infrastructure for Parallel Codes* (Score-P) and the visualization tool Vampir.

Finally, Chapter 6 summarizes this thesis and provides an outlook on future research directions.

2. Efficient Memory Access in OpenMP

In modern supercomputers the performance of parallel programs highly depends on efficient memory accesses. For the de-facto standard for shared memory programming – OpenMP – this even becomes more significant with an increasing amount of cores and an increasing complexity of the parallel architectures in general. On the one hand new programming features which allow standard-compliant and convenient data and thread placement became part of the specification [33], on the other hand the application of these features still requires a deep knowledge of the programming paradigm and the parallel architecture in order to achieve a good performance. This chapter will show the relevance of this knowledge application. Furthermore, the performance examinations stress the importance of an adequate tool support to analyze the application performance on new micro-architectures. The chapter clarifies that one issue which slows down the development of a tool for such new architectures is the lack of a standard-compliant interface for the tool support.

The chapter is structured as follows. First it gives a brief overview of OpenMP and the features that are most relevant for this thesis (Section 2.1). It presents the benchmarks and system architectures which are used for the different kind of analysis (Section 2.2 and Section 2.3). Section 2.4 presents the systematic assessment of the paradigm for target devices. Furthermore, different patterns for OpenMP tasking are presented and analyzed on NUMA architectures in Section 2.5. Finally, conclusions are drawn in Section 2.6.

2.1. Overview of OpenMP

OpenMP [71] is a directive-based *Application Programming Interface* (API) to express parallelism in C, C++ and Fortran programs. Beside a set of compiler directives it provides a collection of library routines and environment variables to control the parallelism. The specification published by the OpenMP *Architecture Review Board* (ARB) – a consortium of research institutions, soft- and hardware vendors – is portable across architectures. The programming paradigm is user-directed, which means the actions taken by a compiler or runtime system for the parallel execution of a program are explicitly specified by the programmer. OpenMP is the de-facto standard for parallel shared memory programming in the area of HPC. However, since version 4.0 it also support the programming for heterogeneous architectures

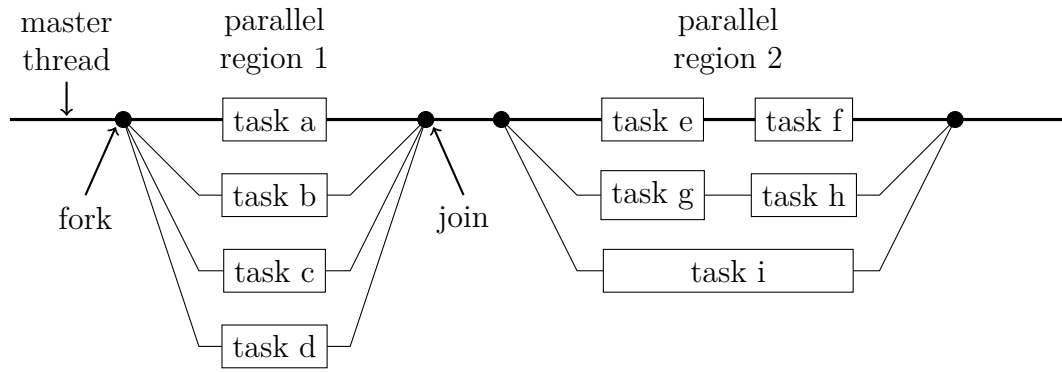


Figure 2.1.: OpenMP fork-join model.

like accelerators or coprocessor which do not share the same physical or virtual address space of an application. Since this complete thesis has a focus on OpenMP, this section gives a brief overview of the execution and memory model, and summarizes the main features. This overview does not cover the complete specification, but only those parts required for the comprehension of the following chapters.

Execution Model The execution model used in OpenMP is a fork-join model, first formulated by Conway [19] in 1963. Figure 2.1 depicts the model. Each OpenMP program starts the sequential execution with the *master thread*. After a thread fork, multiple threads execute tasks, which have been defined implicitly or explicitly by the programmer. A typical example for the use of implicit tasks are *worksharing constructs*, where for instance the iterations of a loop are distributed across the threads. The *parallel region 1* in the figure is an example for that, where each thread executes exactly one task with a subset of the iterations. In contrast, *parallel region 2* is an example for the execution of explicit tasks (e.g., created by the `task` construct), where each thread executes a different amount of tasks of different length. The additional threads spawned by the master thread are called *worker threads*. Both – master and worker threads together – form a *team* of threads and participate in the parallel region. After each parallel region the threads are joined and only the master thread continues the (sequential) execution. Here, it is up to the runtime implementation whether the worker threads are destroyed or put into an idle state and managed in a corresponding thread pool. The join at the end of a parallel region is an example for an *implicit barrier*. Besides that, threads can be synchronized by an *explicit barrier* within a parallel region, if the programmer uses the corresponding construct.

OpenMP parallel regions may be nested arbitrarily, as depicted in Figure 2.2. With each new parallel region a new thread team is created, in which the thread encountering the `parallel` construct becomes the master. Thus, an OpenMP program can have multiple master threads (one for each team). A master thread has always the thread ID 0, which means that thread IDs are only unique within the

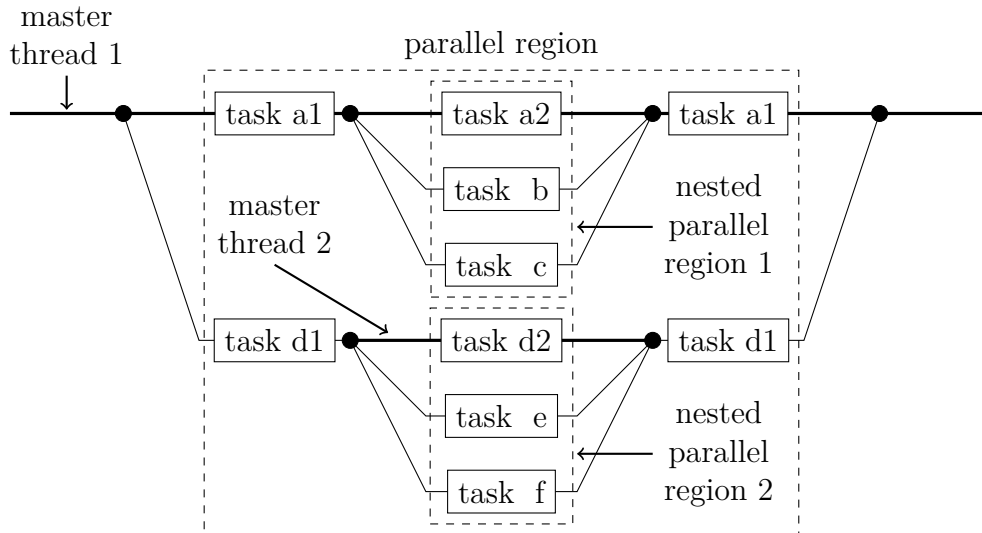


Figure 2.2.: OpenMP nested parallelism. A $\bullet$ marks a fork or a join.

same thread team. After the complete execution of the tasks of the inner nested parallel region, the thread switches back to the task which created the new team.

Depending on the architecture, the thread affinity is essential for the performance of an application. In order to control this, OpenMP allows to bind the threads to the processor. Once a thread is assigned to a place, it should not be moved to another place by the runtime system. The places to which a thread can be assigned are managed in an ordered list, referred to as *place partition*. This list describes the places which are currently available. The thread affinity can be specified as *master*, *close* or *spread*. In the *master* policy, all threads forked by a master thread will be assigned to the same place as the master thread. Based on the place partition, the *close* thread affinity policy instructs the runtime system to assign the threads as close as possible to the master thread. In contrast, the *spread* policy distributes the threads among the place partition in a way that the distance is as big as possible. For a certain multi-socket architecture the OpenMP affinity can be used to bind all threads of a team either to the same socket or to distribute them among all sockets.

Target Device Offloading The execution of an OpenMP program always starts on the *host device*. However, since version 4.0 the specification also supports *target devices*, which can be used to offload compute-intensive code parts. A target device may be an accelerator such as a *General Purpose Graphics Processing Unit* (GPGPU), a coprocessor like a Xeon Phi or just a logical execution engine running on the same physical processor. A target device may or may not share the same address space, but always has its own threads, which cannot migrate between the devices. Furthermore, the used instruction set of a target device may differ from the instruction set of the host device. In order to guarantee data consistency between

2. Efficient Memory Access in OpenMP

the host and the target device, the used data has to be mapped explicitly (with exception of scalar variables, which are mapped implicitly). For target devices with a distributed memory this means that the data has to be transferred (i.e., copied) to or from the host device. The mapping can be done by using the corresponding clause on a `target data` construct, or directly with the `target` construct, which offloads the following code region to the target device.

Memory Model OpenMP versions prior 2.5 lack a formal definition of a consistent memory model, which led to misunderstandings about how memory behaves in OpenMP programs [44]. In order to overcome this issue, later versions include a memory model based on [14, 15] in the specification. In this context, it is important to know that OpenMP neither guarantees coherence behavior of an application nor to be data race free. Both is up to underlying system or rather the programmer. However, OpenMP guarantees a certain consistency, based on the *flush* operation. This consistency is defined as *relaxed-consistency*, which means that each thread may have a temporary view of memory, because it is not required to have a consistent memory at every point in time. This relaxed-consistency gives a runtime implementation the freedom to keep data values in local structures such as machine registers, caches or other local storages in order to hide memory latencies. The flush operation to a set of variables enforces the consistency between the temporary view and the memory and discards the temporary view.

Many OpenMP directives support data sharing clauses. Here, the basic distinction is done between *shared variables* and *private variables*. A shared variable within the corresponding code region always refers to the original variable, whereas a private variable refers to a variable of same type and name, but is private to the thread (e.g., has a different memory address). The data can not directly be moved between temporary views, without going through memory. Thus, if a thread accesses the data of a shared variable, a flush operation has to be executed. The access to a private variable (or its reference pointer) of another thread is not allowed and results in undefined behavior. The same holds for the access to *threadprivate* data that is defined globally. The flush of the temporary view of all visible variables is issued implicitly in a barrier region, at the begin and the end of a parallel, critical or ordered region (including combined worksharing constructs), and during lock API routines in order to guarantee consistency.

2.2. System Architectures

For experiments and evaluations in this thesis different systems and architectures have been used. The systems have been chosen for the following reasons: The 4-socket bullx s6010 is a representative for a multi-core architecture and a commodity *Symmetric Multiprocessor* (SMP) system. The *Bull Coherence Switch* (BCS) sys-

tem is a hierarchical NUMA architecture with a huge core count. The Xeon Phi coprocessor represents a new many-core micro-architecture. Although the systems are not equipped with the latest CPUs and hardware components, the results are still transferable to others systems of the same class, because they are representative and all investigations are not specific to the used hardware components.

4-sockets bullx s6010: The bullx s6010 computed node is equipped with four Intel Xeon 7550 (Nehalem-EX) processors with 8 cores each. Thus, the system has 32 physical cores and 64 logical cores with respect to the *Simultaneous Multi-Threading* (SMT) capabilities running at 2.0 GHz. All four processors are connected with the Intel *QuickPath Interconnect* (QPI), which creates a system topology with four NUMA domains. The main memory for the complete compute node is about 64 GB. The system is running CENTOS Linux 7.3.

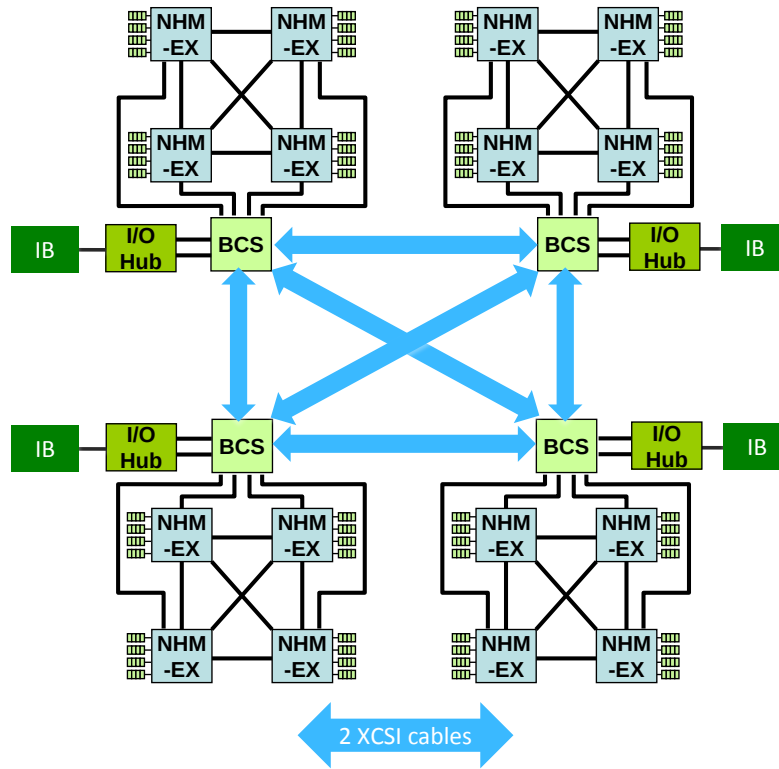


Figure 2.3.: Architecture overview of the BCS system, based on [38, 23].

16-sockets BCS System: The BCS system combines four bullx s6010 boards to a 16-socket 128-cores machine. As depicted in Figure 2.3 each board is equipped with four Intel X7550 (Nehalem-EX) processors with 8 cores each. The on-socket network uses the point-to-point QPI from Intel, while the boards are connected via *eXtended QPI* (xQPI) using the proprietary chip from Bull. Due to this two level design the reachable latency and memory bandwidth differs between cores accessing

2. Efficient Memory Access in OpenMP

memory within the same socket, across sockets on the same board and across boards. Thus, the architecture is a hierarchical NUMA system. The peak performance of the complete system is about 1 TFLOPS. The amount of main memory ranges between 256 GB and 2 TB. Since the complete system is coupled cache-coherently, it is running a single system image of CENTOS Linux 7.3.

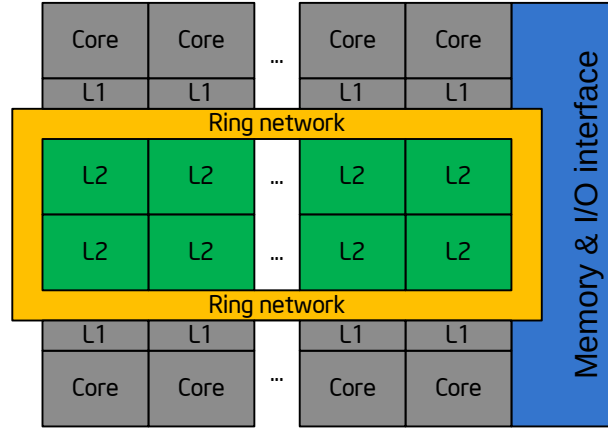


Figure 2.4.: Architecture overview of the Intel Xeon Phi coprocessor, based on [40, 23].

Intel Xeon Phi Coprocessor: In 2011 Intel announced the first product based on the *Many Integrated Core* (MIC) architecture as a coprocessor. The coprocessor is a device with a x86-based architecture attached to a host system via *Peripheral Component Interconnect Express* (PCIe) bus. A coprocessor can be used to offload compute-intensive parts of the code to the device in order to achieve a better application performance. Since the coprocessor is a many-core architecture, only the scalable code parts will benefit. The sequential parts (e.g., file IO) still have to be done on the host system. Due to the PCIe bus, the bandwidth between the host and the coprocessor is limited to 6 GB/s, which is slow compared the memory bandwidth within the host or the coprocessor. Thus, algorithms which require too much communication between host and coprocessor might not benefit as well. Depending on the concrete model, the amount of cores differs between 57 and 61. The experiments shown in this thesis have been done on a Xeon Phi 5110P coprocessor with 60 cores clocked at 1053 MHz. The coprocessor is hosted in a two-socket Intel *Sandy-Bridge* (SNB) based system with two Intel Xeon E5-2650 CPUs. Furthermore, the coprocessor offers full cache coherency across all cores. Due to the four-way SMT capabilities and a *Single Instruction Multiple Data* (SIMD) vector length of 512 bit, one register can store up to eight floating point numbers in double precision or respectively sixteen in single precision. Furthermore, the vector unit supports *Fused Multiply-Add* (FMA) instructions and thus can execute up to sixteen double precision operations per cycle. Consequently, the coprocessor has a double precision peak performance of about 1 TFLOP. However, this also means that an application

which is not SIMD vectorizable might not benefit from the compute capabilities of the coprocessor. Compared to host processor system the available main memory is (depending on the model) limited to 8 to 16 GB (GDDR5). Figure 2.4 depicts a high-level overview of the architecture. All cores are connected in a ring network and have a private level 1 and level 2 cache. The system is running the Intel *Manycore Platform Software Stack* (MPSS) 3.7.

2.3. Benchmarks

This thesis will use a selection of benchmarks to characterize and analyze the performance of a shared memory system. All measurements have been performed (at least) 10 times and the minimum time has been taken, unless otherwise stated. The minimum time (or rather the maximum performance) is used because it represents the time a system can achieve without any software artifacts included. However, if the deviation is high, this will be discussed as well. The used selection of benchmarks does not claim completeness, but is chosen in order to depict different effects as illustrative as possible and be applicable on real-world (OpenMP-related) problems at the same time. The effects which will be shown include the influence of the memory bandwidth to a given application, effects caused by paradigm-specific overheads, thread and data placement effects on NUMA architectures, and runtime specific behavior of different tasking patterns.

STREAM Although the number of *Floating Point Operations per Second* (FLOPS) is often considered to define the maximum achievable performance of a modern supercomputer, the available memory bandwidth is one of the most limiting factors for a good performance for many important applications (e.g., for all benchmarks in the SPEC OMP2001 suite [4] or for parts of the quantum chemistry and solid state physics software package CP2K [8]¹). In order to measure the sustainable memory bandwidth of a given system, McCalpin developed the STREAM benchmark [58], which uses simple vector kernels for the determination. In this thesis it is used in order to elaborate a performance model for the assessment of different hardware and software aspects. In order to keep the analysis uniform, only the bandwidth of the triad operation (scaled vector-vector addition) is taken into account.

EPCC Microbenchmarks As mentioned above, OpenMP uses directives to express the parallel behavior of an application. It is up to the implementation if these directives are directly translated into the machine-specific assembly code, an intermediate representation or replaced by a call into the corresponding runtime. In each case the use of the OpenMP constructs might produce an additional overhead which limits the performance or scalability of a certain application. In order to quantify the overhead of the key constructs, the EPCC Microbenchmarks [17, 18] have been

¹<https://www.cp2k.org>

2. Efficient Memory Access in OpenMP

```

1  start = getclock();
2  #pragma omp parallel
3  {
4      for(int j=0; j<innerreps; j++){
5          #pragma omp for
6          for(int i=0; i<omp_get_num_threads(); i++){
7              delay(delaylength);
8          }
9      }
10 }
11 time = (getclock() - start);

```

Listing 2.1: EPCC kernel to determine the parallel time of a **for** construct, based on [17].

```

1  start = getclock();
2  for(int j=0; j<innerreps; j++){
3      delay(delaylength);
4  }
5  time = (getclock() - start);

```

Listing 2.2: EPCC kernel to determine the reference time of a **for** construct, based on [17].

developed by Bull et al. As methodology the overhead o_x of an OpenMP construct x is defined as

$$o_x = \frac{t_p - \frac{t_s}{p}}{r_i}, \quad (2.1)$$

where t_p is the execution time of the program on p processors, t_s the sequential execution time and r_i is the number of executed constructs (referred to as *inner repetitions*). Thus, o_x is the difference between the measured parallel execution time and the time given a perfect scaling. In order to measure the parallel and sequential time Bull and O'Neill. define the number of executions r_i for a given directive as **innerreps**. For each encountered directive a routine **delay()** which contains a dummy loop of length **delaylength** is executed. Listing 2.1 shows how $t_p \cdot r_i$ is determined (i.e., the time for the execution of r_i constructs) on the example of a **for** construct. The innermost loop ensures that every thread executes the same dummy loops and thus has the same execution time than the others. The corresponding sequential (reference) time $t_s \cdot r_i$ is measured by executing the code in Listing 2.2. In both cases the value of **delaylength** is chosen that the time taken by the routine and the overhead of the directive is in same order of magnitude. The EPCC Microbenchmark suite includes corresponding benchmarks as given in Listing 2.1 and 2.2 for most of the constructs in OpenMP. However, since the benchmark was published before OpenMP added support for tasking or heterogeneous computing, the original publication does not include any of the latest features of the standard (i.e., **task** or **target** constructs). In this thesis the EPCC Microbenchmarks are used for the evaluation of new hardware or new OpenMP features. For the latter one the technique used for the overhead determination is applied to a given new directive by extending the benchmark.

Conjugate Gradient Method As a representative real-world compute kernel, I developed a benchmark which implements a CG method [42], which approximates the solution of a sparse linear equation system. The method belongs to the class of Krylov subspace methods. These methods are iterative algorithms, which improve an already given solution with every iteration step. Algorithm 1 shows the used (unpreconditioned) CG method which approximates an equations system $A * x = b$. Here, let be $N \in \mathbb{N}$, $A \in \mathbb{R}^{N \times N}$, $b, x, q, p, r \in \mathbb{R}^N$, and $\delta, \rho, \alpha, \beta, \epsilon \in \mathbb{R}$. $\langle p, q \rangle$ denotes the dot product of p and q , and $\|r\|_2$ the Euclidean norm of r . As a starting point the algorithm guesses the first solution for the CG method randomly (line 1). The current approximation of the solution x_k is updated in line 13. Not taking rounding errors of floating point arithmetic in modern supercomputers into account the exact solution of the equation system is computed after latest N iterations. The accuracy can be determined by computing the corresponding residuum (line 16). If this convergence criteria is fulfilled the algorithm stops, because the approximation of the solution for x is accurate enough. The CG method does only work, if the matrix A is symmetric positive-definite. However, since the method is used as a benchmark for performance evaluations and the kernel operations are the same for other sparse algorithms, this does not limit the applicability and transferability of the evaluations performed in this thesis. The relevance for HPC applications is given, because in many simulation codes a CG solver is also one of the most compute-intensive parts. For instance the FEM simulation package iMOOSE presented in this thesis (refer to Section 1.2) uses a CG method for the solving of the equation system. The relevance can also be seen by the fact that Dongarra et al. proposed a high performance CG benchmark for ranking high-performance computing systems [28] as a new metric. They stress that such a benchmark strives a better correlation to existing codes than the Linpack benchmark [29], which is used to evaluate the TOP500² fastest supercomputers in the world. Especially, the most compute-intensive part of the CG solver – the *sparse Matrix-Vector* (spMV) product (line 10) – is also very relevant for many other iterative *Partial Differential Equation* (PDE) solvers.

In order to reduce the memory overhead, typically only the non-zero values of the matrix are stored. Here, different formats became state of the art. For architectures with a hierarchical cache/memory structure, the *Compressed Row Storage* (CRS) format has evolved as one of the standard formats. For this reason this format was chosen for the developed benchmark. The CRS format stores only the non-zero values and the column index in an array of length nnz . In order to determine the end of a row, another array for the row offset is required. This row offset array is of length n , where n is the dimension of the (square) matrix. Since all matrix entries are stored consecutively in memory, the spatial locality of the hardware cache (as available in homogeneous commodity systems) can be fully exploited by the spMV kernel – at least with regard to the matrix access – without limiting the potential for a parallel execution by using a corresponding domain decomposition. With respect

²<http://top500.org>

Algorithm 1 Pseudo-code of the used CG method.

```

1: Choose  $x_0$ 
2:  $r_0 \leftarrow b - A * x_0$ 
3:  $p_0 \leftarrow \vec{0}$ 
4:  $\rho_0 \leftarrow 0$ 
5: for  $k \in \{1, \dots, N\}$  do
6:    $\rho_k \leftarrow \langle r_{k-1}, r_{k-1} \rangle$ 
7:    $\beta \leftarrow \frac{\rho_k}{\rho_{k-1}}$ 
8:    $p_k \leftarrow r_{k-1} + \beta * p_{k-1}$ 
9:    $\triangleright$  spMV: most compute-intensive
10:   $q_k \leftarrow A * p_k$ 
11:   $\delta \leftarrow \langle p_k, q_k \rangle$ 
12:   $\alpha \leftarrow \frac{\rho_k}{\delta}$ 
13:   $x_k \leftarrow x_{k-1} + \alpha * p_k$ 
14:   $r_k \leftarrow r_{k-1} - \alpha * q_k$ 
15:   $\triangleright$  Check convergence
16:  if  $\|r_k\|_2 < \epsilon$  then break

```

to the right-hand side vector, the access pattern might not be optimal (depending on the sparsity pattern of the matrix) regarding the cache behavior, because of the indexed access. Furthermore, the unmodified CRS format might also not perform best on modern heterogeneous architectures. A detailed analysis of the different storage formats is out of scope of this work and can be found in [7], for instance.

Since the values in the matrix are only read once for each spMV product and given the fact that the required two floating point operations can be executed much faster than the required load/store operations in modern architectures, this benchmark is a good example for the big class of memory-bound applications (refer to Section 2.4 for a more detailed performance model).

Beside the characterization of the memory performance of a system, the benchmark can be used to evaluate the performance of a certain runtime system or a programming approach. This includes for instance different aspects of the operating system (e.g., memory policy, scheduling strategy) or the quality of a certain OpenACC or OpenMP runtime implementation. For that, I use different programming methods to solve the linear equation system and evaluate the performance. Depending on the underlying problem, the matrix for the system of linear equations can be very irregular. In this case the strategy for an adequate load balancing is an important aspect. In order to evaluate this, the benchmarks include configurable OpenMP scheduling strategies, different tasking patterns and a version which precalculates a good data distribution explicitly. The latter one can be used as an upper performance limit for a certain system. This helps to evaluate either the programming approach or the quality of the runtime implementation for the first

ones, because the performance to a solution close to the optimal is known. The following nomenclature is used in order to refer to the different (selected) versions of the benchmarked:

- ***cg_omp***: OpenMP-based version using a worksharing construct with a configurable OpenMP schedule (e.g., static or dynamic), which distributes the matrix row-wise. The parallelization here is done on algebraic operation kernel level, where each operation within one iteration using the worksharing constructs for the matrix-vector, vector-vector and the dot product.
- ***cg_omp_dd***: OpenMP-based version using an explicit data distribution of the matrix, based on the number of non-zero. Depending on the sparsity pattern of the matrix an adequate load balancing is required for the spMV operation in order to get a good performance. Instead of using a dynamic schedule for the row-wise partitioning of the matrix, this version pre-calculates the number of rows for each thread in a way that the amount of non-zero values are distributed as equal as possible. This avoids the overhead for the dynamic OpenMP schedule and allow NUMA-aware memory/thread placement. Since the matrix does not change during the solving process and the pre-calculation is only done once, the overhead to compute the data distribution explicitly is negligible. All other kernels required for the algorithm using the same worksharing constructs as in version *cg_omp*.
- ***cg_mkl***: A version using the Intel *Math Kernel Library* (MKL) [46] for all kernel operations of the CG algorithm. The MKL uses OpenMP worksharing constructs internally.
- ***cg_task_sp***: OpenMP-based version using OpenMP tasks with a single-producer, multiple-executor pattern. Details to this version can be found in Section 2.5.
- ***cg_task_pp***: OpenMP-based version using OpenMP tasks with a parallel-producer, multiple-executor pattern. Details to this version can be found in Section 2.5.
- ***cg_openacc***: OpenACC-based version using OpenACC kernels on operation level.
- ***cg_cuda***: Version based on the *Compute Unified Device Architecture* (CUDA) paradigm for NVIDIA GPGPUs.

For all versions based on OpenMP the memory policy is configurable. The intention of this benchmark implementation is neither to implement a complete linear equation solver library nor to discuss new numerical or programming methods or models. For the following sections it is only used as a representative example for a real-world compute kernel to illustrate and discuss different performance aspects.

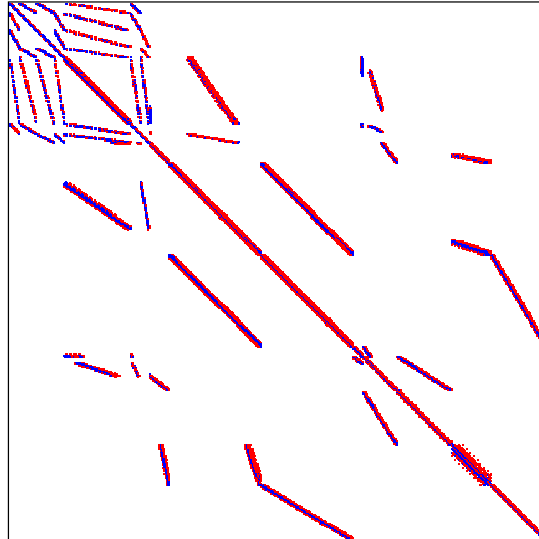


Figure 2.5.: Sparsity pattern of the used matrix.

The performance of the CG method might not only depend on the used version, the memory/thread affinity or the used hardware, but also on the properties of the matrix. One important factor is the size of the matrix, because the performance will behave completely different if the equation system fits into the caches of the system or requires the complete hierarchical memory subsystem. Even more important for the performance is the sparsity pattern of the matrix. For instance, this pattern determines if an adequate load balancing with an OpenMP worksharing construct with static schedule is possible. Furthermore, the indirect access to the index vector of a spMV operation is directly influenced. If multiple consecutive matrix entries are non-zero elements, the spatial cache locality is also for this vector beneficial. If the non-zero matrix elements have a regular pattern, hardware prefetcher can help to hide the latency of a given system. However, since in real-world problems (especially in engineering research and development) this is often not the case, a more representative data set was chosen for all evaluations made in this thesis. The matrix represents a computational fluid dynamics problem (Fluorem/HV15R) and is taken from the University of Florida Sparse Matrix Collection [25]. Although the matrix is not positive-definite (and thus does not fulfill the convergence criteria for a CG method), the performance results can be used for other matrices, because reaching convergence or not is not important from the computational perspective. Instead of converging the problem all measurements do 1000 CG iterations. The matrix was chosen for the following reasons. The dimension of the square matrix is $n = 2,017,169$ and the number of non-zero elements is $nnz = 283,073,458$, which results in a memory footprint of approximately 3.2 GB. Hence, the data set is big enough not to fit into the caches of used systems, even on the 16-sockets machine (refer to Section 2.2). Furthermore, as can be seen in Figure 2.5, the sparsity pattern is not regular and thus the CG method can be used for representative performance analyses of unbalanced algorithms.

2.4. Systematic Assessment of OpenMP for Target Devices

Driven by the demand for more and more compute capabilities, an increasing dissemination of accelerator-based systems could be observed during the last decade. While at the beginning of this development often a laborious rewrite of the applications was required, approaches came to existence which promise to minimize the effort. The intention is to replace the time-consuming kernel rewrite for accelerators using specialized programming paradigms like CUDA or *Open Computing Language* (OpenCL) with the use standard programming models like OpenMP, *Portable Operating System Interface* (POSIX) threads or MPI. However, having a user-friendly method which enables a programmer to port an application to an accelerator does not necessarily mean that the applications performs as expected. Here, the performance might not only be limited by the available hardware, but also by the expressiveness of the used programming model or its certain implementation.

Furthermore, the efficient performance analysis for new micro-architectures or new features of the programming paradigm might not be possible due to the absence of an adequate tool support. A systematic methodology as presented in the following is an essential step to overcome this limitation. However, it does not replace the necessity of an appropriate tool support, especially for new features. For this reason, Chapter 3 proposes a portable and vendor-independent solution for this issue.

Based on [23] and [81] this section presents various aspects to assess the performance of accelerator-based systems systematically. The steps for this systematic methodology are the following:

1. Determination of basic performance characteristics.
2. Assessment of programming paradigm-specific overheads.
3. Scalability determination.
4. Model-driven performance prediction.
5. Evaluation with standard benchmark suites.

The evaluation of these aspects is done with a focus on the OpenMP target constructs and the Intel Xeon Phi coprocessor as representative platform applying this paradigm feature. Since in [23] a preproduction system of the Xeon Phi system was used and newer compiler versions are available, all measurements have been done with the latest available hardware and software components. Although, the methodology is evaluated with OpenMP and a Xeon Phi Coprocessor in the following, the approach is robust with other programming paradigms and other hardware platforms. The basic performance characteristics of the hard- and software infrastructure of the given accelerator are determined by applying the STREAM and the

2. Efficient Memory Access in OpenMP

LINPACK benchmark. For both benchmark corresponding benchmarks for other programming paradigms are available. The goal of second step is to measure the paradigm-specific overheads for a given runtime implementation. This includes the performance behavior of all API calls and directives. In this thesis this done with the EPCC benchmark, including an extension for target constructs. The scalability analysis of a given application has to be done with one or more representative benchmarks. Here it is determined with the CG benchmark as a real-world example. In order to assess and predict the potential of new hardware with such a representative benchmark the fourth step as to be applied. Here, the idea is to create a corresponding performance model for the given application. In this this thesis, the Roofline [92] performance model is applied to the most compute-intensive part of the CG kernel for the model-driven performance prediction. As a last step in the methodology, a standard benchmark suite is used and the performance results are compared with the 2-socket host system. The goal here is to evaluate the system including all soft- and hardware components as a whole for a broad range of representative real-wold applications.

By applying all steps of this methodology to a large SMP system and an accelerator-based system, the comparison of the results on both systems allows to assess their actual performance and show their limitations. In this section the BCS system is used as a representative for big shared memory system with a hierarchical NUMA architecture and the Intel Xeon Phi coprocessor as a representative for accelerator-based systems. It is important to understand that these two systems – although they are both x86 and have a similar peak performance of about 1 TFLOP – are completely different in many points. The BCS system couples four 4-socket systems to a huge 16-socket machine which takes six height-units in a server, while the Xeon Phi Coprocessor is a relatively small extension card plugged into the PCIe bus of a host system. The BCS system is available with up to 2 TB of main memory, while the Xeon Phi is limited to 16 GB. The intention of the comparison is not make a statement about which hardware is better or worse for a certain use case. It is rather made to get an impression that although the systems are different, the evaluation methods are the same and thus applicable to other systems. Furthermore, it will show that an established system can be used to bring the performance results of a quite new system into relation. From the perspective of the runtime performance analysis, the fact that both systems have the same amount of cores is beneficial for the assessment of the potential scalability for OpenMP constructs on the target device. As programming model for the Xeon Phi Coprocessor different paradigms can be used. However, this work focuses on only two of them:

- **Native execution on the device:** Since each coprocessor runs a specialized Linux kernel a direct login to the device is possible. In order to execute a program on the device it needs to be cross-compiled on the host system first by using the corresponding compiler switch. Furthermore, a cross-compiled version of the OpenMP runtime is available, so that any OpenMP program can be executed natively on the device.

- **OpenMP target constructs:** Instead of executing the complete program on the device, selected code regions can be offloaded to the device by using the OpenMP target constructs (refer to Section 2.1). This has the advantage that parts of the code with a poor performance on the device (e.g., file I/O, serial code) can be executed on the host. Furthermore, only parts of the work can be offloaded, so that the compute capabilities of the host can be used as well to solve a given problem.

All measurements in this section have been done with the Intel 16.0 compiler. The MPSS version is 3.7.2.

1. Determination of basic performance characteristics One of the most basic performance characteristic for every supercomputer is the peak performance. As mentioned before, the Linpack benchmark evaluates how close a system comes to this theoretical peak by solving huge dense linear equation systems. For new micro architectures like the Xeon Phi coprocessor the benchmark has been optimized correspondingly. Heinecke et al. [41] did this for a single and multi-node systems based on Xeon Phi coprocessors and reached an efficiency of 79.8% on a single Xeon Phi and 76.1% on a 100-node cluster with a hybrid Linpack implementation.

Beside the theoretical amount of arithmetic operations (typically measured in FLOPS) the reachable memory bandwidth is one of the most important metrics for the evaluation of a given hardware. Especially in many sparse linear algebra kernels the bandwidth is the limiting performance factor. Thus, the knowledge of the bandwidth enables a reliable prediction of the performance of an application (e.g., a CG solver as used in this thesis).

Figure 2.6 shows the memory bandwidth on the two systems used for the evaluation in this section. The used memory footprint is always about 2 GB. In order to prevent that the operating or runtime system migrates threads among different cores or NUMA domains during the execution of the benchmark, all threads have been bound to the cores with different policies (refer to Section 2.1). Since only the BCS system is a NUMA system, a performance difference can be only obtained for this architecture for these different policies. In contrast, the Xeon Phi has a uniform memory access from all cores and thus no policy is specified. Furthermore, the data is page aligned and initialized in parallel in order to get a balanced data distribution. The memory bandwidth on the Xeon Phi coprocessor has been determined by native execution on the device. In order to achieve a good performance, specific compiler options for an adequate software prefetching were set.

The results show that the memory bandwidth for a close binding strategy increases slower for smaller amounts of threads. This is because the binding first fills a whole socket before the next socket is used and thus not all available memory controllers are used. The figure clearly shows that the scalability across the NUMA domains is

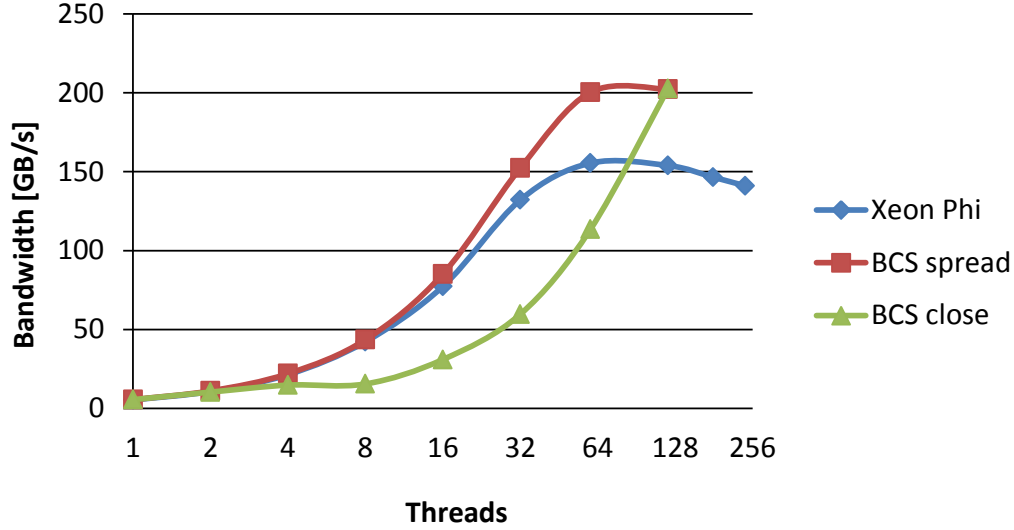


Figure 2.6.: STREAM memory bandwidth of the BCS system and the Intel Xeon Phi Coprocessor for different amount of threads and two binding strategies on the BCS.

given, because the bandwidth doubles with each doubled amount of threads as soon as one socket is filled (16 threads). On the other hand, the spread binding strategy shows that the memory bandwidth does not increase for more than 64 threads anymore. This shows that the memory controllers are already saturated with by using half of the cores on each multi-core processor of the system. However, the maximum bandwidth of about 200 GB/s is good and the same for both strategies (as expected). The maximum bandwidth achieved on the Xeon Phi coprocessor is about 155 GB/s. Given the fact that the coprocessor is a rather small extension card plugged into the PCIe bus and the BCS system is a big SMP system requiring 6 height-units in a rack, this is a rather good result as well. Furthermore, the figure shows that scalability of the memory bandwidth is given for all 60 cores. However, since an application executed on the Xeon Phi requires to utilize at least 2 threads to saturate the decodes stage of the pipeline, this also limits the performance of memory-bound applications.

In total the BCS system achieves an about 40% higher memory bandwidth than the coprocessor. However, one have to keep in mind that it uses 16 processors and 16 memory controllers for that. The coprocessor achieves a better bandwidth than 8 of the Xeon X7550 processors on a single chip, which can be seen on the close strategy on the BCS system. The memory available on the latter one is much larger than on the Intel Xeon Phi coprocessor, and for larger data sets the comparison needs to take the transfer time through the PCIe bus into account.

2. Assessment of programming paradigm-specific overheads Beside the determination of the basic performance characteristics of a new hardware, the evaluation

of the programming paradigm-specific overheads is a basic requirement for a good performance of a given application on a new hardware. Thus, the choice the benchmark highly depends on the used programming paradigm. In case of OpenMP, the EPCC Microbenchmarks can be used to measure the overheads of the OpenMP constructs. A low overhead and also the scalability of these constructs are an important prerequisite for the performance of an OpenMP application. This paragraph shows how the benchmark can be used to assess the scaling potential of new micro-architectures. Here, the Intel Xeon Coprocessor described above is used as an example for a new innovative accelerator technology. In order to assess if the measured overheads for the OpenMP constructs are reasonable, an established system is required for the comparison. In this case this is done with the BCS system, because it is a huge shared memory system, where a comparable number of threads can be used. Slow benchmark results do not necessarily mean that a new (accelerator) architecture is not usable for OpenMP programs. A target device might run a separated instance of the operating and the OpenMP runtime system. Thus, it might also mean that the used system or any other software component is not optimized for the target device. However, in this case the same version of the Intel OpenMP runtime is used for a fair comparison. Since the EPCC Microbenchmark does not include new constructs, a small extension will be presented in order to assess the `target` construct.

Since the given hardware belongs to the class of many-core architectures, the focus in this paragraph is on the analysis of those OpenMP constructs which require synchronization. Table 2.1 shows the overheads for the `syncbench` for different amounts of threads. Depending on the architecture the maximum amount is 128 threads (the BCS system has 128 cores and no SMT activated) or 240 threads (the used Xeon Phi coprocessor has 60 cores with 4-way SMT capabilities). The part on the top shows the results achieved on the BCS system and the part in the middle those measured while executing natively on the coprocessor. The results in the bottom are also from the coprocessor, but in a slightly modified version where the constructs are executed within an OpenMP target directive from the host system³.

As expected, the overheads are increasing with the amount of threads for all measurements, because more threads need to be synchronized. It can be seen that the serial overheads on Xeon Phi coprocessor is higher compared to the BCS system for all constructs. This is reasonable, because the frequency as well the general capabilities of a single core in this system are higher. However, since the coprocessor was designed for highly-parallel applications this limitation is not relevant for the overall assessment of the architecture. More important is the fact that the comparison between the BCS system and the native execution on the coprocessor shows that all

³In the original publication [23] the measurements have been done with Intels *Language Extension for Offload* (LEO), because the corresponding OpenMP `target` constructs were not available at this early point in time. Although the results are similar, there are slightly differences, which will be analyzed in this section.

2. Efficient Memory Access in OpenMP

BCS System								
#Threads	PAR	FOR	PAR FOR	BAR	SING	CRIT	LOCK UNLO	RED
1	0.24	0.0	0.26	0.01	0.03	0.04	0.05	0.27
2	7.21	2.91	7.26	2.74	3.60	1.47	1.50	7.95
4	9.13	5.24	9.27	5.15	5.97	2.11	2.12	9.88
8	12.48	7.72	12.27	7.44	9.14	1.74	1.77	23.33
16	14.09	8.55	14.45	8.15	12.86	1.75	1.80	27.40
32	16.60	11.94	16.87	11.73	23.22	1.74	1.75	32.34
64	17.07	11.25	17.02	11.17	42.69	1.81	1.79	33.15
128	21.17	15.19	22.00	14.55	86.06	2.26	2.06	30.96
Intel Xeon Phi coprocessor (native)								
#Threads	PAR	FOR	PAR FOR	BAR	SING	CRIT	LOCK UNLO	RED
1	1.65	0.26	1.97	0.10	0.21	0.31	0.32	1.95
2	7.65	2.16	8.14	1.84	2.20	1.14	1.11	8.97
4	8.58	3.47	8.93	3.17	3.41	1.67	1.68	11.75
8	11.88	4.55	12.53	4.27	4.60	1.69	1.71	14.75
16	12.67	6.54	13.04	6.18	8.37	1.73	1.76	23.49
30	16.17	7.85	16.87	7.52	11.48	1.85	1.84	30.95
60	20.42	9.52	20.25	9.24	19.61	3.30	3.02	38.25
120	22.09	12.28	22.70	10.43	20.42	6.89	6.70	50.56
240	25.31	14.96	26.16	12.80	23.06	7.18	7.28	44.27
Intel Xeon Phi coprocessor (offload)								
#Threads	PAR	FOR	PAR FOR	BAR	SING	CRIT	LOCK UNLO	RED
1	1.69	0.26	1.96	0.10	0.21	0.31	0.42	1.90
2	7.70	2.16	8.10	1.84	2.20	1.14	1.17	8.75
4	8.58	3.21	9.03	2.89	3.29	1.69	1.75	9.56
8	11.66	4.84	11.92	4.52	4.91	1.71	1.76	12.40
16	13.01	6.70	13.21	6.36	8.32	1.75	1.79	23.18
30	16.32	7.88	16.67	7.63	11.92	1.82	1.89	29.54
60	20.88	10.00	20.55	9.69	19.32	5.05	3.25	33.77
120	22.62	11.71	23.39	10.76	20.28	6.80	5.62	39.25
236	24.54	15.00	25.78	13.21	22.46	6.92	6.84	41.22
240	550.63	481.61	544.94	470.93	467.80	6.03	9.59	1067.91

Table 2.1.: Overhead in microseconds for the OpenMP constructs `parallel`, `for`, `parallel for`, `barrier`, `single`, `critical`, `lock/unlock` and `reduction`. The measurements were performed with the EPCC Microbenchmark `syncbench` on the BCS system and the Intel Xeon Phi Coprocessor.

overheads are in the same order of magnitude for a comparable amount of threads. Only the results for a **single** construct and for the **lock/unlock** measurements are about three times higher on the coprocessor. Since the overhead with about 7 microseconds for the execution with 240 threads is still smaller than the overheads for all other constructs executed with the same amount of threads, this will not limit the scalability of OpenMP applications. On the other hand the overhead for the execution of a **single** construct is a factor of three better compared to the BCS system.

Since the Xeon Phi coprocessor can be used for the native execution of highly-parallel application or as device for target offloading of the compute-intensive parts, it is also essential for the assessment of the architecture that the overheads for the OpenMP constructs are in the same order of magnitude when executed within a target region. The table clearly shows that this is the case for almost all amounts of threads. However, there is one exception. When 240 threads (i.e., one thread per SMT unit or rather four threads per core) are used, the overheads for a **parallel**, a **for**, a **parallel for**, a **barrier**, a **single** or a **reduction** construct/clause increase significantly compared to the native execution. At first this is not as expected, because the same OpenMP runtime system is used on the device. The reason for this behavior is that the offloading mechanism requires an additional service process (i.e., the coi daemon) for the communication between the host and the accelerator. Thus, the execution of the offloaded target region with 240 threads leads to an over-subscription of the system, which results in the extensive increase of the overheads. An execution with only 236 threads gives the operating system the opportunity to use the resources of one core for the service process exclusively and thus avoids the performance decrease of the OpenMP application. As a consequence, a user of the coprocessor should not use more than 236 threads when the device is used for target offloading. In conclusion, the results show that the OpenMP overhead will not limit the performance or scalability of applications executed on the Xeon Phi coprocessor.

```

1  start = getclock();
2  #pragma omp target
3  for(int j=0; j<innerreps; j++){
4      delay(delaylength);
5  }
6  time = (getclock() - start);

```

Listing 2.3: EPCC kernel to determine the reference time of a **target** construct, based on [23].

Since the EPCC benchmark is limited to OpenMP version 2.5, the overheads of newer OpenMP features of the standard cannot be determined. However, for the assessment of the efficiency of the kernel offloading approach it is important to determine the overhead of a **target** construct. In order to measure the overhead for this construct, the EPCC benchmark was extended correspondingly. Here, the same methodology as described in Section 2.3 was applied. Listing 2.3 shows the

2. Efficient Memory Access in OpenMP

```
1  start = getclock();
2  for(int j=0; j<innerreps; j++){
3      #pragma omp target
4      {
5          delay(delaylength);
6      }
7  }
8  time = (getclock() - start);
```

Listing 2.4: EPCC kernel to determine the parallel time of a `target` construct, based on [23].

determination of the reference time $t'_s \cdot r_i$ by measuring the time of a dummy loop in the `delay` function `innerreps` times within a single target region. The determination of the offloading time $t'_p \cdot r_i$ is done by measuring the same `delay` function in `innerreps` target regions (Listing 2.4). Thus, the overhead o_t for the `target` construct is determined by applying Equation (2.1) (with $p = 1$):

$$o_t = \frac{t'_p - t'_s}{r_i}. \quad (2.2)$$

The measurement on a Xeon Phi coprocessor shows an overhead for the `target` construct of $o_t = 37.38 \mu s$. Given the fact that the constructs requires synchronization with the target device, this overhead is quite low and in the same order of magnitude with the overheads for other constructs. For instance, the execution of a `parallel` construct with 240 threads on the device has an overhead of $o_p = 22.09 \mu s$. This comparison shows, that the overhead for a `target` construct will not prevent the efficient usage for the offloading or compute-intensive kernels. However, for the overall performance of the construct the amount of data transferred to or from the device is essential. This cannot be measured by using an EPCC-like benchmark, because this highly depends on the certain application. Thus, performance tools are required to analyze the performance in detail. Since the used communication protocol might be vendor-specific, a corresponding standard-compliant tool support would limit the effort a tool developer has to spend.

3. Scalability determination Especially for highly-parallel architectures the scalability of an application is an important key factor. The goal of this step in the methodology is to assess if a given applications can benefit from the parallel resources of the hardware. Furthermore, it can show the maximum amount of processes or thread that should be used for this application on this hardware. Figure 2.7 depicts the scalability for the presented CG solver for two different versions (`cg_omp_dd` and `cg_mkl`, refer to Section 2.3) and binding policies. Once again, the binding policy on the Xeon Phi coprocessor is not specified, because the architecture allows a uniform memory access. Thus, the binding policy does not influence the performance, as long as all threads are bound to the physical cores. In the figure it can be seen that in general the scalability is given for both systems. The hand-tuned version

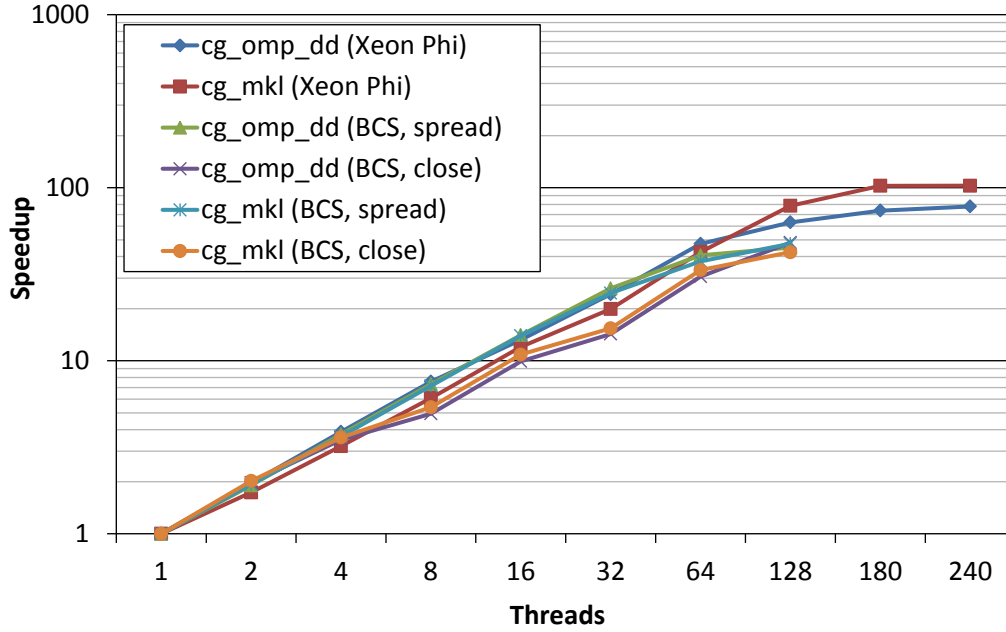


Figure 2.7.: Scalability of the CG benchmark (1000 iterations) on the Xeon Phi coprocessor and the 128-core BCS machine for different versions and binding policies.

cg_omp_dd reaches a maximum speedup of about 78 on the (60 cores) Xeon Phi coprocessor when the benchmark is executed with 240 threads. It can be seen that the use of the SMT capabilities is beneficial, because the speedup is only about 47 when only the 60 native cores are utilized. The scalability of *cg_mkl* is even higher with a maximum speedup of about 102 when executing the benchmark with 180 or 240 threads. However, this is only the case because of a very poor sequential performance of this version. The sequential execution time of the hand-tuned version is about 30 % lower than the sequential execution time of the MKL version. Furthermore, the parallel time is lower, although the speedup of *cg_mkl* is higher. This already shows the limited meaningfulness of the scalability, which strongly depends on the performance of the sequential baseline version. However, the comparison is important, because the scalability is a prerequisite for highly-parallel architectures. On the BCS system the maximum speedup is about 48 for *cg_omp_dd* (spread binding) and about 47 for *cg_mkl* (close binding). Given the fact that the memory bandwidth on this system is already saturated with 64 cores this is a reasonable result and shows that the CG kernel is a memory bound kernel. Furthermore, the figure shows the different behavior of the two binding strategies. By using the *close* strategy, first all cores on the same socket are used. Thus the memory controller on one socket becomes the limiting factor and one can see a lower increase of the speedup when using 8 instead of 4 threads, for instance.

In summary, the scalability analysis of a new (highly-parallel) micro-architecture

2. Efficient Memory Access in OpenMP

shows that one prerequisite for a beneficial use is given. However, the meaning of this analysis is very limited without taking other metrics like the total time to solution or the throughput into account. This especially holds, if the single core performance of an architecture is poor.

4. Model-driven performance analysis The consideration of the EPCC benchmark and the comparison with a given architecture shows how one can assess the potential of a new architecture for the efficient execution of OpenMP applications. The fact that the overheads for the OpenMP constructs is low does not necessarily mean that a given applications performs as expected, of course. Furthermore, also the scalability analysis does not enable a conclusion of the overall performance of a given application on a given hardware. In order to predict and assess the performance of an application on a given hardware reliably, one has to make a corresponding performance model which respects the limitations of the hardware. The Roofline Model [92] is one popular method which fulfills this purpose. The model gives an estimation of the maximum performance in GFLOPS with respect to the operational intensity oi of the algorithm by using (at least) two limits: the theoretical peak performance p and the maximum memory bandwidth bw . Here, the operational intensity defined as

$$oi = \frac{fops}{dops}, \quad (2.3)$$

where $fops$ is the amount of floating point operations for the kernel and $dops$ is the amount of data that need to be loaded or stored in order to execute these floating point operations. Then the performance limit L is defined as

$$L = \min(p, oi \cdot bw). \quad (2.4)$$

Thus, algorithms with a high operational intensity are bound by the peak performance p , while algorithms with a low operational intensity are bound by the memory bandwidth bw . Depending on the algorithm these two upper limits can be further decreased. On the computational side the algorithm might have a lower ceiling, if no SIMD vectorization is possible. In this case the amount of computed results per clock tick is lower than the given hardware can produce. Another computational ceiling factor is an unbalanced combination of different floating operations. For instance, if a given hardware has special support for FMA, but the algorithm does not use any multiplications at all. Examples for lower bandwidth ceilings are missing software prefetches or an algorithm which does not allow optimal memory placement in a NUMA system and thus cannot saturate the complete local memory controller. For an adequate performance model for a given algorithm it is important to analyze these ceilings, which is done on the example of the presented CG algorithm (refer to Section 2.3) as a representative real-world compute kernel in the following.

Since the determination of the operational intensity might be very extensive for complex algorithms, it is in terms of productivity a good approach to only model the

corresponding hotspots. In HPC these hotspots are often basic linear algebra kernels like matrix-matrix, matrix-vector or vector-vector operations. This also holds for the CG solver. Table 2.2 shows that the share of the spMV operation is about 84 % on the BCS system and about 94 % on the Xeon Phi Coprocessor. Thus, the Roofline Model is only applied to this kernel in the following. Since the spMV operation has a balanced relation of additions and multiplications and due the fact that the kernel is SIMD vectorizeable, for both used systems the theoretical double precision peak performance of about $p = p_{bcs} = p_{phi} = 1$ TFLOPS is chosen as upper computational limit. For the upper memory bandwidth limit bw the measured maximum STREAM bandwidth is chosen for two reasons. First, due to the hardware properties on the both system architectures, the theoretical peak memory bandwidth is not achievable at all and thus the STREAM benchmark becomes firmly established to determine the maximum. Second, no limits with respect to the data placement (affinity) are given, which is of certain importance regarding the BCS system. By using the paged aligned memory, the operating systems first-touch policy and the OpenMP thread binding mechanisms, the data is placed as close to the processing core as possible. Since only a subset of the matrix data for the CG algorithm presented in Algorithm 1 is used by each of the processing threads, no additional traffic (e.g., for the cache-coherency) across the NUMA domains is required. However, the data of the vectors is needed by all threads and thus cannot be placed optimal. In order to avoid an overloading of one of the memory controllers it is distributed equally. Although the memory placement and thread binding on a Xeon Phi Coprocessor is not as important as on a huge NUMA system like the BCS machine, the same strategy is used for both.

For the determination of the operational intensity oi it is assumed that

$$c < nnz * (s_f + s_i) \quad (2.5)$$

and

$$n \ll nnz, \quad (2.6)$$

where n is the dimension of the matrix, nnz the number of non-zero elements, c the size of the last level cache of the system, s_f the size of a double precision value and s_i the size of a single integer variable. This allows the assumption that the complete matrix is too big to be kept in the cache on the one hand. On the other hand, one can assume that the right-hand side vector and the row index vector of the CRS matrix format can be kept in the last level cache on both systems and thus not need to be loaded or stored during one single spMV operation. Given this, only the data in the array of length nnz have to be taken into account for the determination of the amount of data operations $dops$. Since the algorithm computes the solution of the linear equation system in double precision, s_f bytes need to be load from the value array. In addition, s_i bytes are required from the row index. Assuming that an integer is stored as 32 bit value and a double precision variable as 64 bit value, the amount of data which needs to be loaded for the innermost part of the spMV can

2. Efficient Memory Access in OpenMP

be determined as

$$dops_{spmv} = s_f + s_i = 12 \text{ B}. \quad (2.7)$$

One add and one multiply operation is required and each matrix element is only needed once, so it cannot be kept in the cache for a later reuse. With Equation 2.3 this leads to an operational intensity for the spMV of

$$oi_{spmv} = \frac{2 \text{ FLOPS}}{12 \text{ B}} = \frac{1 \text{ FLOPS}}{6 \text{ B}}. \quad (2.8)$$

Given that, the upper performance limit for a spMV kernel on a BCS system is given as

$$L_{bcs} = \min(p, oi_{spmv} \cdot bw_{bcs}) = \min(1 \text{ TFLOPS}, \frac{1 \text{ FLOPS}}{6 \text{ B}} \cdot 200 \frac{\text{GB}}{\text{s}}) = 33.3 \text{ GFLOPS}, \quad (2.9)$$

and for the Xeon Phi as

$$L_{phi} = \min(p, oi_{spmv} \cdot bw_{phi}) = \min(1 \text{ TFLOPS}, \frac{1 \text{ FLOPS}}{6 \text{ B}} \cdot 155 \frac{\text{GB}}{\text{s}}) = 25.8 \text{ GFLOPS}. \quad (2.10)$$

Figure 2.8 depicts the applied Roofline Model for both systems including the ceiling for missing FMA or missing SIMD instructions (although not required for the model of a spMV). Since the operational intensity is quite low for the spMV kernel it is memory-bound on both platforms and can only reach a fraction of the theoretical peak performance.

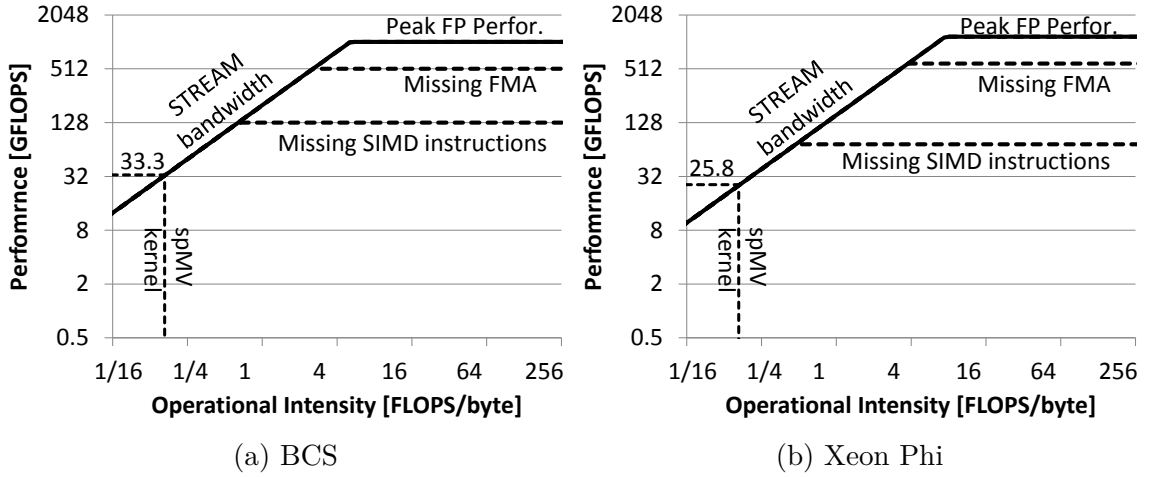


Figure 2.8.: Roofline Model applied to the Intel Xeon Phi and the BCS system.

The denser the matrix is, the more valid is the assumption in equation (2.6). However, for very sparse matrices (i.e., those holding only a couple of non-zero values per row) this assumption is not necessarily correct (depending on the size of the matrix) and the operational intensity oi needs to be determined differently (i.e.,

System	#Threads	Time [s]	dxpay/ daxpy	dot product	spMV
Xeon Phi	240	30.22	3.49 %	2.17 %	93.77 %
BCS	128	21.08	5.38 %	6.38 %	88.13 %

Table 2.2.: Execution time shares and the best solving times (1000 iterations) for the linear algebra kernels on a BCS system and a Xeon Phi Coprocessor.

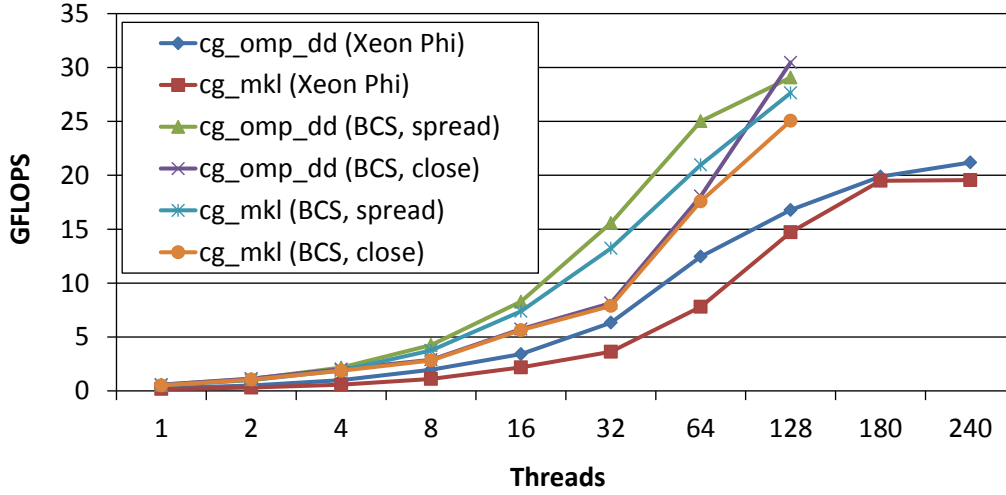


Figure 2.9.: Performance of the spMV within the CG method on the Xeon Phi coprocessor and the 128-core BCS system.

increasing the expected data operations *dops*).

Figure 2.9 shows the absolute performance of the spMV operation when executed within the CG benchmark in GFLOPS. It can be seen the maximum performance on the BCS system is about 30 GFLOPS for the hand-tuned version *cg_omp_dd*, which is close to the upper performance limit of $L_{bcs} = 33.3$ GFLOPS. Furthermore, the figure shows that the binding strategy is only important, if a subset of the machine is used (e.g., only half of the available cores). Here, one should use the *spread* policy in order to make use of all available memory control. The maximum performance of the benchmark on the Xeon Phi coprocessor is about 21.2 GFLOPS. Although the gap to the upper performance limit is higher compared to the BCS system, this is also close to the prediction of 25.8 GFLOPS. One of the reasons for not reaching the upper limit is that the indexed access of the vector in a spMV using the CRS storage format requires the use of the corresponding gather instruction in order to SIMD vector registers. In this case, the compiler is not able to apply this instruction automatically, as one can see in the generated assemble code. Thus the upper limit cannot be reached, where the influence of this issue is bigger on the Xeon Phi coprocessor, because of the register lengths, which is 512 bits instead 256 bits on

2. Efficient Memory Access in OpenMP

the BCS system. The performance of the MKL version is always lower compared to *cg_omp_dd*. Here, the reason for the lower performance (especially on the big BCS system) is that the hand-tuned implementation is less generic and thus has the full control of the correct data and thread placement.

In summary, applying the Roofline model to the spMV kernel shows how one can predict the performance of a new micro-architecture by combining two basic metrics: the theoretical peak performance and the memory bandwidth of the system. The manual determination of the operational intensity of a given application is more complex, especially for complex algorithms using many different operations. One way to automatize this determination is the use of hardware performance counters, which are included in most modern micro-architectures. This allows counting the floating point and load/store operations of a complete application or for a certain function, and thus use this information for the calculation of the operational intensity. However, the detailed analysis of this opportunity is out of scope of this work.

5. Evaluation with standard benchmark suites Beside the comparison of the basic performance characteristics, the paradigm-specific overheads, the scalability or the model-driven performance evaluation the performance of OpenMP programs of a new micro-architecture can be assessed by the application of standard benchmark suites. A popular example is the SPECComp benchmark suite [3, 64], which includes a number of science, engineering and data processing applications. All benchmarks in this suite can be executed with different sizes of data sets in order to enable the evaluation on different sized SMP systems. For the evaluation of newer OpenMP features like explicit tasks the Barcelona OpenMP Tasks Suite [30] can be used. For the evaluation of target device offloading the SPEC ACCEL benchmarks [51] are being ported from OpenACC to OpenMP at the moment. Another example for a standard benchmark suite is the NAS parallel benchmark suite [6]. In [81] these were used in order to evaluate the Xeon Phi coprocessor and compare the performance with the 2-socket Intel SNB based host system.

Table 2.3 shows the elapsed time for all benchmarks of the suite in data set size C for the Xeon Phi coprocessor and the SNB system. All measurements have been performed with the Intel 13 compiler. Here, the sequential time and the parallel time with the best effort performance are given, which was reached when executing the benchmarks with 32 threads (i.e., on all physical cores) on the SNB system or rather 240 threads (i.e., using all SMT capabilities) on the Xeon Phi coprocessor. The benchmarks are unmodified and have been cross-compiled for Xeon Phi for a native execution on the device. Depending on the benchmark, the speedup on the SNB system is between 6 and 24 and on the Xeon Phi system between 44 and 114. This shows that all benchmarks scale well on both systems and that the Xeon Phi coprocessor can deliver a good scalability for standard kinds of applications.

Table 2.3.: Runtime (in seconds) and speedup of the NAS parallel benchmarks on the Xeon Phi and a 2-socket SNB system.

Bench- mark	SNB			Xeon Phi		
	1 Thread	32 Threads	Speedup	1 Thread	240 Threads	Speedup
IS	23.12	1.38	16.75	192.49	2.46	78.25
EP	186.81	8.11	23.03	1518.42	13.34	113.82
MG	64.04	8.03	7.98	498.94	9.63	51.81
FT	306.11	19.19	15.95	2393.01	53.97	44.34
BT	1241.63	82.61	15.03	9433.52	132.29	71.31
SP	826.25	137.69	6.00	12264.29	164.59	74.51
LU	1109.76	62.23	17.83	9835.09	163.33	60.22

However, the more relevant metric *time to solution* shows the limited meaning of the scalability analysis again, because the elapsed parallel time is slower on the Xeon Phi coprocessor for each benchmark of the suite. Given the fact that the theoretical peak performance for each core on the both systems is roughly the same, but the Xeon Phi has much more cores, this result shows that for many real-world applications (represented by the benchmark suite), the Xeon Phi coprocessor is not the first choice or at least a reasonable performance cannot be reached without specific code tuning for that architecture. The reason for the bad performance on the Xeon Phi coprocessor is again the low single core performance.

2.5. Patterns for Task-parallel Programming on NUMA Architectures

As mentioned above, it is possible in OpenMP to generate explicit task which are executed in parallel instead of using implicit tasks generated by worksharing constructs, for instance. Typical examples for the usage of explicit tasks are recursive algorithms or programs with an irregular and dynamic workload balance. Here, an OpenMP runtime implementation has a huge freedom on how to schedule the generated tasks. A simple approach is to put all tasks into the same queue such that a thread of the current team can fetch it from the queue as soon as the thread completed a former task. However, this strategy requires a lock on the queue, which might limit the scalability of this naive scheduling algorithm. A second strategy is to manage the tasks in thread-local queues. This allows lock-free task processing as long as all threads have tasks in the local queue. In order to avoid idling threads, task-stealing from other thread queues can be applied as soon as the local queue is empty. In this case locking is required again, but not as global lock, because only two threads need to be synchronized.

2. Efficient Memory Access in OpenMP

In [88] and [89] task-parallel programming on NUMA architectures have been analyzed on different architectures and for different runtime implementations. Based on this work, this section will give an overview about the patterns a user can apply in order to benefit from tasking in OpenMP in terms of performance.

Related Work For the introduction of the tasking feature in OpenMP, Ayguadé et al. showed in an early work that an implementation can be as efficient as an implementation for worksharing constructs [5]. For the efficiency of tasking runtime implementations, the Barcelona OpenMP Task Suite [30] can be applied. However, this work does not focus on the behavior of NUMA machines as it is done in this thesis. In [69] the complex system characteristics are taken into account for the task scheduling algorithms. Here, Olivier et al. propose a hierarchical strategy by leveraging different methods like task-stealing on behalf of all threads within the same socket, which reduces the costs of remote stealing. Broquedis et al. presented in [16] a NUMA-aware memory manager, which converts the system information into scheduling hints, in order to address memory affinity issues. These hints enable a dynamic workload distribution. However, it has not a strong focus on the OpenMP tasking features. In addition to this two works, this thesis takes a hierarchical NUMA system into account. Furthermore, the patterns which can be used by a programmer are analyzed in detail with respect to the task generation and scheduling, including a deep analyzes of the implementation internals.

Tasking Patterns Basically, the differentiation regarding the tasking patterns can be made between two different approaches [88, 89]:

- **single-producer multiple-executors:** In this pattern all tasks are created within a `single` construct, which is nested in a parallel region. The created tasks can be queued and then executed by all other threads of the team. The advantage of this pattern is that it often requires only little changes to the code and the data structures. Furthermore, it is less error-prone, because the single thread task creation can avoid data races. The thread executing the `single` construct can put all data needed for the computation of the task into a corresponding `firstprivate` clause, which ensures the initialization of the private data. Since there is an implicit barrier at the end of the `single` construct, it is guaranteed that all tasks are processed up to end of the region. The downside of this approach might be that an implementation only queues the task to the local queue of the thread executing the `single` construct. Thus, all other threads have to apply task-stealing to the same queue. An example for this pattern is shown in Listing 2.5, where `N_TA` tasks are created by one thread and processed by a team of `N_TH` threads.
- **parallel-producer multiple-executors:** In this pattern all threads of a team create the tasks by using an additional worksharing construct. Listing 2.6 shows an example for this pattern. The `N_TA` loop iterations for the task

creation are distributed among all threads (e.g., with an static schedule). This might support a runtime implementation to queue the tasks locally to each thread-local task queue. Thus, a thread can first process the local tasks before it has to apply task-stealing. The schedule of the worksharing construct also has a performance impact if the workload of each task is different. The implicit barrier at the end of the for construct waits for the termination of all tasks. For the pattern a worksharing is not necessarily required, because the code within a parallel region is executed by all threads. However, without a worksharing construct the programmer has to ensure that not all threads do exactly the same work and that an adequate synchronization is performed at the end of a parallel region (e.g., an explicit barrier or an appropriate task synchronization construct).

```

1 #pragma omp parallel num_threads(N_TH)
2 {
3   #pragma omp single
4   {
5     int i;
6
7     for (i = 0; i < N_TA; i++)
8     {
9       #pragma omp task
10      {
11        some_computation();
12      }
13    }
14  }
15 }
```

Listing (2.5) Single-producer,
multiple-executors tasking
pattern.

```

1 #pragma omp parallel num_threads(N_TH)
2 {
3
4   int i;
5
6   #pragma omp for
7   for (i = 0; i < N_TA; i++)
8   {
9     #pragma omp task
10    {
11      some_computation();
12    }
13  }
14 }
15
```

Listing (2.6) Parallel-producer,
multiple-executors tasking
pattern.

For the evaluation of the two different patterns the CG benchmark can be used as a representative real-world kernel. As described in Section 2.3, the version *cg_task_sp* implements a single-producer pattern and the version *cg_task_pp* a parallel-producer pattern. The focus here is again on the spMV operation, because the matrix of the equation system can be very irregular depending on the underlying problem. In order to avoid performance issues it is important to avoid load imbalances. Since the best distribution can be calculated in advance for a spMV operation in a CRS storage format, the goal here is not to improve the performance of the corresponding version *cg_omp_dd*. Furthermore, with such a dynamic approach, the data placement on NUMA system cannot be guaranteed anymore. Nevertheless, the results of the tasking version are useful, because the version *cg_omp_dd* form an upper limit for the performance. Knowing this upper performance limit enables the assessment of the runtime implementation quality, which is much more difficult to do for algorithms where an advanced calculation of a good distribution is not possible. Thus, the tasking versions of the CG benchmark (or rather the spMV

2. Efficient Memory Access in OpenMP

operation) is used as a representative kernel for algorithms with a highly dynamic workload.

Data Affinity In order to reach a good application performance on a NUMA system, an appropriate data affinity is required. The main goal here is always to keep the data as local as possible to the processing thread and thus avoid expensive remote memory accesses across multiple NUMA domains. In case of a hierarchical NUMA machine with multiple levels of domains (like the used BCS system) this becomes even more important. For algorithms which allow an efficient static distribution of the workload (e.g., by using an OpenMP worksharing constructs), this can be ensured by initializing the data in parallel and using the *first-touch* policy [87]. This policy is used by most operating systems which are relevant for HPC applications. Here, one has to keep in mind that a memory allocation (e.g., a `malloc` call) will reserve the memory in the virtual address space, but the physical memory pages will not be stored immediately. Instead of that, the page will be stored into the physical memory which belongs to the core which first touches the address as long as free memory is available on this NUMA node. Otherwise the page will be stored in another domain where still free memory is available. By using the same (static) schedule for the data initialization as is used in the computation the programmer can ensure local data accesses (with exception of the page boundaries, if the chunks are not page aligned). However, for dynamic schedules or for explicit task this cannot be done, because the scheduling during the computation might be completely different as it was for the initialization. Although several studies [55, 54, 37] including an OpenMP affinity extension library [80] for the application of a *next-touch* mechanism exist, no OpenMP standard-compliant method for page migration have come to existence up to now. However, using the parallel-producer tasking pattern allows also providing a better data placement.

For the evaluation of this claim first of all an adequate analysis tool is required. In context of this thesis I developed the tool *numamem* for Linux operating systems, which is also installed on the compute cluster of RWTH Aachen University. The tool allows printing the current page distribution of a process across the NUMA domains. Furthermore, it can be used to sample the development of the page distribution during the execution of an application, where MPI/OpenMP hybrid applications are supported explicitly. The information needed for the analysis is directly acquired from the Linux kernel.

Figure 2.11 shows the page distribution for five different initialization strategies: (i) round robin initialization across the NUMA domain enforced by the *numactl* tool, (ii) parallel initialization with a worksharing construct and static schedule, (iii) row-wise initialization done by tasks in a serial-producer pattern, (iv) row-wise initialization done by tasks in a parallel-producer pattern, and finally (v) serial initialization done by one thread. The analysis has been performed with the *nu-*

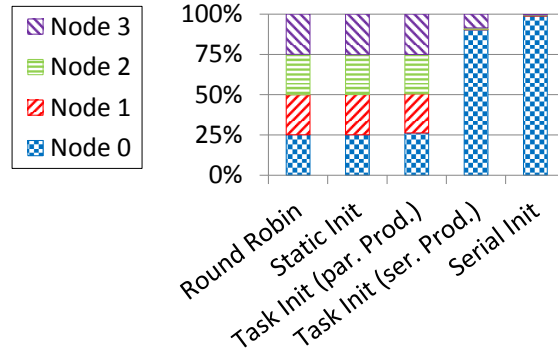


Figure 2.11.: Page distribution over the NUMA nodes after the matrix initialization.

mamem tool on the 4-socket machine presented in Section 2.2 and for the same matrix in CRS format used for the investigations with the CG benchmark (refer to Section 2.3). As expected all pages are stored on NUMA node 0 for the serial initialization, because of the first-touch policy. For the same reason the pages are equally distributed across all four domains when parallel initialized with a static schedule. The distribution with a round robin strategy looks exactly the same. However, although each node stores the same amount of pages, the memory access might not be local during computation, because the access pattern is not round robin on a page base. Nevertheless, having a round robin distribution is still better than having no distribution at all, because it at least increases the chance of local memory accesses. The comparison of the two different tasking initialization strategies clearly shows that data is distributed much more regular with parallel-producer pattern than with the single-producer. The reason is that the initialization tasks are computationally very cheap and short-lived and the other threads on NUMA node 0 – besides the one performing the task creation – execute the tasks at a fast enough pace. Thus, a significant amount of task stealing from the other nodes does not occur.

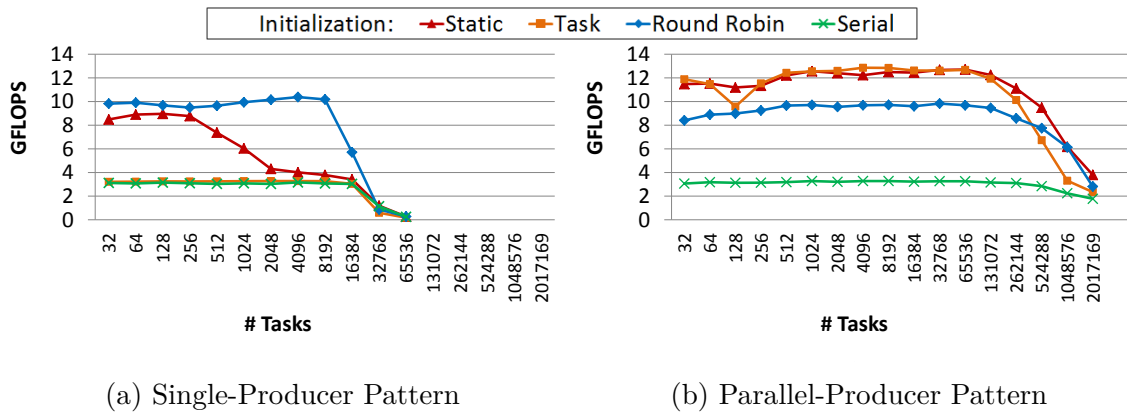


Figure 2.12.: Performance of the spMV kernel within the CG method for different initialization strategies and different amount of threads.

2. Efficient Memory Access in OpenMP

Evaluation Beside the assurance of a dynamic work distribution for a dynamic workload, the amount of used tasks has a huge performance impact on an application. On the one hand the use of too few tasks (e.g., only one task per thread) approximates a static runtime behavior and on the other hand too many tasks increase the overhead for the task creation and scheduling with respect to the amount of work per task. In both tasking versions of the CG benchmark only the spMV operation is parallelized with OpenMP task, all other operations use the same worksharing constructs as for the other OpenMP versions. The work for the spMV operation is distributed by chunks of matrix rows and the chunk size cs is the same for each task, calculated as

$$cs(ta) = \begin{cases} \lfloor N/ta \rfloor, & \text{if } N \% ta = 0 \\ \lfloor N/ta \rfloor + 1, & \text{otherwise,} \end{cases} \quad (2.11)$$

where N is the dimension of the square matrix and ta the number of tasks.

Figure 2.12 shows the performance of the spMV operation as part of the CG benchmark for both tasking patterns with different initialization strategies and different amount of tasks. The measurements have been performed with the Intel 12.1 compiler. The performance results correlate with the distribution of the memory pages across the NUMA domains as shown in Figure 2.11. A serial or a task initialization reaches a poor performance for the single-producer pattern as a consequence of the bad page distribution (refer to Figure 2.12a). A distribution with a round robin or static schedule strategy delivers about 10 GFLOPS, where for the latter one the performance decreases when more than 256 tasks are generated. This means that a chunk size of about $cs' = 7880$ rows is the limit for the static schedule distribution in conjugation with the single-producer pattern, whereas a round robin distributions still delivers a good performance with a finer task granularity of 8192 tasks, which corresponds to a chunk size of $cs'' = 247$ rows.

Furthermore, the figure shows that the parallel-producer pattern delivers a better performance in general (refer to Figure 2.12b). However, the performance for a serial initialization is still below 4 GFLOPS. The best results with about 13 GFLOPS have been obtained by using a static or a task-based distribution of the memory pages. The performance is not as sensitive regarding the amount of tasks as it is for the single-producer pattern. It drops down with more than $2^{17} = 131072$ tasks, which corresponds to a chunk size of only $cs''' = 16$ matrix rows per task. This shows that the pattern still works even with a huge amount of tasks or rather a fine task granularity. The fact that results for the parallel-producer variants in conjunction with a parallel initialization outperforms the others shows that the programmer has a much better influence on data locality by using this pattern.

Analysis and Assessment of Runtime Implementations All performance results presented up to this point have been reached by using the Intel compiler with the

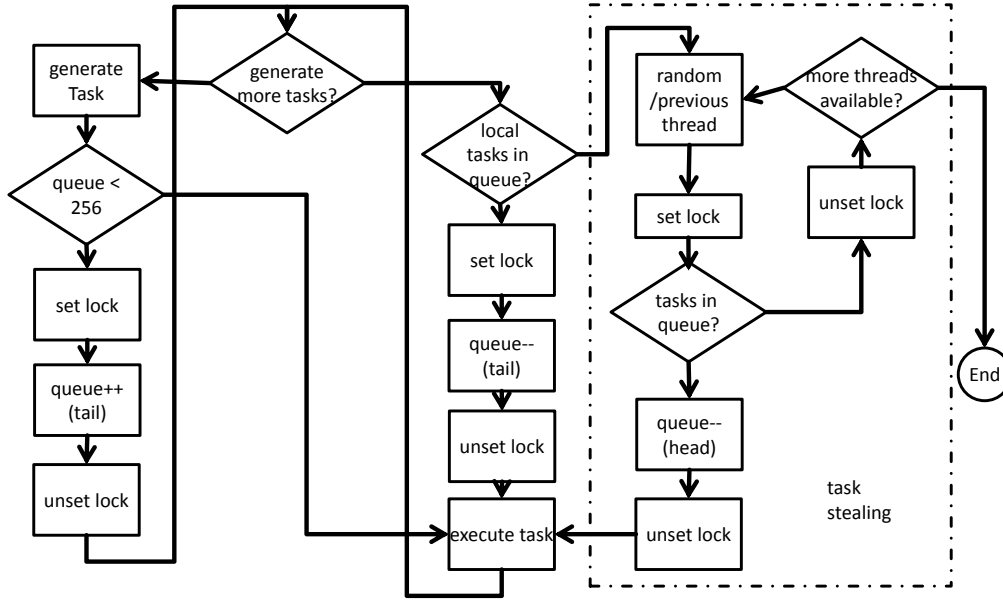


Figure 2.13.: Task scheduling in the LLVM runtime.

corresponding OpenMP runtime implementation. They showed that although the task scheduling is performed dynamically (and thus the memory placement cannot be optimal), good performance results can be reached by using a parallel-producer pattern in combination with appropriate memory page distribution. The reason for the much better performance is the use thread-local tasking queues in the runtime implementation. Figure 2.13 shows the principle control flow of each thread for the task generation and execution in the Intel OpenMP runtime. After a task was generated, it will be put to the end of the thread-local (double-ended) queue, if less than 256 (local) tasks are waiting. If there are already more than 256 tasks queued, the new generated task will be executed immediately. This will be done as long as no more tasks have to be generated. In this case each thread starts executing local tasks from the tail of the queue until it is empty. The right-hand side of the figure depicts the task-stealing algorithm implemented in the Intel OpenMP runtime. If it is the first time a thread has to steal a task from another thread queue, a random thread is chosen. If the queue of this random thread is not empty, the task will be removed from the head of the queue and executed. Furthermore, the thread number will be saved, in order to use the same thread queue for the next task-stealing, if the local queue remains empty. If a task-stealing was not successful, another thread will be picked until no more threads are available. In order to avoid data races within the runtime implementation all operations on the queues are locked.

As mentioned before, a runtime implementation has a huge freedom for the implementation of the task queues. The fact that Intel's implementation is using thread-local queues, does not mean that other vendors do the same. Thus, it is worth to analyze this more detailed. Figure 2.14 shows the performance of the two

2. Efficient Memory Access in OpenMP

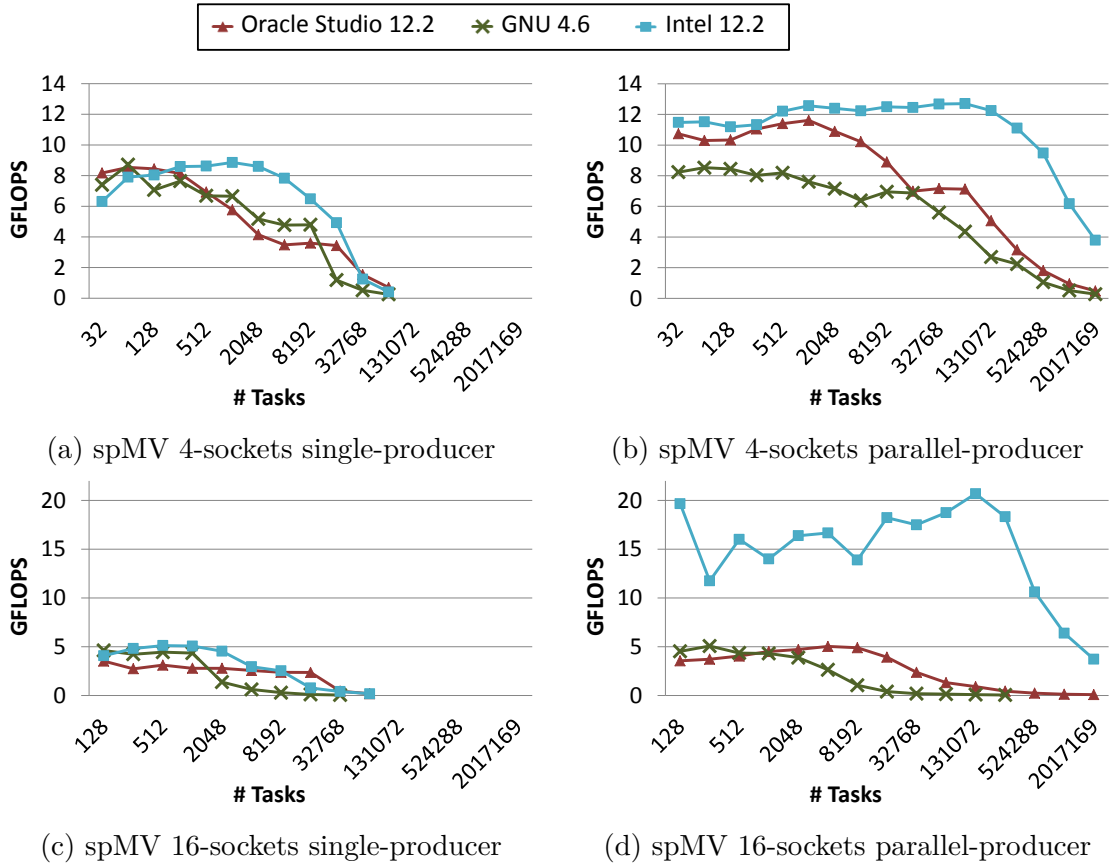


Figure 2.14.: Performance of spMV for different implementations.

tasking patterns for different runtime implementations. In addition to the Intel compiler 12.1, the CG benchmark was built with the Oracle Studio 12.2 and the GNU 4.6 compiler and executed with the corresponding runtimes. Figure 2.14a and 2.14b show the performance results of the spMV on the 4-socket Bull system. For the single-producer pattern all runtimes behave similar regarding the performance and reach a maximum of about 9 GFLOPS, where the Intel compiler slightly benefits from a small task granularity of up to about 8000 tasks. The parallel-producer pattern delivers a better performance for the Intel and the Oracle runtime. Although the maximum performance for the GNU compiler does not increase, the pattern is still beneficial, because it is less performance sensitive regarding the task granularity.

The results on the bigger 16-socket Bull system show that the importance of the parallel-producer increases with the system size. Here, the absolute performance is even lower compared to the 4-socket systems, although it has four times higher compute capabilities (refer to Figure 2.14c). While the performance of the Intel runtime increases by using the parallel-producer pattern, the performance for the other implementations remains poor (refer to Figure 2.14d). This clearly shows that the runtime implementations from Oracle and GNU are not prepared for the OpenMP

tasking paradigm on huge NUMA systems. The fact that the performance is poor independent of the used tasking pattern is a hint that task generation and scheduling strategy (or rather the queuing / task-stealing algorithm) is the same in both cases. This hypothesis can be proofed (at least for the GNU implementation) by comparing how many tasks are executed by a different thread than it was generated from. As expected, table 2.4 shows that more than 95 % of the tasks have been executed by a different thread when using a single-producer pattern for all runtime implementations and on both systems. For the GNU runtime implementation this also true for the parallel-producer pattern. In contrast, in the Intel runtime implementation only about 3 % of the tasks are executed by a different thread on the for socket system or rather about 9 % on the BCS system. For the Oracle runtime the usage of the local tasks is high on the 4-socket systems. However, the execution by a different thread increase to about 41 % on the bigger machine, which correlates with the performance results.

Table 2.4.: Percentage of tasks that were executed by a different thread than they were created from for the single- and parallel-producer pattern on the 4-sockets and 16-sockets systems using the CG kernel with 1024 tasks.

	4-sockets		16-sockets	
	single	parallel	single	parallel
Intel	96.21 %	2.87 %	99.22 %	8.61 %
GNU	96.87 %	96.90 %	99.04 %	99.14 %
ORACLE STUDIO	95.97 %	4.04 %	98.24 %	41.02 %

On the one hand, the much better performance of the parallel-producer pattern shows the importance of the deep knowledge of the paradigm and the parallel architecture. On the other hand, the performance optimization requires a good understanding of the (vendor-specific) implementation details (as it is depicted in Figure 2.13). While a performance analysis with a corresponding tool can help to understand the runtime behavior of a given application, the root of the performance issue might be ambiguous. The reason for that is that the root for the performance issue is within the runtime implementation and a portable tool does not have any information about these details. Thus, only the vendor-specific internal function names of the given implementation can be shown. Even if the performance tool shows which task was generated and executed by which thread, this analysis is not portable for every runtime implementation. A solution to this issues is standard-compliant tools interface which delivers information directly from the runtime system and allows a unique depiction of the runtime behavior for every runtime implementation.

2.6. Summary and Conclusion

This chapter presented a systematic approach for the assessment of OpenMP on target devices. This includes the evaluation of both – the micro-architecture and the software infrastructure or rather the application. The analysis in five steps shows on the example of a Xeon Phi coprocessor that the prerequisites for a good performance on this highly-parallel device is given by good basic performance characteristics, low paradigm-specific overheads and a good scalability. Furthermore, the model-driven performance analysis shows that a prediction of the upper performance limit for a certain application can be done by the determination of the operational intensity. However, the evaluation with standard benchmark suites shows that a good performance is not possible for real-world applications without architecture-specific code tuning or rather that not all classes of HPC applications are good candidates for the execution on the Xeon Phi coprocessor.

The evaluation of two different tasking patterns shows, that even without the knowledge of the runtime implementation an improvement of the performance can be reached by generating the tasks in parallel (instead of just executing in parallel). The reason for that is that those implementation using local task queues have to apply less task stealing and thus less (global) locks are required. Furthermore, the data access is better on large NUMA systems, because of better utilization of the local memory controllers. This is getting more important with an increasing size of the system.

Furthermore, this chapter showed that the performance of real-world benchmarks might be not as good as expected by analyzing the basic performance characteristics or the scalability of a given hardware/application. Thus, adequate tool support is required in order to analyze the application performance on a new (target device based) micro-architecture. One of the issues which slows down the development of tools for such new devices, is the lack of a standard-compliant interface for the tool support of target devices. In order to overcome this issue the next section will present a corresponding solution.

3. Improving Standard-compliant Tool Support

The development of massive parallel systems also drives the further development and extension of existing programming paradigms as well as the introduction of new programming approaches. Many of these approaches in the past often required a time-consuming rewrite of applications. However, with more complex and larger applications, programmer productivity came into focus and more user friendly methods came to existence. Here, especially directive-based concepts promise to shift some of the burden from the programmer to the compiler and the runtime environment. Nevertheless, even with user friendly paradigms, an adequate tool support is essential in order to analyze and optimize the performance or to check the correctness of an application. Although many state-of-the-art tools exist, they often use different methods for the data acquisition. One of the reasons is that, depending on the range of feature or the functionality, different runtime information is required. Depending on the specific programming paradigm, the absence of a well-defined, standard-compliant tools interface is another reason.

First of all, from a programmers perspective the tools method for the data acquisition or the usage of a standard-compliant tools interface is secondary, as long as a powerful tool exists for the given use case. However, the absence of such an interface means that an analysis tool has to be ported to every new hardware architecture and possibly for every runtime implementation, especially if vendor-dependent interfaces exist. Furthermore, for every new feature a new method for the data acquisition has to be developed and implemented. Last, the quality of the runtime information and the detail level might be limited, because (depending on the runtime vendor) no direct information from the runtime system can be acquired (e.g., if the implementation is closed-source). As a consequence, the effort a tool developer has to spend into the development and porting of a tool, might also limit the productivity of an application programmer, because the tool might not be available for the used platform or might lack on the support of latest features of the programming paradigm.

In order to fill this gap, modern programming paradigms like *OpenACC* [70] or *CUDA* [67, 68] define a special (vendor-independent) tools interface, which is part of the specification. However, in the de-facto standard programming paradigm for shared memory programming OpenMP such a standard-compliant tools interface is missing at the moment. For this reason, the ARB has published an API for first-

3. Improving Standard-compliant Tool Support

party performance tools¹ as a technical report [32] in 2013. This OMPT interface specifies a portable API to construct performance analysis tools for OpenMP. It is targeted to be part of the OpenMP 5.0 specification, which is expected to be released in 2017 or 2018. As a preparation for this future release version, another technical report (TR4) has been published in November 2016. This technical report includes the tools interface into the OpenMP specification and thus it will be part of OpenMP 5.0 as long as the ARB will not vote for a remove of the feature again.

This chapter is structured as following. Section 3.1 gives a brief overview on existing tools interfaces in OpenACC and CUDA. In Section 3.2 an introduction into the basic functionality and features of the OpenMP tools interface will be given, because the developed generic epoch model for OpenMP (refer to Chapter 4) is based on it. Section 3.3 discusses a proposal for the extension of OMPT for the target device features for heterogeneous programming, which are part of OpenMP 4.x. My main contributions in these sections are an evaluation and comparison of the design of the tools interface in OpenMP and OpenACC [27], the extension proposal [20] including a prototype implementation and its realization in the revised technical report [31]. At the moment of writing this thesis, the complete interface is still under development and exists in different versions. In order to differentiate these versions the following nomenclature is used:

- **Technical Report 2 (TR2):** The original proposal for an OpenMP tools interface as published by the ARB [32].
- **Revised Technical Report 2 (rTR2):** The revised TR2, as published by the OpenMP Tools Subcommittee [31]. This report includes changes based on the research presented in this thesis (refer to Section 3.3).
- **Technical Report 4 (TR4):** The public draft of OpenMP 5.0 including the tools interface, which was published in November 2016 [72].

¹The term *first-party* tool is used in order to express that the tool runs within the same address space of an application process. In contrast to that a *third-party* tool runs as a separated process (e.g., a debugger).

3.1. Interface Designs

As mentioned before, programming paradigms as OpenACC and CUDA define special tools interfaces which are part of the specification. This section gives a brief overview on the used designs for both paradigms.

The profiling interface for OpenACC provides a set of event callbacks that are invoked by the runtime during the application run. It provides events for runtime and device initialization or shutdown, events for the encountering of data directives (e.g., enter, exit, update etc.) or allocation/deallocation, events for kernel launching or enqueueing and wait events which are triggered when a host thread has determined that a kernel queue is empty. The function signature is very generic and thus the same for all specified events. With this interface a reliable method for tracing-based performance tools exist. However, there is no support for asynchronous sampling at the moment, which means that sampling-based performance analysis tools do not have any standard-compliant mechanism to obtain information from the runtime at discrete points in time.

Compared to the OpenACC interface, NVIDIA's *CUDA Profiling Tools Interface* (CUPTI) [68] is more complex and provides more features for the creation of profiling and tracing tools that target CUDA applications. CUPTI is divided into four different APIs: the Activity API, the Callback API, the Event API and the Metric API. For this work especially the Activity API is of certain interest, because it allows collecting a trace of events on a *Graphics Processing Unit* (GPU). Here, the events are stored in specific *activity records* in the devices. In order to reduce the overhead of reporting device events as soon as they occur, these events are stored in an *activity buffer* and transferred back to the host. For instance this might happen as soon as the buffer is full or the application stops. However, the point in time for the transfer of the activity buffer is not defined explicitly. The Activity API uses a callback mechanism in order to request and return activity buffers instead. Thus, these callback routines are invoked by the runtime system whenever CUPTI needs an empty buffer.

3.2. Overview about OMPT

Although no standard-compliant interface for OpenMP performance analysis tools existed yet, many tools with OpenMP-specific performance analysis and support came to existence [35, 85, 1, 53, 48, 36, 9]. These different tools use different techniques in order to obtain their performance information.

One popular method is to capture the relevant information by using source code instrumentation. In 2002 Mohr et al. proposed the *OpenMP Performance API* (POMP API) [61] in order to define a standard for a performance measurement

3. Improving Standard-compliant Tool Support

library. Here, the idea is to instrument a certain OpenMP application in order to monitor OpenMP events during the runtime. The proposal emphasizes that the instrumentation can be done either by the compiler or by a source-to-source translation. For the latter they provide a tool called *OpenMP Pragma And Region Instrumentor (OPARI)*, which is used by many popular performance analysis tools (e.g., TAU [85], ompP [35], KOJAK [62], Scalasca [36], VampirTrace [52]). The successor OPARI2, which includes several improvements, is used by the joint measurement infrastructure *Score-P* [53]. In order to enable a performance tool to analyze OpenMP programs, a tool developer can rely on the functions specified in the POMP API. With source-to-source instrumentation this approach is completely independent of the specific OpenMP implementation, because the source code is parsed and directly modified. Although this early proposal for a standard performance tools interface includes sustained concepts, it still has some drawbacks which led to a rejection in the OpenMP Language Committee.

One disadvantage is that the responsibility for the instrumentation is not clearly addressed, because it can be done by the compiler or a source-to-source instrumentor. In the latter case a recompilation of the entire OpenMP application is necessary. In the best case this makes the performance analysis slightly inconvenient, in the worst case it prevents an adequate analysis. For instance, this is the case if an OpenMP parallelized third-party library like *Basic Linear Algebra Subprograms (BLAS)* is used, because orphaned OpenMP directives might be used here.

Another drawback of the POMP API is, that the overhead can be significant (e.g., an iteration of a worksharing loop might consume less time than is required to monitor in a tool [32]). Due to the source-to-source instrumentation, a user can easily map the measured performance to his code structure. However, this interferes with the optimizations a compiler or a runtime can do and thus does not necessarily show correct runtime behavior.

A detailed comparison of OMPT and OPARI2 was done by Lorenz et al. [56]. They conclude that the overhead of both methods is low, but the performance information is more accurate when using OMPT. Another popular instrumentation method for source-to-source translation of OpenMP applications is the ROSE framework [76]. Alternatives to source-to-source translation are to either wrap the vendor-specific calls into the corresponding runtime by using a preloading mechanism or to use binary instrumentation. Both methods avoid the recompilation step of the program before the performance analysis. However, this approaches lack on portability, because the entry points differ between different runtime vendors and might change between different versions. Furthermore, a compiler might directly generate inlined code instead of invoking the runtime. Such inlined code regions cannot be recognized by a performance tool by using one of the above methods.

Tools like the Sun/Oracle Collector [48] or the Intel Amplifier XE [9] are based

on asynchronous sampling during runtime. Depending on the sampling interval, the measurement overhead is low compared to event-driven performance tool. In order to enrich the collected performance information with more details, parts of Intel's OpenMP runtime are instrumented and have a special communication mechanism with the tool. If an OpenMP application uses the Intel runtime, this leads to a higher level of detail in the presented performance measurements, because the information is directly delivered by the runtime. However, if a runtime from a different vendor is used, this detail level cannot be obtained, because of the proprietary interface.

Design objectives: In order to overcome these issues, the OpenMP Language Committee and the performance tools community intend to have OMPT as a standard-compliant interface. Here, one of the design objectives of OMPT is the portability. The goal is to simplify the port across different hard- and software stacks and to have a wider range of performance analysis tool available. Furthermore, OMPT allows a tool to associate costs during the execution to both: the OpenMP program and the OpenMP runtime. In order to increase the user and vendor acceptance, OMPT is not only designed to have negligible overhead when no tool is attached to the program, but also to avoid an unreasonable burden on the runtime developer as well as the tool developer. One strategy to achieve this objective is to define a set of features as mandatory and the more detailed features as optional. This allows a runtime vendor to implement a subset of the interface without losing standard compliance. However, this small set of mandatory features still enables a performance tool to rely on the fundamental information provided by all runtimes, as well as a possibility to gather and analyze more detailed information about the program behavior at execution time.

Events, states, inquiry functions: OMPT consists of three key components: events, states and inquiry functions. The events are intended to be used by instrumentation-based tools. Each event is implemented as a callback function directly invoked by an OpenMP runtime. In order to receive those callbacks, a tool developer has to register the events individually by providing a function pointer. Many events are begin/end pairs which enables a tool to measure the time of a certain OpenMP construct or region. For this work the complete set of runtime events (not only the mandatory) is essential, because they are used for the epoch model for correctness checking as described in Section 4. Here, the following selected event pairs are especially important:

- *thread_{begin,end}*: Invoked after a runtime created and initialized a thread and before it is destroyed. One of the objectives of OMPT is that the event sequence reports the program behavior during the execution, which is not necessarily the logical structure of an application. Thus, the event might or might not be called by each worker thread for each new parallel region, because an OpenMP runtime might use a thread pool management.

3. Improving Standard-compliant Tool Support

- *parallel_begin*: Invoked after a (implicit) task encounters a parallel construct. In OpenMP a new implicit task is either generated by encountering a parallel construct or by an implicit parallel region, which surrounds the complete program, all target regions or all team regions.
- *parallel_end*: Invoked after a parallel region was left including the closing synchronization barrier. This event is only invoked by the master thread of the parallel region.
- *barrier_{begin,end}*: Invoked before and after the execution of a barrier region.
- *implicit_task_begin*: Invoked before an implicit task begins to execute. Thus, the event occurs for each worker thread in a new parallel region.
- *implicit_task_end*: Invoked after the implicit task completes its work. This means that the event potentially marks the end of a parallel region of a worker thread.
- *idle_{begin,end}*: Invoked when a thread waits for work outside a parallel region. Basically, this means that between the begin and end no user code is executed. This information can be essential for a performance analysis tool, because it might give hints to underutilized threads (e.g., caused by a load imbalance).

Regarding to the OMPT specification a compliant OpenMP runtime has to maintain the current state information for each thread in addition. This state information is the second key component of OMPT. It can be queried by a sampling performance tool at any time and is an approximation of the thread's state. The states are categorized into idle states, work states, overhead states, barrier wait states, task wait states and mutex wait states. Furthermore, a thread may report an undefined state, if no association with the runtime is possible. This might be the case if the runtime spawns additional non-user threads for management purposes.

The third key component of OMPT are the inquiry functions. For a tool developer it is important to know that it is unsafe to call any OpenMP runtime library routine within one of the callback, because this may cause a deadlock or additional callbacks. For this reason several inquiry functions are defined in the interface in order to request additional information from the runtime. This includes information about the ancestor level of a parallel region or an explicit task, the thread or task id or information about the current task frame, for instance. The symbols for the inquiry functions must not be global symbols in order to prevent that they are called by a user application accidentally. A tool receives a corresponding lookup routine while initializing the interface instead. This lookup routine is used in order to resolve the pointer to the inquiry functions. With this mechanism it is guaranteed that the

interface functions are also available in in preloaded tool, if the OpenMP runtime is dynamically loaded.

Runtime vs. Source Code Instrumentation: One disadvantage of OMPT is the missing source code location in the callback events. This was not specified in the technical report (*TR2*) on purpose. The reason is, that an OpenMP runtime does not necessarily require to maintain this information. Thus, adding a parameter for the source code location to each callback event would add an additional implementation burden to the runtime developer, which contradicts on of the most important design objectives of OMPT. Since real-world applications might consists of thousands of lines of code, the missing information would massively limit the usability of a performance or correctness checking tool. To overcome this limitation three different approaches can be considered.

First, a proposal about changing the signatures in OMPT can be made to the Language Committee. From a tool developers perspective this is the most convenient approach. However, since this might directly influences the code generation done by the compiler such a change would lower the acceptance of the Committee and thus decrease the chance for standardization. It would add an unreasonable burden to the OpenMP runtime implementers. As a compromise the information of the source code location could be made an optional feature. The consequence for tools which need this information compulsively would be that alternative approaches for runtimes which do not provide the source code location have to be implemented anyway. Thus, this compromise is not suitable at all.

Second, a hybrid strategy can be implemented in an OpenMP tool. Here, a tool developer can combine the use of the tools interface with source-to-source or binary instrumentation. In this case the instrumentation provides only the mapping to the source code location. However, due to compiler optimization strategies there might be no direct mapping of the user directives to the collected performance information. Furthermore, this would make a recompilation of the application necessary, if source-to-source instrumentation is used. If binary instrumentation is used, the runtime overhead might be much higher.

Third, the source code location can be determined from the program's symbol table. The prerequisite for this method is that the application is compiled with including debugging information, because otherwise only the mapping to the assembler code is possible. For example the binary file descriptor library (libbfd²) can be used in order to translate the program symbol into the source code location. This has to be done in the performance tool while processing the callback event and cannot be done as a post-mortem analysis for all collected events. Unfortunately, this approach is not platform independent and adds some additional overhead to

²<http://sourceware.org/binutils/>

the program execution.

The strength of the tools interface compared to source code location is the accuracy of the performance information. This is, because internal parameters of the runtime can be handed over to the tool and thus a higher level of detail can be achieved. All compiler optimization artifacts are provided and thus a more realistic runtime behavior is reflected. Furthermore, the fact that no recompilation is necessary increases the usability of tool using the standard-compliant interface significantly.

3.3. Extension for OpenMP Target Devices

As mentioned before the OpenMP ARB published a technical report (*TR2*) in order to address the absence of a portable tools interface. However, this technical report only covers the OpenMP 3.1 functionality. The capability of OpenMP increases with every new release version, in order to address developments in parallel computer architectures and to improve expressiveness. One of the new features of OpenMP 4.x are the target constructs (refer to Section 2.1 for details), which enable code execution on an attached target device (such as a GPGPU or an Intel Xeon Phi coprocessor). As a consequence, also the OMPT interface needs to incorporate these new features. In [20] a corresponding extension has been proposed. Based on this proposal, the technical report has been revised [31].

Extended states, events and inquiry functions: For sampling-based performance tools new wait states have to be defined. Basically, three target wait states have been added to the revised technical report (*rTR2*):

- **omp_state_wait_target:** An OpenMP thread is waiting on a target region to complete.
- **omp_state_wait_target_data:** An OpenMP thread is waiting on a target data mapping operation to complete. In OpenMP this data mapping can be a copy operation to or from a device with distributed memory or just an address mapping to or from a device which shares memory with host.
- **omp_state_wait_target_update:** An OpenMP thread is waiting for a target update operation to complete. Here, a target update operation is similar to a target data mapping concerning the physically attached memory.

Furthermore, the interface has been extended for event-driven performance analysis tools. Table 3.1 gives an overview on which information is required for the target-specific extension and defines the exact point in time when an OpenMP runtime has

Description	Event (begin)	Invocation	
		<i>after</i>	<i>before</i>
Identification of target-specific OpenMP constructs	target	a task encounters a target construct	the target region is executed
	target data	a task encounters a target data construct	the variables are created or mapped in the data environment
	target enter data	a task encounters a target enter data construct	the variables are created or mapped in the data environment
	target update	a task encounters a target update construct	the data consistency is established
Data mapping operations	data map	–	a variable is mapped (event for each variable)
Kernel invocation	kernel submit	the target* begin event and after all variables are mapped	target function is invoked on a device

Table 3.1.: Relevant information for the target-specific OMPT events.

to invoke the corresponding callback. For simplification and a better overview only the begin and enter events are listed in the table. The end or exit events are defined correspondingly (refer to [20] for details).

The first group of events is used in order to identify when an OpenMP task encounters a target-specific construct (i.e., *target*, *target data*, *target enter data* or *target update*). The *target* begin event is invoked by an OpenMP runtime after the task encounters a target construct, but before the region is executed by a target device. The *target data* event is invoked after a task encounters a target data construct, but before the variables are created or mapped in the *device data environment*. In OpenMP a data environment is defined by those variables that are associated with the execution of a given region. Furthermore, for each device an implicit device data environment exists. If no corresponding associated variable to the original variable is present in the device data environment, a new one will be created and mapped. The *target enter data* event is similar to the *target data* event. It is invoked after the runtime encounters a target enter data construct and before the specified variable(s) are created or mapped into the device data environment. However, in OpenMP a target enter data construct is a stand-alone directive, which means that it has no associ-

3. Improving Standard-compliant Tool Support

ated user code. Furthermore, a target task is generated, which encloses the target enter region. This allows asynchronous data mapping by using the *nowait* clause. The definition of the event for the stand-alone data unmapping construct *target exit data* is corresponding. The *target update* is invoked after a task encounters a target update construct, but before any of the variables in the given list is made consistent.

All the events for the identification of the target-specific OpenMP constructs share the same signature. This basically means, that a performance analysis tool receives the same information (i.e., the same parameters) with this event. In detail, these are the id of the enclosing task, a target id, the device id of the executing device and a code pointer to the outlined function. A tool can use the different IDs in order to identify the code regions and map each begin event to the corresponding end event. Nevertheless, not all of the IDs have to be globally unique. In particular, only the IDs of the implicit tasks have to fulfill the requirement for globally unique IDs. Therefore, for instance a target ID has only to be unique within the same implicit task.

The point in time a construct is encountered during the runtime, does not say anything about the time a mapping (or a data transfer) takes. For instance, the time between a target begin and a target end event includes the execution of the complete user code in between. For that reason an additional *data map event* is required. The *begin event* is invoked just before a variable is mapped to or from a target device. The *end event* is invoked just before a variable mapping is completed. The time between the *begin* and the *end event* can be the actual mapping time and/or the transfer time to or from a target device. In general this event is intended to be invoked on a per variable basis. From a performance tool perspective this has the benefit, that each variable can be analyzed individually. For instance, a user can see if it is more beneficial to transfer a small amount of variables with a big chunks size or vice versa. Here, another advantage of OMPT compared to other techniques like source-to-source instrumentation can be seen. Due to the fact that the information is directly delivered by the runtime, a higher degree of information detail is available. However, OpenMP has a so called “as if rule”, which basically means that a compiler has the freedom to translate certain constructs into others which behaves the same. The main reason for this rule is to leave a high degree of freedom to a compiler/runtime vendor to do advanced optimization. For instance, although the OpenMP specification describes a `omp parallel for` as being the same as

```
1  #pragma omp parallel
2  {
3      #pragma omp for
4      {
5          ...
6      }
7  }
```

the compiler can translate it to the following:

```

1  #pragma omp parallel
2  {
3      #pragma omp for nowait
4      {
5          ...
6      }
7  }

```

This avoids having the additional implicit barrier at the end of the *omp for* construct. Since no user code is executed in between and the user cannot detect any difference, this barrier is not required at all. The OMPT instrumentation is defined here as applying to the implementation chosen by the compiler or runtime, but not the abstract execution model described by the OpenMP specification. A valid optimization for a runtime concerning the data mapping might be to transfer multiple variables at once in order to saturate the network bandwidth between the host and the target device (e.g., the *PCIe* bus). For this reason the current OMPT draft (*rTR2*) [31] intends to have a list of variables for one single *target data map* event instead of having only one variable per event. This slightly differs from the original proposal [20] and is the result of the discussion in the OpenMP Subcommittee meeting. In detail, the information delivered with the corresponding signature for data mappings are the following:

- *Target ID*: Indicates the instance of the target construct associated with this operation.
- *Number of items (nitems)*: The number of items that are mapped or transferred.
- *List of host pointers*: List of lengths *nitems* including the pointers which are mapped/transferred on the host device.
- *List of target pointers*: List of lengths *nitems* including the pointers which are mapped/transferred on the target device.
- *List of number of bytes*: List of length *nitems* including the amount of data which is mapped/transferred to or from a target device for each host/target device pointer.
- *Mapping Flags*: A set of flags that indicate two properties of the mapping. First, the direction of data motion. This can be *to*, *from* or *tofrom*. Second, the flags indicate if the data transfer is made synchronously or asynchronously. This information is important for a tool developer, because in the latter case a tool cannot visualize the transfer time between a host and a target device for the given set of variables. The absence of this information could easily lead to wrong conclusions about the runtime behavior of an application.

The last event in Table 3.1 is the *kernel submit*. This event is invoked by an OpenMP runtime before a kernel is submitted for execution on a target devices and

3. Improving Standard-compliant Tool Support

after all data is mapped. The event delivers the information about the target ID, the host side operation ID, the number of request teams and the number of teams which was actually granted by the runtime.

In addition to these new defined target-specific events, the revised OMPT specification (*rTR2*) intends to invoke a callback for the generated *target task*. In OpenMP a target task is a task that is generated by a *target*, *target enter data*, *target exit data* or *target update* construct and was introduced with version 4.5. The execution of this target task may be deferred if the programmer adds a *nowait* clause to the target construct, which makes the execution and the data transfers asynchronous. In order to distinguish a target task from an initial, implicit or explicit task a corresponding parameter is specified in the signature of the task create event.

OpenMP provides different runtime library routines in order to enable the application programmer to affect and monitor threads, processors or the parallel environment (e.g., to determine the number of threads of the current thread team or the properties of the underlying system). However, calling these routines within an OMPT callback might lead to infinite recursions, deadlocks or other undefined behavior. For instance, the invocation of such a runtime library routine might cause another callback, which might lead to an infinite recursion by calling the runtime again. Because of this restriction in OMPT, additional inquiry functions are required for the target-specific extension of the interface. The first proposed function (`ompt_get_target_device_id()`) returns the ID of the active target and can be called in any of the new defined event callbacks safely. The second inquiry function (`ompt_get_target_id()`) is similar and returns the current target region ID. If the used target device has a separated memory controller, it is also very likely that host and target device have different hardware components and run with different clock generators. Furthermore, typical target devices run different operating systems and distinct instances of an OpenMP runtime. However, for the performance analysis a tool has to put the host-sided and the device-sided events into a unified temporal order. Due to the absence of a global time stamp, a mechanism to synchronize the time stamps of the host and the target device is required. Thus, the extension proposal defines a corresponding inquiry function (`ompt_get_target_device_time`), which queries the current time stamp on given target device. With this function a certain target device time can be translated into the corresponding host time. All target device inquiry functions are only allowed to be called in the extend of a target region or in the one of the target-specific callback defined above. Otherwise the behavior is unspecified.

Target Buffering API: By definition, all of the events specified in Table 3.1 (based on [20]) are invoked on the device that encounters the respective *target*, *target data*, *target enter data*, *target exit data* or *target update* construct. This is important since OpenMP implementations might use multiple instances of a runtime (e.g., one for

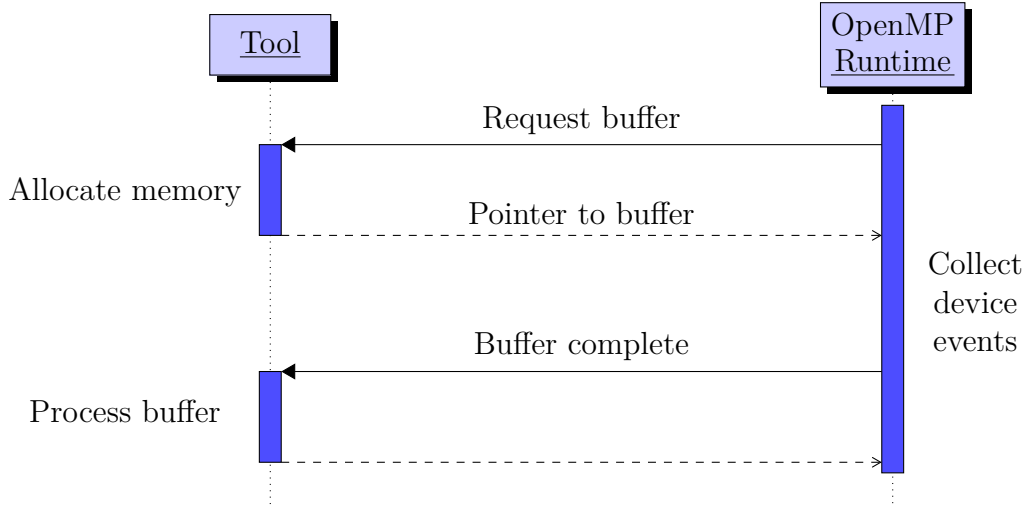


Figure 3.1.: Sequence diagram for OMPT buffering API.

the host and one for the target device). For instance this is the case, if a host and a target device does not share the same address space. Typically, these devices operate asynchronously with respect to a host device. In general, all OpenMP constructs can be used within a target region. Thus, also all kind of OMPT events can occur on a target device. In order to collect those events a tool developer needs a mechanism to transfer the collected performance information back to the host instance of a tool. In principal, two different approaches exist here.

First, a user can register a further tool instance for each target device. This tool instance has to be started on the target device and also has to register all events of interest as it would be done on the host device. This approach was used in [26] and [20]. Dietrich et al. developed a small library called *MIC Performance Tools Interface (libmpti)*, specially designed for the Intel Xeon Phi. The downside of this approach is that the tool developer has to implement a device-specific transfer mechanism in order to analyze the collected performance information on the host device. To do this in a portable way it is conceivable to use standard-compliant OpenMP mechanisms. This could be either an OpenMP device memory runtime routine or a **target data** construct with a corresponding **map** clause (refer to Section 2.1). However, as explained before, OMPT restricts the usage of both – OpenMP constructs and Execution Environment Routines – within a callback, because it might lead to infinite recursions, deadlocks or other undefined behavior. Thus, no portable, vendor- or hardware-independent mechanisms exist for the transfer of collected target performance information. In the extension proposal [20] an inquiry function was specified as a second approach in order to address this lack of portability. The discussion of this topic in the Tools Subcommittee lead to a slightly different interface for the target device tracing in the current revised technical report on OMPT (*rTR2*). The idea of this interface is to record events that occur during the execution

3. Improving Standard-compliant Tool Support

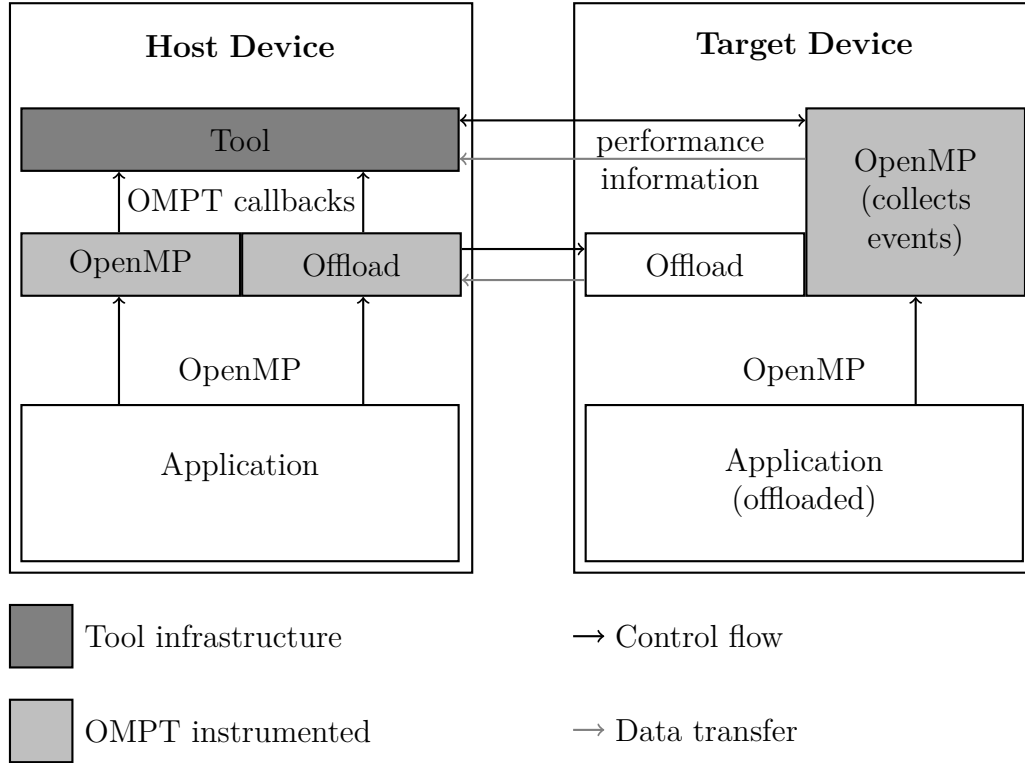


Figure 3.2.: Data transfer and control flow for target devices.

of a target region on the target device in a specific trace buffer. Figure 3.1 shows the sequence diagram for the tracing API. The runtime invokes a callback which requests memory for the trace buffer in the tool context. As soon as the memory provided by the tool is filled with events, a second callback will be invoked and the tool can store or analyze the trace buffer. The complete mechanism for the trace buffer registration and the performance information handling works similar to the mechanism specified in the CUPTI [68]. Figure 3.2 shows the data transfer and the control flow of the tracing interface. It can be seen that the main application is executed on the host device and the offloaded part (i.e., the target region) on a target device. The application is linked against two instances of the OMPT instrumented runtime library on the host and the target device. In addition, this example assumes a separated OpenMP and offloading library, which implements the complete functionality of the device constructs. An implementation might also combine the functionality in the same library. Since the behavior of nested target regions (also known as reverse offloading) is unspecified in OpenMP, the offloading library on the target device has not to be instrumented. The tool executing on the host device receives all registered callback directly from the runtime system. With the tracing interface no additional tool infrastructure is required on the target device. The communication is done directly with the runtime system instead. The control flow between the host and device is bidirectional, because the tool registers events in the runtime, the runtime requests memory in the tool context and the tool provides the

```

1  /* OMPT record */
2  typedef struct ompt_record_ompt_s {
3      ompt_event_t type;                /* event type */
4      ompt_target_time_t time;          /* record timestamp */
5      ompt_thread_id_t thread_id;      /* record's thread ID */
6      ompt_target_id_t target_id;      /* host context */
7      union {
8          ompt_record_thread_begin_t thread_begin; /* for thread begin */
9          ompt_record_idle_t idle;                /* for idle */
10         ompt_record_parallel_begin_t parallel_begin; /* for parallel begin */
11         ompt_record_parallel_end_t parallel_end; /* for parallel end */
12         ompt_record_task_create_t task_create; /* for task create */
13         ompt_record_task_dependence_t task_dep; /* for task dependence */
14         ompt_record_task_schedule_t task_sched; /* for task schedule */
15         ompt_record_scoped_implicit_t implicit; /* for implicit task */
16         ompt_record_scoped_sync_region_t sync_region; /* for sync */
17         ompt_record_target_data_t data; /* for target data */
18         ompt_record_target_data_map_t data_map; /* for target data map */
19         ompt_record_target_kernel_t kernel; /* for target kernel */
20         ompt_record_init_lock_t lock_init; /* for lock init */
21         ompt_record_lock_destroy_t lock_destroy; /* for lock destroy */
22         ompt_record_mutex_acquire_t mutex_acquire; /* for mutex acquire */
23         ompt_record_mutex_t mutex; /* for mutex */
24         ompt_record_scoped_nested_lock_t nested_lock; /* for nested lock */
25         ompt_record_scoped_master_t master; /* for master */
26         ompt_record_scoped_worksharing_t worksharing; /* for workshares */
27         ompt_record_flush_t flush; /* for flush */
28     } record;
29 } ompt_record_ompt_t;

```

Listing 3.1: The union type for the OMPT tracing API [31].

memory to the runtime system. The performance information is transferred to the tool as soon as the provided trace buffer is filled or the execution of the applications finishes. The data and control flow between the tool and the target device instance of the runtime is the logical flow. Physically, the communication might be between the different runtime instances only.

Record Format For the acceptance of such a standard-compliant tracing API, the data format of the buffer is essential. The current OMPT proposal (i.e., the technical report (*TR2*) as well as the revised (*rTR2*) document) specifies for each of the defined events its own signature. Although, some signatures are shared for similar events, the amount of information delivered differs. For instance, the length of the signature for a thread end event is 64 bit, because only the ID for the thread is handed over. In contrast to that, the length of the signature for a parallel begin event is 320 bit (assuming a 64 bit architecture). As a consequence, the collected buffer entries are different for each event type. In order to store them in the same buffer two approaches exist. First, the entries can be stored unaligned, where the current entry is stored directly next to its predecessor. This has the advantage, that no memory is wasted due to alignment padding. However, this approach complicates compiler optimization strategies on the one hand and it makes it more complicated for the tool developer on the other hand. The latter one can be solved by defining an iterator function in the interface. With the

3. Improving Standard-compliant Tool Support

help of this iterator function a tool developer can advance a cursor and thus process the complete tracing buffer without knowing the size of each entry in advance. Since both approaches have advantages and disadvantages, the current specification (*rTR2*) uses a hybrid strategy which defines a special *union*-type as can be seen in Listing 3.1 and two inquiry functions (`ompt_target_advance_buffer_cursor` and `ompt_target_buffer_get_record_type`) with an opaque handle to process the buffer. In each of the records the event type, the record time stamp, the thread ID and the host context is stored for each event at the beginning (Listing 3.1). This is necessary, because the information is required for an adequate performance analysis and it is not given implicitly as it would be the case outside of target regions. This is due to the fact that by buffering the events, the processing of the performance information changes from an on-the-fly analysis to an asynchronous one. The event-specific information is stored in the union-type `record`. Here, for each OMPT signature a corresponding structure was defined (refer to Appendix A for the complete definition). The length of the union `record` is determined by its biggest component, so that each buffer entry has the same size. This has the advantage, that a tool developer can use pointer arithmetic in order to process the buffer instead of using a specific iterator type. The drawback of wasted memory for smaller entries is expected to be negligible, because the signature of the more frequent events like implicit task begin are relatively long anyway, compared to the signatures of the less frequent events like thread end or runtime shutdown. However, the final decision about the data format has not been made yet and is substance of further discussion in the OpenMP Language Committee. A more detailed analysis of the record format will be discussed in Section 3.4.

In order to keep the maximum flexibility, future versions of OMPT should support both approaches for collecting target device performance information: the manual method and the tracing API. For device with separated memory (like GPGPUs or the Intel Xeon Phi coprocessor) this not important, because the data collection without the tracing API is very inconvenient anyway. However, OpenMP does not restrict having a target device within the same address space. For future devices fulfilling this characteristic it might be better to register and collect the callback events directly instead of buffering them in the tracing API. This is especially true if the target and the host region share the same OpenMP runtime instance. Furthermore, future extensions could extend the concepts of the tracing interface to the host device as well. For a tool developer this is a very convenient (asynchronous) analysis method, because the performance data collection is implemented in a standard-compliant fashion within the OpenMP runtime. However, for synchronous analysis this method is not suitable, because the point in time of the data delivery is up to the runtime implementation. Furthermore, this approach increases the burden for the runtime implementer.

As discussed above, the main motivation for the definition of the tracing interface are nowadays existing target device (such as GPGPUs or the Intel Xeon Phi copro-

cessor) and their distributed memory nature. Mainly the fact that several instances of the OpenMP runtime can run in parallel for the same application (depending on the number of used target devices) and existence of different, vendor-dependent communications interfaces require such a portable and standard-compliant tracing interface. However, from the tool developer perspective, the present well-defined data structures have another benefit, because of the generic design of the trace record structures. Even if the tracing interface is not used by a tool at all (e.g., because no analysis for target devices are done), the tool developer can still rely on the data structures in order to collect the event data in internal buffers without the need to redefine these structures for each tool. Furthermore, defining interfaces between different tools can be simplified.

3.4. Evaluation of the OMPT Extension

One of the most important design objectives of the OpenMP target constructs as well as the OMPT interface is to be architecture- and platform-independent and portable. The proposed target extension for OMPT including the tracing interface as described above is designated for the same demand. However, in order to show the applicability of the designed interface, this paragraph will evaluate the functionality of an LLVM-/Intel-based OpenMP implementation³. Although, OMPT is not part of the OpenMP specification yet, a prototype implementation of the interface is included in this runtime. For target offloading Intel uses an additional library called *liboffload*, which basic concepts are described in [66]. The compiler basically replaces all target-related directives with calls into the *liboffload*. The communication between the host and the target device is managed by the *Coprocessor Offload Infrastructure* (COI), which is a user-level offload library. Basically, this library enables the user to create processes on the Xeon Phi coprocessor (refer to Section 2.2), to create FIFO pipelines between the host and target device, to manage memory buffers, to transfer code and data or to invoke functions on the device. The abstraction of the communication over the PCIe bus is done in the *Symmetric Communications InterFace* (SCIF) layer. COI uses SCIF as low-level communication layer. The *liboffload*, COI and SCIF are part of Intel’s MPSS.

In order to evaluate the functionality of the proposed extensions, all target-related callbacks have been implemented and published on a separated branch⁴ of the LLVM runtime. Based on this implementation the technical university of Dresden extended Score-P, as discussed later (refer to Section 5.2).

The implementation of the buffering API is based on COI. Figure 3.3 depicts the implementation details in the *liboffload*. Here, a new **Tracer** class was added to each

³<http://openmp.llvm.org>

⁴<https://github.com/OpenMPToolsInterface/LLVM-openmp>

3. Improving Standard-compliant Tool Support

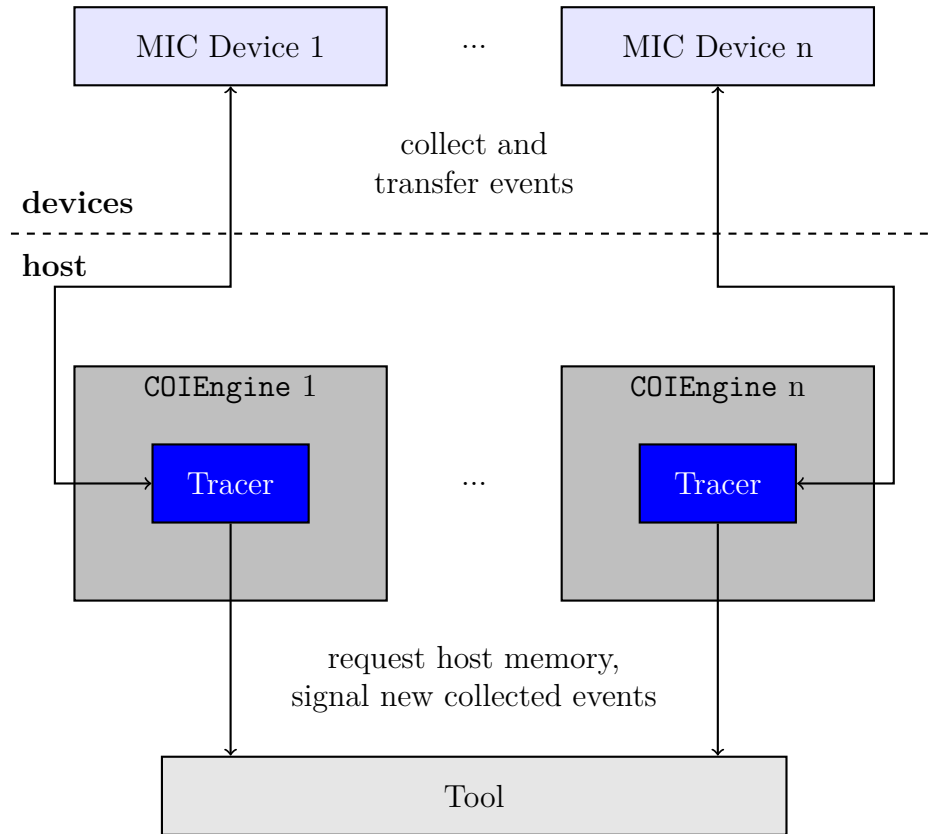


Figure 3.3.: Tracing API implemented with COI.

instance of the **COIEngine**. The **Tracer** class encapsulates all tracing-related methods. It manages the device-sided event registration and the transfer of the collected OMPT events. The complete tracing-related signal handling between the host and the target device over COI is done here. The amount of **COIEngines** depends on the number of attached devices. Thus, separated tracing of multiple devices is possible. The **COIEngines** instances run on the host only and communicate directly with the devices over COI. The buffer structure for the performance information is based on the native **COIBuffer** of the COI API. Here, the advantage is that the mapping of host and device memory is managed by COI. The events for each device are collected per thread in private buffers. On the one hand this means that a tool has to handle many different buffers for devices with a huge amount of threads. On the other hand this strategy avoids global locks on the tracing buffers on the device. This is an important design decision, because the amount of events occurring during runtime and the number of threads on such a device can become large. Thus, global locks on a single tracing buffer can become very expensive in terms of performance. With one buffer per thread the overhead within the liboffload is limited to the local store of the events and the transfer back to the host at certain points in time. However, the design of OMPT gives runtime vendor the freedom and flexibility to implement the buffer management differently, without losing standard-compliance. For instance,

a runtime could request only one global buffer instead of one buffer per thread. The size of a buffer is specified by the tool in the corresponding OMPT callback. As soon as the buffer is full, the collected performance information is transferred back. The responsibility for the allocated buffer remains within the liboffload, which means that a performance tool cannot expect the pointers to be valid outside the callback. Thus, the performance information needs to be copied or directly analyzed.

Although the implementation of the tracing API is fully functional, some performance issues need to be discussed. As soon as a device thread encounters the first event or after a buffer is full, this is signaled to the host. The host itself hands over this request to the tool by calling the corresponding callback. All of these requests are blocking and the device thread is waiting until the tool has allocated the memory, instead of executing the further instructions. However, this potential performance issue can be limited by storing the performance information in a small cache and copy them into the final buffer as soon as it is available. Since the COI API has no direct support for this at the moment, corresponding extensions of the interface are necessary. A similar performance issue occurs for full buffers. Here the device thread again has to wait until the attached tool processed the records in the buffer. Thus, the overall performance crucially depends on how long the tool is in the corresponding buffer callbacks. Furthermore, this directly influences the runtime behavior on the target device. As a consequence, a performance tool developer should proceed the buffer as fast as possible in order to keep this influence on the application performance small. A simple copy of the performance information is favored over direct performance analysis in the callback. The analysis should be deferred to a point outside the OMPT callback.

Another minor limitation of the current implementation is, that it is not possible to transfer all events from the target device back to the device. The events for a thread end or the runtime shutdown on the device are collected, but cannot be transferred back to the device, because the liboffload is already shutdown when these events occur.

Trace Record Format As discussed in the Section 3.3, the current revised technical report (*rTR2*) has a hybrid approach regarding the tracing record format, which give an OpenMP runtime implementer the freedom to choose between the more convenient and the compact one. This paragraph will evaluate the memory overhead of the first, which avoids that each buffer entry has a different size and thus the need for special iterator functionality. The disadvantage of a higher memory overhead is expected to be negligible, because the more frequent events have a longer signature anyway. In order to evaluate this hypothesis, the following section will analyze the LLVM OpenMP runtime with a representative workload of target-specific applications. For this representative workload selected benchmarks from a preliminary version of the SPEC ACCEL [51] Target Offloading suite where

3. Improving Standard-compliant Tool Support

Record Event Type t_i	Minimal Size [B]	Actual Size s_i [B]
ompt_record_thread	8	8
ompt_record_thread_type	12	16
ompt_record_new_thread_type	12	16
ompt_record_wait	8	8
ompt_record_parallel	16	16
ompt_record_new_implicit_task	16	16
ompt_record_new_workshare	24	24
ompt_record_new_parallel	36	40
ompt_record_task	8	8
ompt_record_task_switch	16	16
ompt_record_new_task	32	32
ompt_record_t	64	72

Table 3.2.: OMPT Record Sizes.

used. Since the analysis focus the memory overhead and not the performance, the small test data sets were used. The benchmarks were executed on an Intel Xeon Phi 5110P coprocessor with 60 cache-coherent cores clocked at 1.053 GHz and 8 GB memory. For all measurements the Intel 16.0 compiler and MPSS 3.4.3 was used. Furthermore, a small OMPT tool was implemented, which registers all events on the target device and collects them in thread local buffers by using the specified tracing mechanism. The buffer size per thread is set to 17 KB. Thus, a corresponding `ompt_buffer_complete_callback` event will occur as soon as the buffer limit exceeds in order to collect more events. The tool counts the amount of records for each record type and determines its size. Table 3.2 lists all trace records as implemented in the current version of the liboffload. The records slightly differ from those specified in the revised technical report (*rTR2*), because the implementation is on the level of the original technical report (*TR2*), not on the revised one (refer to Listing 3.1). However, the differences are negligible, because the length of the signatures did not change significantly and most changes concern only the name of the record. The minimal size for each record is the sum of the length of each component in the structure. However, most compilers will add padding to the structure in order to improve the performance by guaranteeing aligned access to the memory. This is also done by the Intel compiler, as one can see on the actual size s_i . The table shows that the minimal record size is 8 Byte for the records `ompt_record_thread`, `ompt_record_wait` and `ompt_record_task`. The biggest record is the record for a new parallel region (`ompt_record_new_parallel`) with a length of 40 Bytes. The minimal size for the whole record, as it is written into the buffer, is determined by the sum of all common components (i.e., the event type, the time, the thread ID and the device activity ID) and the biggest record type. Thus the size s_r for each buffer entry is determined by

$$s_r = s_c + p + \max(s_i), \quad (3.1)$$

where s_c denotes the length for all common components, p the padding added by the compiler and s_i represents all buffer types. Thus, the length for `ompt_record_t` is

$$s_r = 28 \text{ Byte} + 4 \text{ Byte} + 36 \text{ Byte} = 72 \text{ Byte}. \quad (3.2)$$

The overhead o_i for each record type t_i can be determined by

$$o_i(s_i) = \frac{s_r}{s_c + s_i} - 1 = \frac{72 \text{ Byte}}{28 \text{ Byte} + s_i} - 1. \quad (3.3)$$

The overall overhead o_t is defined as

$$o_t = \frac{\sum_{i=0}^{i_{max}} a_i * (s_i + s_r)}{\sum_{i=0}^{i_{max}} a_i * (s_i + s_c)} - 1, \quad (3.4)$$

where a_i denotes the amount of collected events of type t_i and i_{max} is the number of event types.

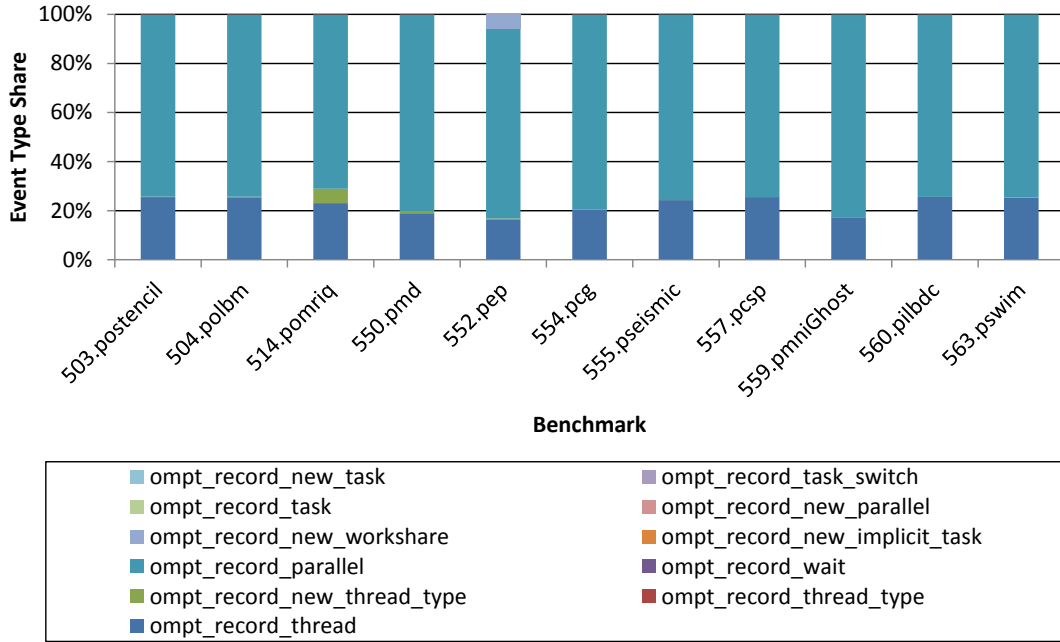


Figure 3.4.: Share of OMPT event types in selected SPEC ACCEL benchmarks.

Figure 3.4 shows the share of the different event types in the buffer for all executed SPEC ACCEL benchmarks. It can be seen that `ompt_record_parallel` has the most significant share, followed by `ompt_record_thread`. The first one has a memory overhead of

$$o_i(\text{ompt_record_parallel}) = \frac{72 \text{ Byte}}{28 \text{ Byte} + 16 \text{ Byte}} - 1 = 0.64, \quad (3.5)$$

which means that 64 % of the transferred data does not contain any information for this event type. The overhead for the second one is $o_i(\text{ompt_record_thread}) = 1$,

3. Improving Standard-compliant Tool Support

which is even higher and means that twice as much data as actually needed is transferred. As depicted in the figure, all other event types are less significant and thus the overall overhead is dominated by the two event types mentioned above.

Record Type t_i	Amount a_i	Payload [MB]	Size [MB]	Overhead o_i
thread	3,094,973	111	223	1
thread_type	0	0	0	0.64
new_thread_type	236	0.010	0.017	0.64
wait	0	0	0	1
parallel	9,010,238	396	649	0.64
new_implicit_task	0	0	0	0.64
new_workshare	0	0	0	0.38
new_parallel	19,761	1.344	1.423	0.05
task	0	0	0	1
task_switch	0	0	0	0.64
new_task	0	0	0	0.20
SUM	12,125,208	509	873	$o_t = \mathbf{0.7}$

Table 3.3.: SPEC ACCEL 557.pcsf: Tracing buffer sizes for all event types.

Since the share of the traced event record types are similar for all executed benchmarks, the overall overhead is also comparable. Table 3.3 shows the measurement results for the benchmark *557.pcsf*. The `ompt_record` prefix of t_i was removed for a better overview in the table. Due to the iterative character of the code, this benchmark has with about 12 million events the highest amount of transferred data of all executed benchmarks. Again, it is dominated by the two event types mentioned above. For the event type `ompt_record_parallel` about 649 MB of data was transferred from the target device to the host, where only about 396 MB relevant information was included. For the complete run more than 509 MB of performance information was transferred. Taking the fact into account, that all benchmarks were only executed with the small test data set and this one had only a runtime of about 150 s, this is a big amount of performance data. Although the transferred data per thread ($\frac{873 \text{ MB}}{240 \text{ Threads}} = 3.64 \frac{\text{MB}}{\text{Thread}}$) relativizes the amount of information, this shows that it might not be useful to collect all the data for such a highly-parallel device.

The overall overhead for this benchmark is $o_t = 0.7$. On the one hand this shows that expectation of a negligible memory overhead is wrong, because its impact on the interface is much higher than expected and should be further discussed in the OpenMP Tools Subcommittee. Especially, the question whether it is better to have a convenient tools interface with unique record type size or one with a small additional memory overhead. On the other hand, other interfaces have a similar design. In OpenACC for instance, only one signature for all events exists at all. Thus, storing the information in the generalized data structures results in a corresponding

memory overhead as well. Furthermore, one has to remind that although the data structure delivered to a host thread is given by the specification for both interfaces, there is optimization potential for the runtime developer. In the used prototype implementation the specified trace record `ompt_record_t` (refer to Listing 3.1) was also used for the transfer from the target device to the host. However, the specification does not enforce that. The data can be transferred in a completely different internal buffer structure in order to avoid the expensive data transfer (e.g., over PCIe) and stored in the standard-compliant buffer on the host afterwards. In order to avoid additional copy operations on the host, the corresponding transport layer could directly take this into account. Unfortunately, this is not possible for the prototype implementation in the liboffload, because the sources for the COI layer above are not publicly available. Furthermore, the iterator inquiry function to advance the buffer cursor works with the opaque buffer handle, but not a pointer directly into the buffer. This means, that the unpacking of a single trace record during the buffer processing does only require the additional memory for a single record and not the complete buffer including the padding added by the `union`-type. The unpacking is done by the inquiry function `ompt_target_buffer_get_record_type`, which returns a pointer the record which may or may not point into the internal OpenMP runtime buffer. Tracing is known to have a big overhead in general and thus an adequate filtering for long running applications with many threads is necessary anyway. This can either be done by the tool developer by only registering a subset of events or by the user if a corresponding functionality exists in the tool (e.g., by defining a specific analysis type).

3.5. Summary and Conclusion

In this chapter an overview of the OMPT interface was given. The main contribution here is the specification of a complete target-specific extension for this interface. The requirements were analyzed and advantages and disadvantages were discussed in detail. Furthermore, the proposal was intensively discussed with the OpenMP Tools Subcommittee, which led to a formal specification in the current revised technical report (*rTR2*). Based on this, the new technical report 4 (*TR4*) was extended by the complete OMPT interface, which is the draft for the future OpenMP 5.0 specification.

For the evaluation of the functionality and applicability of the proposed extensions, the defined interface was implemented in a wide spread open source OpenMP runtime. The memory overhead of the tracing interface was analyzed based on this implementation on an Intel Xeon Phi coprocessor. The insight of this examination is important for the further steps for the integration of the interface in OpenMP 5.0 as well as for specific improvements of a runtime. An evaluation of the performance overhead of the complete extension and its applicability to real-world performance analysis tools will be done in Section 5.2.

3. Improving Standard-compliant Tool Support

4. Epoch Model for OpenMP

In this chapter a generic and formal OpenMP epoch model will be presented. The main objective of the model is to divide the execution of an OpenMP program into phases in order to determine a strict partial order for each of the phases, which enables to analyze which parts are potentially executed concurrently. This allows tools to validate the correctness of a given application. The model is based on [24] and was extended by OpenMP target and task constructs. Although the epoch model focuses on OpenMP, it is applicable to other programming paradigms correspondingly.

The chapter is structured as follows. Section 4.1 introduces the required terminology for the model. In the following two sections the epoch generation is defined in two steps. First, Section 4.2 builds a formal method to generate the thread-local epochs by using the OMPT events as input. Second, the algorithm which is used to assign these partial (thread-local) epochs to global epochs is described in Section 4.3. The model will be evaluated in Section 4.4, before it will be extended by the OpenMP *task* construct in Section 4.5. Finally, Section 4.6 briefly discusses the transferability of the epoch concept to further parallel programming paradigms.

4.1. Terminology

For the epoch model three hierarchy levels based on [83] and [84] are defined: events, periods and epochs.

Definition 1 (Event) *An event is a memory event or an OMPT event observed in an OpenMP thread at runtime, where the following event types are defined and Σ_{ev} denotes the set of all events:*

Definition 1.1 (Memory event) *A memory event is a read/write memory access or a memory allocation/deallocation. The set of all memory events will be denoted as Σ_{mem} .*

Definition 1.2 (OMPT event) *An OMPT event is a callback performed by an OpenMP runtime as defined in the OMPT interface. The set of all OMPT events will be denoted as Σ .*

Definition 1.3 (Synchronization event) *A synchronization event is an OMPT event which implies that all threads of the same team are synchronized by the OpenMP runtime. The set of all synchronization events will be denoted as Σ_{sync} .*

4. Epoch Model for OpenMP

During the execution of a given program, a tool can collect a list of such events for each OpenMP thread. Furthermore, the recorded OMPT events are used to divide the observed memory events into disjoint periods:

Definition 2 (Period) *A period contains all memory events $\sigma_{mem} \in \Sigma_{mem}$ of an OpenMP thread that occurred between two consecutively observed OMPT events $\sigma_i, \sigma_j \in \Sigma$. The set of all periods will be denoted as Ψ .*

Since an OpenMP application will neither start nor end with an OMPT event, there is an initial and a final period in addition to this definition. The initial period will contain all observed memory events before the first encountered OMPT event and the final period will contain all observed memory events after the last encountered OMPT event.

The periods are still defined with respect to a single thread base and thus do not provide any information about synchronization points of the OpenMP application. However, this information is required in order to determine the execution order of certain memory events and periods executed by multiple threads. Thus, the concept of epochs is introduced:

Definition 3 (Epoch) *An epoch contains all periods of all OpenMP threads that occur between two synchronization events $\sigma_i, \sigma_j \in \Sigma_{sync}$. The set of epochs will be denoted as Ω .*

An epoch groups multiple periods of multiple threads with respect to a certain synchronization event. Since an OpenMP program can consist of multiple teams, not all active threads of an OpenMP program must pass this event. This holds for programs using multiple nested parallel regions. Typical examples for synchronization events are the *barrier_end* event or the *parallel_end* event. Furthermore, the event *parallel_begin* is defined as synchronization event, because the execution was single threaded or (in case of nested parallelism) a nested thread wakes up idling child threads. Thus, all (nested) threads are synchronized implicitly. The definition of the *parallel_begin* event as a synchronization event (*parallel_begin* $\in \Sigma_{sync}$) already shows that the definition in this epoch model slightly differs from the definition of a *synchronization point* given by the OpenMP specification. This includes the following constructs: `master`, `critical`, `barrier`, `taskwait`, `taskgroup`, `atomic` and `ordered`.

Definition 3.1 (Thread-local epoch) *A thread-local epoch is a subset of an epoch $e \in \Omega$, which only contains the periods of the same OpenMP thread.*

The definition of a thread-local epoch is required for the determination of the epochs, because the epoch generation has to be done on a per-thread basis first (refer to Section 4.2).

Epoch Types: Three different types of epochs are defined: *master epochs*, *parallel epochs* and *nested epochs*. This distinction is used with respect to the OpenMP execution model, which defines one master thread and $n - 1$ worker threads per parallel region. This organization unit is called a thread team of size n .

A *master epoch* covers the part of execution where only the master thread executes user code. This does not necessarily mean that this epoch is single-threaded, because an OpenMP runtime may decide to fork threads at any time or not to destroy the thread after a join. Many runtimes manage their threads in a thread pool instead, in order to avoid the overhead for creating and destroying threads for OpenMP applications using many parallel regions during execution. As described in Section 3, OMPT specifies corresponding idle events which might be part of a master epoch. However, it is important to understand that the worker thread will not execute any user code in a master epoch. The begin of a master epoch can either be marked by the initial period or a *parallel_end* event. The latter one is the case after the join of a parallel region.

A *parallel epoch* contains periods encountered within the same parallel region of an OpenMP program. All events before a *parallel_begin* are executed by the master thread of the new created parallel region. This does include the creation of the thread team and thus the execution of the `parallel` construct. Thus, a *parallel_begin* event actually marks the end of the previous epoch and is not assigned to the begin of the parallel epoch. Since the OpenMP execution model allows nested parallelism, this previous epoch can be a master, parallel or nested epoch. A parallel region might consist of multiple parallel epochs, because multiple synchronization events (e.g., explicit barriers) might occur during execution. Thus, the end of a parallel epoch is marked by a *barrier_end* or an *implicit_task_end* event.

A *nested epoch* contains periods encountered in a nested parallel region. Thus, a nested epoch always runs in a parallel context as well. However, the validation of memory events as data race free within a nested epoch does not imply a data race free program, because multiple nested epoch can be executed in parallel. In contrast to that, a program can only execute one parallel epoch at a certain point in time. Figure 4.2 illustrates the concept of master, parallel and nested epochs. This concept will be discussed in detail in the evaluation section with a certain example code (refer to Section 4.4). Multiple nested epochs might be nested in a parallel nested epoch. For instance this is the case if explicit barriers are encountered within a nested parallel region. Furthermore, the depth of the epoch nesting level depends on the nesting level of the OpenMP program and is not limited in the model.

The differentiation between these different epoch types is important, because depending on the epoch type different algorithms for the correctness check (e.g., data race detection) are necessary. For instance, within a master epoch data races caused by the user code are impossible. However, data races within the OpenMP runtime

4. Epoch Model for OpenMP

are still possible. In the case of the existence of one or more nested epochs some kind of hierarchical data race detection is needed in order to avoid false positives. Here, a data race detection algorithm can first analyze the inner most nested epochs. If no races are detected in the inner most level, this epoch can be merged to the next outer level and can be processed as if only the outer thread executed the complete nested epoch in serial. This can be repeated recursively until only the most outer level remains.

Happen-before Relations: The main goal of the presented OMPT-based epoch model is the determination of a strict partial order of certain events, periods and epochs. Based on [83] and the terminology use for the data race detector ThreadSanitizer [84] the following definitions will be used:

Definition 4 (Happens-before relation) *A happens-before relation (denoted by a “ $\prec$ ”) is a strict partial order of a set of events, periods or epochs.*

Thus, the happens-before relation guarantees that a certain *event*, *period* or *epoch* precedes another. For epochs this means:

Definition 4.1 *Given two epochs $e_1, e_2 \in \Omega$. Then it holds $e_1 \prec e_2$ if e_1 is observed before e_2 and one of the following conditions is fulfilled:*

- $\exists \sigma_{sync} \in \Sigma_{sync}$, where σ_{sync} is the first event in the first period of e_2 and σ_{sync} is observed before e_2 .
- $\exists \sigma_{sync} \in \Sigma_{sync}$, where σ_{sync} is the last event in the last period of e_1 and σ_{sync} is observed before e_2 .
- $\exists e_i, e_j \in \Omega : e_1 \preceq e_i \prec e_j \preceq e_2$ (transitive).

If a synchronization event separates two epochs, a happens-before relation can be determined. This is either the case if the synchronization event is the last event of that epoch, which was observed first or if the synchronization event is the first event epoch of that epoch, which was observed secondly. Furthermore, the partial order of epochs is transitive, such that a happens-before relation can be determined if an epoch in between exists which already determines the happens-before relation.

For periods, the definition is as follows:

Definition 4.2 *Given two periods $p_1, p_2 \in \Psi$, where p_1 is a period contained in epoch e_1 and p_2 is a period contained in epoch e_2 . Then it holds $p_1 \prec p_2$ if p_1 is observed before p_2 and one of the following is true:*

- $e_1 \prec e_2$.
- p_1 and p_2 are executed by the same OpenMP task (explicit or implicit).

- $\exists p_i, p_j \in \Psi : p_1 \preceq p_i \prec p_j \preceq p_2$ (*transitive*).

The happens-before relation of two periods is determined by the observed order if they are executed by the same OpenMP task. If the task is implicit this implies that the periods p_1 and p_2 have been executed by the same OpenMP thread as well. However, if the task is explicit the execution by the same thread is not sufficient for the determination of the happens-before relation, because the order of execution depends on the task scheduling behavior and thus is not runtime/timing independent. Otherwise, the strict partial order can only be satisfied if a happens-before relation for the corresponding epochs exists. As for the epochs, the happens-before relation of two periods is transitive.

The relation for events is defined as following:

Definition 4.3 *Given two events $\sigma_1, \sigma_2 \in \Sigma_{ev}$, where σ_1 is an event contained in period p_1 and σ_2 is an event contained in period p_2 . Then it holds $\sigma_1 \prec \sigma_2$ if σ_1 is observed before σ_2 and one of the following is true:*

- $p_1 \prec p_2$.
- σ_1 and σ_2 are executed by the same OpenMP task (*explicit or implicit*).
- $\exists \sigma_i, \sigma_j \in \Sigma_{ev} : \sigma_1 \preceq \sigma_i \prec \sigma_j \preceq \sigma_2$ (*transitive*).

The happens-before relation of two events is determined by the observed order if they are executed by the same OpenMP task. If the task is implicit this implies that the events σ_1 and σ_2 have been executed by the same OpenMP thread as well. However, (as for the periods) if the task is explicit the execution by the same thread is not sufficient for the determination of the happens-before relation, because the order of execution depends on the task scheduling behavior and thus is not runtime/timing independent. Otherwise, the strict partial order can only be satisfied if a happens-before relation for the corresponding periods exists. As for the epochs and the periods, the happens-before relation of two events is transitive.

If no happens-before relation of two *events*, *periods* or *epochs* can be determined, they are called *concurrent*, which will be denoted as “ $\not\prec$ ”. This does not necessarily mean that they are always executed at the same time in a certain system, but it means that this can potentially happen, depending on the runtime behavior (i.e. the timing). This includes that by definition events or periods are concurrent even if they were executed by the same thread, but a different (explicit) task.

The happens-before relation is also defined between events, periods and epochs. If an event σ_1 is contained in a period p_1 and a relation to a period p_2 is given as $p_1 \prec p_2$, then also $\sigma_1 \prec p_2$ holds. This is also defined between periods and epochs and between events and epochs correspondingly.

4.2. Thread-local Epoch Generation

For the first step of epoch generation an extended *Pushdown Automaton* (PDA) is defined. The first definition was done in [24]. The PDA presented in this thesis was extended to include more OpenMP features. In addition to pushdown automata as described in [2] this PDA has a finite set for the output alphabet and a corresponding output function as used for mealy machines [59]. Thus, the PDA is defined as 9-tuple,

$$M = (Q, \Sigma, \Omega, \Gamma, \delta, \lambda, q_0, Z, F), \quad (4.1)$$

where

- $Q = \{q_m, q_p, q_t, q_{h1}, q_{h2}, q_{h3}, q_s\}$ is the set of states. Here, q_m marks the state for master regions and q_p the state for parallel regions. The state q_t is used for target regions. $q_{h\{1,2,3\}}$ are auxiliary (or rather intermediate) states. q_s marks an invalid sink state.
- Σ is the set of all OMPT events (input set) as specified in the technical report 2 (*TR2*) [32]¹. $\sigma(t)$ denotes that the event was triggered by thread t . If no thread number is denoted, it is not of certain interest or given by the context implicitly. Selected input symbols $\sigma \in \Sigma$ are listed in Table 4.1.
- $\Omega = \{e_m, e_p, e_n, \epsilon\}$ is the output alphabet. Here, e_m represents a master epoch, e_p a parallel epoch and e_n a nested epoch. ϵ is the empty string, which basically means that no epoch is generated. In the presented model each $\omega \in \Omega$ is basically a sequence of input symbols $\sigma \in \Sigma$. By definition the last σ of this sequence is the one read before the current transition, not the one which actually triggered the current transition. This is referred to as *lazy epoch generation*.
- $\Gamma = \{X, Y, \epsilon\}$ is the stack alphabet. The stack alphabet is used to store the nesting level of an OpenMP application, where X denotes a non-nested region and Y a nested region. The nested region can be either spanned by a nested parallel constructs or by OpenMP target regions. ϵ denotes the empty string.
- $\delta : Q \times \Sigma \times \Gamma \rightarrow \mathcal{P}(Q \times \Gamma)$ defines the transition relation, where $\mathcal{P}$ denotes the power set of $Q \times \Gamma$.
- $\lambda : Q \times \Sigma \times \Gamma \rightarrow \Omega$ defines the output function.
- $q_0 = q_m$ is the start state.

¹The names, signatures and amount of events in the original technical report 2 (*TR2*) [32] differ from those specified in the revised technical report 2 (*rTR2*) [31]. Since the functionality remains identically, the epoch model is still applicable, but might need some modifications in a way that an input symbol depends on the parameters delivered with the event.

- $Z = M$ is the initial stack symbol.
- $F = \{q_m\}$ is the set of accepting states. If no accepting state is reached after the execution, the OpenMP program is invalid.

Input Symbol	OMPT Event
σ_s	<i>start</i>
$\sigma_{th}_{\{b,e\}}$	<i>thread_{begin,end}</i>
$\sigma_p_{\{b,e\}}$	<i>parallel_{begin,end}</i>
$\sigma_{ta}_{\{b,e\}}$	<i>implicit_task_{begin,end}</i>
$\sigma_{tk}_{\{b,e\}}$	<i>task_{begin,end}</i>
σ_{ts}	<i>task_switch</i>
$\sigma_{tw}_{\{b,e\}}$	<i>task_wait_{begin,end}</i>
$\sigma_{it}_{\{b,e\}}$	<i>initial_task_{begin,end}</i>
$\sigma_{tt}_{\{b,e\}}$	<i>target_task_{begin,end}</i>
$\sigma_b_{\{b,e\}}$	<i>barrier_{begin,end}</i>
$\sigma_{wb}_{\{b,e\}}$	<i>wait_barrier_{begin,end}</i>
$\sigma_l_{\{b,e\}}$	<i>loop_{begin,end}</i>
$\sigma_{se}_{\{b,e\}}$	<i>section_{begin,end}</i>
$\sigma_{si}_{\{b,e\}}$	<i>single_in_block_{begin,end}</i>
$\sigma_{so}_{\{b,e\}}$	<i>single_others_block_{begin,end}</i>
$\sigma_{ws}_{\{b,e\}}$	<i>workshare_{begin,end}</i>
$\sigma_i_{\{b,e\}}$	<i>idle_{begin,end}</i>
σ_{li}	<i>lock_init</i>
σ_{ma}	<i>mutex_acquire</i>
σ_{md}	<i>mutex_acquired</i>
σ_{mr}	<i>mutex_release</i>
σ_{ld}	<i>lock_destroy</i>
σ_{nl}	<i>nest_lock</i>

Table 4.1.: Selected input symbols $\sigma \in \Sigma$ based on the OMPT events.

The reason for using a PDA instead of a less complex finite-state machine (FSM) is that the additional capabilities of a PDA are required in order to express the OpenMP semantics. OpenMP does not only allow to parallelize simple for-loops, but also enables the use of complex parallel patterns like nested parallelism, task-based approaches or asynchronous target device offloading. Thus, the automaton has to store the parallel nesting level for instance, which is not possible with a FSM and a finite set of states and transitions. In this model the nesting level is represented by the depth and the symbol γ on the stack Γ . As an alternative to the PDA a Petri net [74] can be considered. Petri nets generalize the automata theory due to the opportunity of modeling concurrent events. By applying a set of new rules it is possible to model, analyze and simulate distributed and dynamic systems. However, since the PDA presented in this thesis is the same for each of

4. Epoch Model for OpenMP

the threads, the additional capabilities of this generalization are not required for generation of the thread local epochs as specified in Definition 3.1. For the merge of the local epochs as described in Section 4.3, this might be beneficial. However, the complexity analysis will show that the corresponding algorithm is adequate for typical OpenMP program. For this reason this alternative approach is not further discussed in this thesis.

Figure 4.1 visualizes the PDA including the transition relations and output functions. Not all combinations of the input symbols $\sigma \in \Sigma$ and of the stack symbols $\gamma \in \Gamma$ are depicted in the figure, in order to simplify the representation of the automaton. By definition, all missing relation transitions δ lead to the non-accepting invalid state q_s (sink state). As defined above, an output symbol $\gamma \in \Gamma$ is a sequence of input symbols $\sigma \in \Sigma$. In order to determine this sequence, a transition relation is executed in three steps:

1. State switch.
2. Stack operation(s).
3. Event push back.

Here, the order in which a transition is executed ensures that the current event is always pushed back on the correct stack level, if an implementation of the PDA uses an additional data stack for the event collection.

The complete PDA has to be passed by each OpenMP thread (workers and master) separately, which can be done in linear time $\mathcal{O}(n)$ (where n is the amount of epochs). All host threads start in q_m and will stay in this state for all $\sigma \in \Sigma \setminus \{\text{target_task_begin}, \text{initial_task_end}, \text{implicit_task_begin}\}$. Since an epoch is not defined on a per thread basis, but most epochs will contain periods of multiple threads, an additional merge step is necessary (refer to Section 4.3).

From a user's perspective only the master thread exists at the beginning of an OpenMP program until the first **parallel** or a corresponding combined construct is encountered (fork-join model). However, a runtime implementation might decide to fork several threads at the beginning of a program and manage them in a thread pool. Furthermore, most runtimes will not destroy the threads after the join of the parallel region, but put them back into the thread pool instead. This strategy has the advantage that the overhead of thread creation/destruction is avoided for applications with many parallel regions, because the work can immediately be assigned to an idling thread as soon as a new task is available. In OMPT the events *ompt_event_idle_begin* and *ompt_event_idle_end* mark that a worker thread idles outside a parallel region. Thus, even in a master state q_m one might read input symbols from a worker thread. Furthermore, these events will be part of the generated master epoch e_m . This issue is solved by just assigning these input symbols to

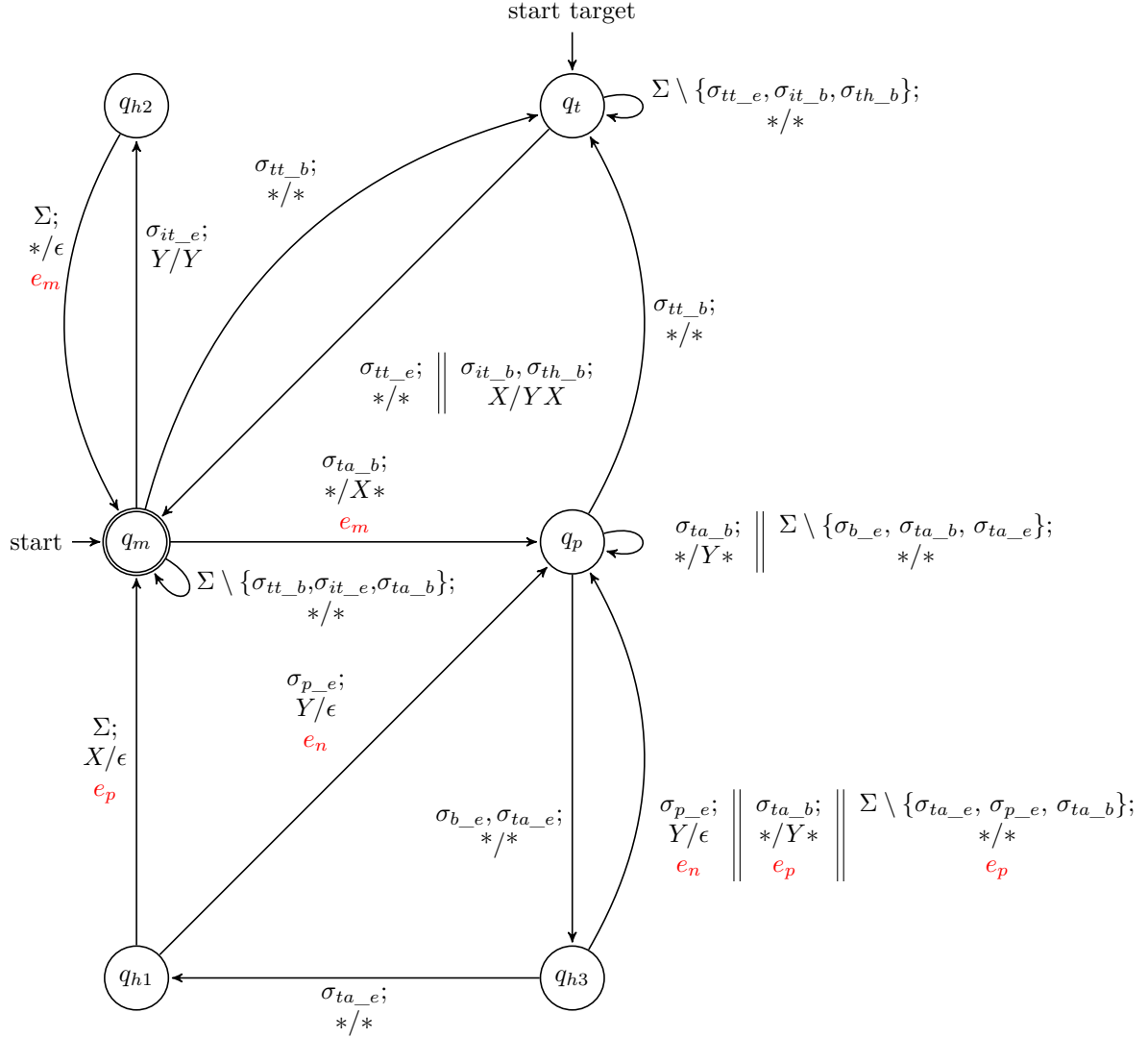


Figure 4.1.: Pushdown Automaton for the epoch generation. The transition relations are labeled as following. First row: input symbol $\sigma \in \Sigma$ (refer to Table 4.1); Second row: stack operation $\text{pop}(\gamma_i \in \Gamma) / \text{push}(\gamma_j \in \Gamma)$, where a $*$ can be any $\gamma \in \Gamma$; Third row (if exists): output symbol ω . If no ω is specified in the third row, the empty string ϵ is emitted implicitly. The symbol $\parallel$ separates different transitions, if there are multiple transitions between two states.

4. Epoch Model for OpenMP

the current master epoch, even if they occurred early or later. Since no user code is executed during the idle phase, this does not limit the usability of the model. The reading of these symbols σ can be seen in transition $q_m \rightarrow q_m$.

In OpenMP an *implicit task* is always generated when a parallel construct or an implicit parallel region is encountered. Thus, the PDA can react to the corresponding event directly, instead of defining a state switch for an *parallel_begin* event (which is also defined as synchronization event). Furthermore, in OpenMP an *initial task* is always associated with an *implicit parallel region*. Implicit parallel regions surround the whole OpenMP program, all target regions and all team regions. As soon as an *implicit_task_begin* is read, an additional X will be stored on the top of the stack (transition $q_m \rightarrow q_p$). Relating to the OpenMP specification, this input symbol always implies that either a parallel region or a target region is encountered by a master or worker thread. In the state q_m the event always implies that a parallel construct (or a corresponding combined construct) was encountered, because in the case of a target construct the PDA switches to q_t before the implicit task can be started on a target device. This works, because *target_task_begin* is always followed by the corresponding *implicit_task_begin* event. If the event before an *implicit_task_begin* was a *parallel_begin*, the current thread is a master thread and worker thread otherwise. Thus, by reading the this symbol a master epoch e_m will be generated (transition $q_m \rightarrow q_p$) for all threads. Furthermore, this symbol will be the first symbol in the event sequence of a parallel epoch e_p (as defined above), because after an *implicit_task_begin* the first user code within the parallel region will be executed.

Since an epoch is a sequence of periods, all input symbols σ will be collected in order to realize the output function λ . Here, the model defines that those symbols σ which lead to an output function $\lambda \neq \epsilon$ belong to the next epoch, not the one which is generated with the current transition (refer to the definition of Ω above). This is important, because some input symbols may or may not close the current epoch. For instance the OMPT event *barrier_end* will complete an epoch if an explicit barrier inside a parallel region was encountered. However, in case of an implicit barrier (e.g., after encountering the end of a parallel region) not the *barrier_end* event will complete the epoch, but the *implicit_task_end* event. OMPT does not distinguish between explicit or implicit barriers. However, this information is required for the epoch model, because one has to decide whether the next epoch will be a master or a parallel/nested epoch and the *implicit_task_end* event cannot be part of an master epoch. Thus, the final decision if the *barrier_end* completes the epoch or not, has to be done after reading the next input symbol. This is referred to as *lazy epoch completion*. The importance can be seen in the transition relation $q_p \rightarrow q_{h3}$, where a *barrier_end* is read, which does not complete the parallel epoch immediately (no output function). In case of an explicit barrier for instance, the completion is done in transition relation $q_{h3} \rightarrow q_p$. Other reasons for this transition can be multiple worksharing constructs with *nowait* clause nested in the same parallel region or end

of a nested parallel region (refer to next paragraph for details). The symbol read in this transition is the first period of the next parallel epoch. If an *implicit_task_end* is read ($q_{h3} \rightarrow q_{h1}$), it will be part of the current epoch as well, so that all outgoing transitions of q_{h1} generate an epoch. After reading the next symbol the epoch will be completed for the implicit barrier.

Nested Parallelism and Target Regions. OpenMP allows multiple nested parallel regions. For the epoch model this means that in the parallel state q_p a further symbol σ_{ta_b} for an *implicit_task_begin* can be read (transition $q_p \rightarrow q_p$). In this case a Y is pushed on the top of the stack, so that an unlimited nesting depth can be realized. A nested epoch is always completed with a *parallel_end* (transitions $q_{h3} \rightarrow q_p$ and $q_{h1} \rightarrow q_p$), providing that a Y is on the top of the stack. In this case the output symbol e_n is emitted and the Y is popped from the top of the stack. Within a nested parallel OpenMP region any number of further parallel epochs can be generated, for instance by encountering an explicit barrier (transition $q_{h3} \rightarrow q_p$). Thus, any OpenMP program with any parallel nesting level can be covered with this model.

Since version 4.x OpenMP allows the execution of code regions on a target device. The memory of such a device might or might not be shared with the host device. Typical examples for devices with non-shared memory are GPGPUs or the Intel Xeon Phi coprocessor. From the fact that target code regions might be executed on a target device with or without a shared memory follows that an specific OpenMP implementation might reuse the host threads or fork new threads on the device. This is reflected in the epoch model. For devices or implementations not using the same memory, the epoch generation of the new forked threads does not start in q_m , but in q_t . Nevertheless, with the input symbol *thread_begin*, which is always the first OMPT event for all threads, the PDA directly switches to q_m . Thus, the PDA behaves the same as it would do on a host device for all following input symbols. However, an additional Y is pushed on the top of the stack (transition $q_t \rightarrow q_m$) in order to differentiate between a host region and a target offload region. Basically, the concept of nested epochs presented for the nested OpenMP region is also used for target regions. For runtime implementations which are using the same thread pool or executing the target region on the host device, the state of the PDA switches from q_m to q_t by reading the input symbol *target_task_begin* and back to q_m as soon as an *implicit_task_begin* is read. This works, because according to the OpenMP specification [71] an implicit task is always generated by an implicit parallel region or when a parallel construct is encountered. The former is the case here, because an implicit parallel region surrounds the whole program as well as all target regions. This implicit parallel region ends by reading the input symbol *implicit_task_end*. In case of a Y on top of the stack it is known, that the current epoch introduces the end of a nested epoch (e.g., a target region). In this case the state of the PDA switches to q_{h2} (transition $q_m \rightarrow q_{h2}$). By reading the next input symbol a (nested) master region is generated and the Y on the top of the stack is

4. Epoch Model for OpenMP

removed (transition $q_{h2} \rightarrow q_m$). Target regions can not only be encountered by the master thread of an OpenMP program, but also by work threads within a parallel region (transition $q_p \rightarrow q_t$).

OpenMP also allows asynchronous target regions by adding a **nowait** clause to the target directive. In general the epoch model supports such asynchronous target offloading. The behavior is the same as for target devices starting in a non-shared memory environment or in a separated thread on the same device. The PDA computation again has to start in q_t and directly switches to q_m with the first *thread_{begin}* input symbol for all threads. Unfortunately, at the time writing this thesis no run-time implementation supporting asynchronous offloading exists and thus the behavior cannot be evaluated in the corresponding section.

Relation to the OpenMP Semantics. This subsection constructs a formalized PDA in order to enable the determination of the execution phases for each OpenMP thread with respect to its concurrency state for any given OpenMP program. The strength of this approach is the well-defined behavior and the direct relation to the OpenMP semantics including concepts like nested parallelism and target device offloading. The generic manner of the PDA guarantees the maintainability within a correctness checking tool and the extendibility of the model for any future extension of the OpenMP specification without introducing side effects.

4.3. Epoch Merging

As mentioned in Section 4.2 the epoch computation of the PDA has to be done for each thread (master, worker and nested threads) separately. As a result the PDA delivers the thread-local epochs (Definition 3.1). However, the definition of an epoch (Definition 3) includes the periods of all thread contributing in the current parallel region. Consequently, a last merging step is required, which assigns the thread local epochs to the global epochs.

Algorithm 2 Merging of epochs

```

1: procedure MERGEPOCHS(mtid)
2:   ▷ Go through all threads
3:   Epochs  $\leftarrow$  GetEpochs(mtid)
4:   for tid  $\leftarrow$  0 to numThreads do
5:     ▷ Skip master thread
6:     if tid = mtid then continue
7:     combineEpochs(Epochs, tid)

```

Algorithm 2 shows the first part of the epoch merge algorithm. First, the epochs

of the master thread `mtid` are assigned to the collection `Epochs`, which will hold all global epochs at the end. Here, all (thread) IDs used in the algorithm are the IDs delivered by corresponding the OMPT callback. Regarding to the OpenMP specification the master thread of a team has always the `mtid=0`. However, OMPT as defined in TR2 uses the value 0 to indicate an invalid thread. Thus, the value of master thread is not known implicitly and is required as input parameter. Second, the algorithm iterates over all worker threads and aggregates each epoch into the corresponding epoch of the collection `Epochs`. This is necessary because all collected information is on a per thread basis up to now. After the merging algorithm is complete, this combined data represents all multi-threaded epochs as defined in the model.

Algorithm 3 Aggregation of epochs

```

1: procedure COMBINEEPOCHS(Epochs, tid)
2:   McurEpoch  $\leftarrow$  Epochs.GetFirst()
3:    $\triangleright$  Go through all epochs of the worker thread
4:   for all WcurEpoch in GetEpochs(tid) do
5:     MsearchEpoch  $\leftarrow$  McurEpoch
6:      $\triangleright$  Search a matching epoch on the master thread
7:     while MsearchEpoch.parallelID  $\neq$  WcurEpoch.parallelID  $\wedge$ 
       MsearchEpoch  $\neq$  Epochs.GetEnd() do
8:       MsearchEpoch  $\leftarrow$  MsearchEpoch.GetNext()
9:      $\triangleright$  Found matching epoch
10:    if MsearchEpoch  $\neq$  Epochs.GetEnd() then
11:      McurEpoch  $\leftarrow$  MsearchEpoch
12:      McurEpoch.addPeriods(WcurEpoch)
13:      McurEpoch  $\leftarrow$  McurEpoch.GetNext()
14:      WcurEpoch  $\leftarrow$  WcurEpoch.GetNext()
15:     $\triangleright$  No matching epoch found
16:    else if WcurEpoch.isMaster() = false then
17:       $\triangleright$  Add as new epoch
18:      Epochs.addAfterSameParent(WcurEpoch)
19:      WcurEpoch  $\leftarrow$  WcurEpoch.GetNext()
20:    else
21:      WcurEpoch  $\leftarrow$  WcurEpoch.GetNext()
return

```

The second part of the merging step is described in Algorithm 3 in detail. Basically, it shows how the aggregation of the per thread epochs into the collection `Epochs` is done. For that the algorithm iterates all epochs of the worker thread in order to find a matching epoch in the collection on the basis that it has the same *parallelID*. In case a matching epoch was found, all periods of the epoch of the worker thread are added to this epoch. If no matching epoch was found, this means that the current epoch is either a master epoch or a (nested) parallel epoch, in which

4. Epoch Model for OpenMP

the most outer master thread did not contributed to. In the first case (line 20), this epoch is just skipped. In the latter case (line 16) the epoch is added to the collection as new entry directly after the last epoch that has the same parent id. This only happens for nested epochs where the outer master thread did not participate and thus is missing in the initial collection. This algorithm processes master, parallel and nested epochs in the same way except for this difference.

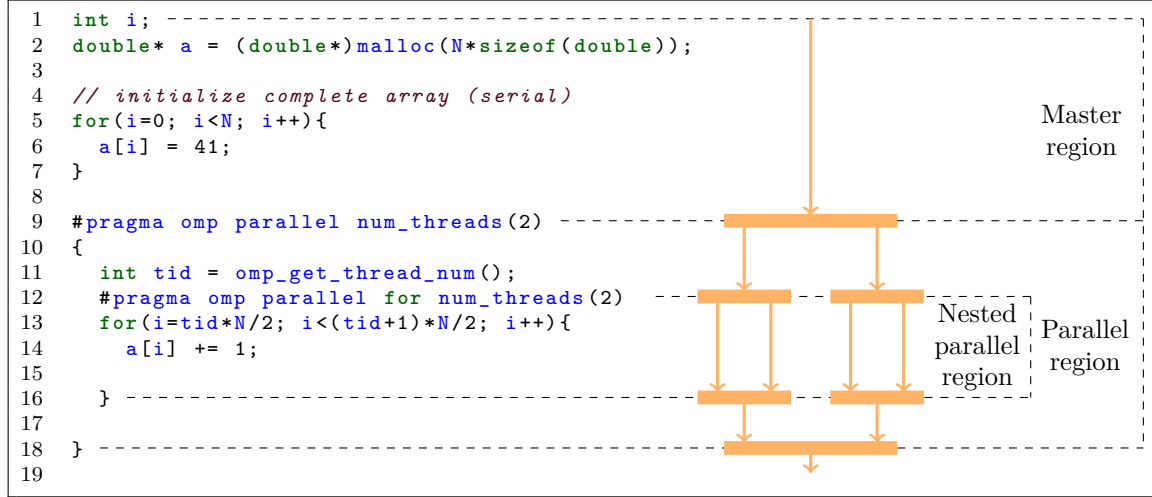
Although the algorithm has an inner and an outer loop the complexity for each thread is $\mathcal{O}(n)$ in most cases (where n is the amount of epochs). For OpenMP programs without nested parallelism this assertion always holds. The reason is that the thread-local epochs are already ordered, because they are stored in the order they occur during runtime. Thus, the assignment of a local epoch to the global one is a simple element-wise merge. The complexity for OpenMP programs with nested parallelism is still $\mathcal{O}(n)$ in most cases, because only for new nested epochs (i.e., those local nested epochs which not have been added to *Epochs* yet) one has to search to end of the collection and add this epoch after that epoch with the same parent parallel ID (line 18). Thus, the assignment for the next threads is again an element-wise merge. However, the complexity is $\mathcal{O}(n^2)$ if $t_o \gg t_i$, where t_o the number of outer threads and t_i is the number of inner threads, because in that case we have to search to the end of the collection up to $n/2$ times in the inner loop (with $t_i = 2$). The complexity also increases with the amount of nesting levels.

4.4. Evaluation of the Concept

In this section, the concept of the presented epoch model is demonstrated. For the correctness checking two different technologies are used: OMPT and binary instrumentation (refer to Section 3 and Section 5.1.2 for details).

In general, a standard-compliant OMPT implementation does not require to provide all of the specified runtime events, but only those which are defined as mandatory. However, the presented epoch model assumes an OpenMP runtime that implements those events specified as optional or at least those events causing transition relations of the PDA (refer to Figure 4.1). Although the LLVM/Intel OpenMP runtime² implements most of events defined in the technical report (*TR2*), this requirement does not fully hold for current implementation, because the target-related events are missing at the moment. For that reason the extended version presented in Chapter 3.3 is used for the evaluation. As mentioned in this chapter the proposed extension is part of the technical report 4 (*TR4*), published by the OpenMP ARB. The goal of this technical report is to be part of the OpenMP 5.0 specification. As soon as the technical report 4 (*TR4*) is part of the standard, parts of this extension can be integrated into the official LLVM runtime. Since the runtime is supported by clang, GNU and Intel compilers, the implementation of the epoch model can

²<http://openmp.llvm.org>



Listing 4.1: A simple OpenMP program with nested parallelism and serial initialization. The arrows on the right-hand side show the amount of active threads for each line of the code.

be used for a broad range of platforms. For evaluation a tool was build, which registers all of the available OMPT events. Furthermore, a memory access trace is integrated which enables the detection of defects caused by unsynchronized data accesses (e.g., data races). Since the details of the memory trace are not important for the evaluation of the epoch model, they are described later in Section 5.1.2. The tool implements the PDA as specified in the previous sections.

Epoch Generation: In the following it is illustrated how the defined (and implemented) PDA computes on a given input string and generates the thread-local epochs. This visualizes the computation of the PDA and shows that the concept works in general. Listing 4.1 shows a simple OpenMP program with two nested loops. Furthermore, it is depicted how the fork-join model works in this case on the right-hand side. With every new (nested) parallel region, further thread will be forked and form a new (inner) thread team. The end of the corresponding parallel scope will join the threads again. In the code all elements of a double precision floating point array of length N are initialized in serial in the master region. The array is divided into two domains, proceeded by two threads in the scope of outer parallel loop (marked as parallel region in the figure). Each outer thread accesses its domain in parallel by using a combined worksharing construct (i.e., the `parallel for` in line 12). In this nested parallel region, each element of the array is incremented by one. In order to avoid the serialization of the nested parallel region, the environment variable `OMP_NESTED` is set to true. Otherwise a compiler or runtime implementation has the freedom to use non-nested parallelism. Thus, an execution of this program will spawn four threads in sum, since also the inner nested loop uses two threads.

4. Epoch Model for OpenMP

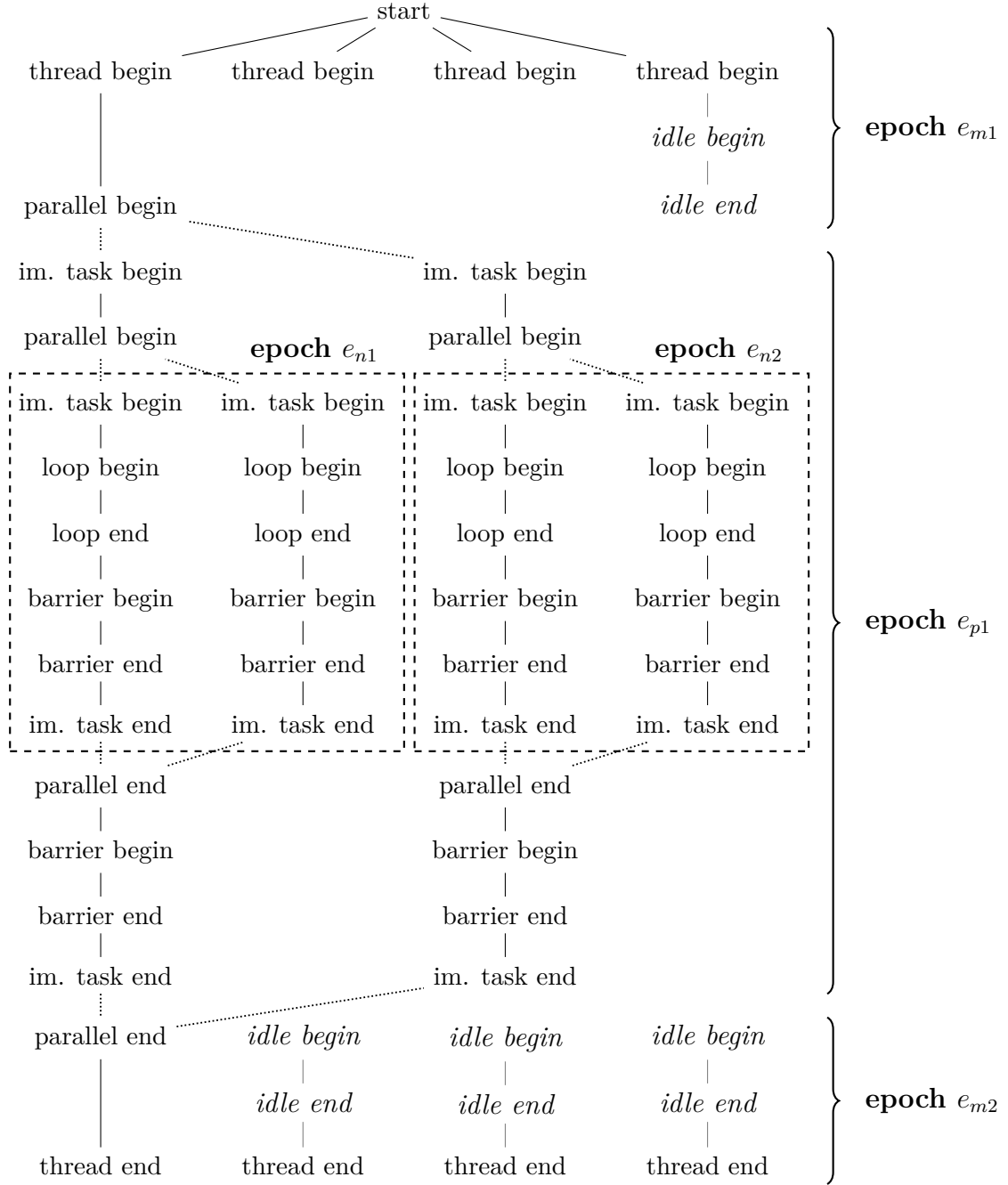


Figure 4.2.: Periods and epochs generated for a nested parallel OpenMP program (refer to Listing 4.1). Dotted edges mark a fork or a join. Italic nodes/gray edges mark idle events which can be assigned to different epochs as well. Dashed boxes mark the generated nested epochs.

Figure 4.2 visualizes the complete result delivered by the tool described above, executed with the simple nested OpenMP program. This includes the thread-local epoch generation and the final merging step. Here, the input string given by the execution order of the program is transferred into to a partial order. The order shown in the figure is not necessarily the order observed during the runtime. The main reason for that is, that OMPT does not provide any scoping information for some events. For instance, a runtime has the freedom to fork or destroy a thread at any time (at least when the thread is not part of an active thread team) and thus a *thread_begin* or a *thread_end* cannot be assigned to a specific parallel region. The same applies for *idle* events. Since the model intends to be as generic as possible these events are not just omitted, but assigned to current master epoch or the previous master epoch, if observed within a parallel region. However, this is not a limitation for correctness checking tools in general, because the code executed between the corresponding *begin* and the *end* is not within the user code, but within the OpenMP runtime. Beside the application of the model in a correctness checking tool, which validates a given OpenMP application, other potential use cases exists. This can be either the validation of the OpenMP specification or a certain implementation of the standard. Here, the OpenMP language committee or runtime implementers can benefit from the well-defined model. However, the fact that for instance *idle* events are directly assigned to the previous master epoch affects this kind of correctness analysis. The detail of this will be discussed in Section 5.1.4.

The result shown in the Figure 4.2 might differ between different runs, because some events might or might not occur – depending on the timing behavior of the current execution. For instance, in the first master epoch e_{m1} only the fourth thread starts to idle before it executes a work package. The first event which can be assigned to a parallel epoch is the *implicit_task_begin* event, because OMPT delivers an ID for the parent task, which allows a unique mapping. The thread (i.e., the *thread_begin* event) might be spawned before or after encountering the parallel region – OMPT does not deliver any knowledge about that.

					X	X	Y	Y	Y	Y	Y	Y						
Γ	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X
Q	q_m	q_m	q_m	q_{h2}	q_p	q_p	q_p	q_p	q_p	q_p	q_{h3}	q_{h1}	q_p	q_p	q_{h3}	q_{h1}	q_m	q_m
λ					e_m								e_n				e_p	
Σ	σ_s	σ_{th_b}	σ_{p_b}		σ_{ta_b}	σ_{p_b}	σ_{ta_b}	σ_{l_b}	σ_{l_e}	σ_{b_b}	σ_{b_e}	σ_{ta_e}	σ_{p_e}	σ_{b_b}	σ_{b_e}	σ_{ta_e}	σ_{p_e}	σ_{th_e}
					e_{n1}													
	e_{m1}				e_{p1}												e_{m2}	

Table 4.2.: PDA computation of the master thread for the nested parallelism example (refer to Table 4.1 for the input symbols).

As define above, the first step for the generation of the epochs is to compute the thread-local epochs by using the PDA. Table 4.2 illustrates every compute step

4. Epoch Model for OpenMP

caused be a new input symbol (i.e., an OMPT event) for the outer master thread. The computation for all worker or nested master threads is done correspondingly. The input sequence can be seen in the row marked with Σ . The row marked by a Q shows the current state of the PDA, which is switched by reading the next input symbol $\sigma \in \Sigma$. As mentioned before, the decision for the transition relation does not only depend on the read input symbol, but also on the symbol on top of the stack, which is shown in row(s) marked by a Γ . These rows on top also illustrate the development of the stack during the execution of the program. The generated output symbol is given in the row marked by a λ . The bottom row marks the input sequence belonging to the generated thread-local epoch. For instance, reading the first σ_{th_b} with a X on top of the stack neither causes a state switch nor a change on the top of stack. In contrast to that, the next input symbol σ_{p_b} (i.e., the outer parallel region) with a X on top of the stack causes a transition relation $q_m \rightarrow q_{h2}$. However, the stack is not changed here as well. With the next input symbol σ_{ta_b} another X is pushed on top of the stack in order to mark the current nesting level. Furthermore, an output symbol e_m is emitted in order to complete the first master epoch. Here, the impact of the lazy epoch generation can be seen, because the input symbol causing the completion of an epoch already belongs to the new epoch. As was discussed this is necessary to decide whether the input symbol completes an epoch or not. For this decision the next input symbol has to be read. The transition relation is $q_{h2} \rightarrow q_p$, because the thread is in a parallel region now.

The generation of the first parallel epoch e_{p1} is interrupted by the nested parallel region. By reading the next σ_{ta_b} a Y is pushed on top of the stack in order to mark this inner parallel region. The state remains in q_p , because any OpenMP construct which can be use in the outer parallel region can also be used in the inner parallel region and thus only the depth of the stack changes with every further σ_{ta_b} . The following three input symbols σ_{l_b} , σ_{l_e} and σ_{b_b} do neither cause a state switch nor a change on the top of the stack, because this sequence of symbols belongs to the nested epoch e_{n1} . The next input symbol σ_{b_e} also belongs to the nested epoch, but causes a state switch $q_p \rightarrow q_{h3}$, because the end of a barrier always initiates the end of the current epoch. Since the next symbol σ_{ta_e} marks the end of the parallel region, the PDA switches to q_{h1} . The next symbol always causes a the completion of the current epoch. The decision whether it is a parallel or a nested epoch is basically made by the symbol on top of the stack. Here, it can be seen that the Y on top of the stack is important in order to switch back to the parallel state ($q_{h1} \rightarrow q_p$). The output symbol e_{n1} is emitted and the Y is removed from the top of the stack. The fact that a X is on the top of the stack now means that the following collected input symbols again belong to the parallel epoch e_{p1} . The end of that epoch is initiated in the same way as for e_{n1} by reading a σ_{b_e} for the end of the barrier ($q_p \rightarrow q_{h3}$) and a σ_{ta_e} for the end of the implicit task ($q_{h3} \rightarrow q_{h1}$). Since there is a X on top of the stack (and not a Y as before), reading the next input symbol σ_{ta_e} causes a transition relation $q_{h1} \rightarrow q_m$ and the emission of e_p for the completion of the parallel epoch e_{p1} .

The PDA computation for the other threads has to be done correspondingly and is not further discussed in this thesis. After applying the merging algorithm (refer to Section 4.3) to all computed local epochs, the result for the global epochs (as depicted in Figure 4.2) is the following:

$$\Omega = \{e_{m1}, e_{p1}, e_{n1}, e_{n2}, e_{m2}\}, \quad (4.2)$$

where the event sequences are given by³

$$\begin{aligned} e_{m1} = \{ & \sigma_s(0), \sigma_{th_b}(0), \sigma_{p_b}(0), \\ & \sigma_s(2), \sigma_{th_b}(2), \\ & \sigma_s(3), \sigma_{th_b}(3), \\ & \sigma_s(4), \sigma_{th_b}(4), \sigma_{i_b}(4), \sigma_{i_e}(4) \}, \end{aligned} \quad (4.3a)$$

$$\begin{aligned} e_{p1} = \{ & \sigma_{ta_b}(0), \sigma_{p_b}(0), \sigma_{p_e}(0), \sigma_{b_b}(0), \sigma_{b_e}(0), \sigma_{ta_e}(0) \\ & \sigma_{ta_b}(2), \sigma_{p_b}(2), \sigma_{p_e}(2), \sigma_{b_b}(2), \sigma_{b_e}(2), \sigma_{ta_e}(2) \}, \end{aligned} \quad (4.3b)$$

$$\begin{aligned} e_{n1} = \{ & \sigma_{ta_b}(0), \sigma_{l_b}(0), \sigma_{l_e}(0), \sigma_{b_b}(0), \sigma_{b_e}(0), \sigma_{ta_e}(0), \\ & \sigma_{ta_b}(3), \sigma_{l_b}(3), \sigma_{l_e}(3), \sigma_{b_b}(3), \sigma_{b_e}(3), \sigma_{ta_e}(3) \}, \end{aligned} \quad (4.3c)$$

$$\begin{aligned} e_{n2} = \{ & \sigma_{ta_b}(2), \sigma_{l_b}(2), \sigma_{l_e}(2), \sigma_{b_b}(2), \sigma_{b_e}(2), \sigma_{ta_e}(2), \\ & \sigma_{ta_b}(4), \sigma_{l_b}(4), \sigma_{l_e}(4), \sigma_{b_b}(4), \sigma_{b_e}(4), \sigma_{ta_e}(4) \}, \end{aligned} \quad (4.3d)$$

$$\begin{aligned} e_{m2} = \{ & \sigma_{p_e}(0), \sigma_{th_e}(0), \\ & \sigma_{i_b}(2), \sigma_{i_e}(2), \sigma_{th_e}(2), \\ & \sigma_{i_b}(3), \sigma_{i_e}(3), \sigma_{th_e}(3), \\ & \sigma_{i_b}(4), \sigma_{i_e}(4), \sigma_{th_e}(4) \}. \end{aligned} \quad (4.3e)$$

Here the following happens before relation holds:

$$e_{m1} \prec e_{p1} \prec e_{m2}. \quad (4.4)$$

Furthermore, the result implies that the following holds for each epoch e_x :

$$\forall \sigma(i), \sigma(j) \in e_x, i \neq j : \sigma(i) \not\prec \sigma(j). \quad (4.5)$$

This means that all events of the same epoch which are executed by different threads are concurrent and thus all the memory accesses within the same epoch are

³The Intel/LLVM OpenMP runtime uses an additional management thread with ID 1. Thus only thread 0,2,3 and 4 executed user code.

4. Epoch Model for OpenMP

potential data races. Furthermore, for tool developers it is especially important to remind that the following holds as well:

$$e_{n1} \not\prec e_{n2}. \quad (4.6)$$

Since no partial order of the nested epochs can be determined a correctness checking tool has to analyze the epochs recursively beginning with the inner most epoch. If no defects can be detected in the inner most epoch the events can be merged to the next outer epoch, such that in this example e_{n1} and e_{n2} do not exist anymore and their events are treat as they were executed by the corresponding master thread (the merged symbols are in bold font):

$$\begin{aligned} e'_{p1} = \{ & \sigma_{ta_b}(0), \sigma_{p_b}(0), \\ & \quad \sigma_{ta_b}(0), \sigma_{l_b}(0), \sigma_{l_e}(0), \sigma_{b_b}(0), \sigma_{b_e}(0), \sigma_{ta_e}(0), \\ & \quad \sigma_{ta_b}(0), \sigma_{l_b}(0), \sigma_{l_e}(0), \sigma_{b_b}(0), \sigma_{b_e}(0), \sigma_{ta_e}(0), \\ & \sigma_{p_e}(0), \sigma_{b_b}(0), \sigma_{b_e}(0), \sigma_{ta_e}(0), \\ & \sigma_{ta_b}(2), \sigma_{p_b}(2), \\ & \quad \sigma_{ta_b}(2), \sigma_{l_b}(2), \sigma_{l_e}(2), \sigma_{b_b}(2), \sigma_{b_e}(2), \sigma_{ta_e}(2), \\ & \quad \sigma_{ta_b}(2), \sigma_{l_b}(2), \sigma_{l_e}(2), \sigma_{b_b}(2), \sigma_{b_e}(2), \sigma_{ta_e}(2), \\ & \sigma_{p_e}(2), \sigma_{b_b}(2), \sigma_{b_e}(2), \sigma_{ta_e}(2) \}. \end{aligned} \quad (4.7)$$

In the next step the merged epoch e'_{p1} has to be validated. This approach can be used for any nesting level, thus that for instance a data race detection can be done for any nested parallel OpenMP program. Further evaluations and use cases for the application of the epoch model will be discussed in Section 5.1.3.

4.5. Extension for Task-based Programs

Since version 3.1 OpenMP can be used as a task-centric programming paradigm. For many non loop-centric, recursive or unbalanced problems this can improve the expressiveness of OpenMP and thus the performance of an application. However, also with explicit tasks defects such as race conditions are possible if the same data is accessed by different threads (and at least written once) without correct synchronization. Thus, for the extension of the epoch model it is at first necessary to discuss those different synchronization points. This discussion can be made by the differentiation of two task generation patterns: recursive and non-recursive.

Non-recursive Task Generation. The simplest cases are algorithms which use the tasking model in order to improve the load balancing behavior without generating any child tasks. In this case the only possible synchronization construct is a **barrier**, which might be implicit or explicit. Since, barriers are already part of the given

epoch model, it is directly applicable for such task-based programs. Within the class of non-recursive task generation one can differentiate between two different patterns:

- (i) single-producer, multiple-executors
- (ii) parallel-producer, multiple-executors

The details of the different patterns and the effects on the performance of an application have been discussed in Section 2.5. The generated epochs look different depending on the used pattern. For instance no input symbols for the `single` construct will be read for the parallel-producer pattern. Furthermore, the task-scheduling might be completely different (in fact this is the reason for the performance benefit of (ii)). However, although the generated epochs look different, for a correctness checking tool the used pattern is irrelevant. The fact that the kernel of the algorithm does not change by applying one of the patterns leads to the execution of the same scheduling points and thus to the same epoch structure. With that the happens-before relations can be determined independent of the used pattern. Since OpenMP barriers are already taken into account in the PDA computation, the model fully supports both patterns without any further modifications for non-recursive task generation.

Since Definition 4.2 and Definition 4.3 assume that two events or rather two periods are executed by different tasks (and not threads), a correctness checking tool can also detect potential data races – even if all tasks were scheduled to the same thread. This is one of the strengths of the model. However, for correctness analysis like data-race detections this approach introduces also the risk for false positives for those OpenMP programs using `threadprivate` data within explicit tasks. This is, because OpenMP defines that each thread has its own copy of all variables declared as `threadprivate`. Thus, the same memory is accessed when the two tasks are executed by the same thread. Since no happens-before relation is determined between two periods (i.e., the tasks), a correctness checking tool would report this as a race condition. However, if the tasks are executed by different threads, the accessed memory is different, because the thread private copy of the variable is used, such that a tool would not report a defect.

In version 4.5 the `taskloop` construct was integrated into the OpenMP specification. This construct allows generating explicit tasks for iterations of one or more associated loops. The behavior is the same as for the parallel-producer pattern and thus the generated epochs will look similar. Here the difference is that `taskloop`-based programs use one implicit barrier less. Thus, a corresponding epoch generation will also have one parallel epoch less. Since no user code is executed in between, this does not limit the applicability of the model on `taskloop`-based programs.

For the evaluation of the epoch model for case (i), the simple example of the single-producer, multiple-executors pattern from Listing 2.5 is used as task-based OpenMP program. Here, one thread generates all tasks which can be executed by

4. Epoch Model for OpenMP

all available threads. The first synchronization point is the implicit barrier at the end of the **single** region. Furthermore, there is an implicit barrier at the end of the **parallel** region. In both regions the task can be executed, where the runtime has to guarantee that all generated task are completed after the end of the parallel region. An execution of the program with a $N_TH = 4$ threads and $N_TA = 100$ tasks leads to following set of epochs⁴:

$$\Omega = \{e_{m1}, e_{p1}, e_{p2}, e_{m2}\}, \quad (4.8)$$

where the event sequence is given by (refer to Table 4.1 event names)

$$\begin{aligned} e_{m1} = \{ & \sigma_s(0), \sigma_{th_b}(0), \sigma_{p_b}(0), \\ & \sigma_s(2), \sigma_{th_b}(2), \\ & \sigma_s(3), \sigma_{th_b}(3), \\ & \sigma_s(4), \sigma_{th_b}(4) \}, \end{aligned} \quad (4.9a)$$

$$\begin{aligned} e_{p1} = \{ & \sigma_{ta_b}(0), \sigma_{si_b}(0), \sigma_{si_e}(0), \sigma_{b_b}(0), \sigma_{wb_b}(0), \sigma_{wb_e}(0), \sigma_{b_e}(0), \\ & \sigma_{ta_b}(2), \sigma_{so_b}(2), \sigma_{so_e}(2), \sigma_{b_b}(2), \sigma_{ts}(2), \dots, \sigma_{ts}(2), \sigma_{b_e}(2), \\ & \sigma_{ta_b}(3), \sigma_{so_b}(3), \sigma_{so_e}(3), \sigma_{b_b}(3), \sigma_{wb_b}(3), \sigma_{wb_e}(3), \sigma_{b_e}(3), \\ & \sigma_{ta_b}(4), \sigma_{so_b}(4), \sigma_{so_e}(4), \sigma_{b_b}(4), \sigma_{ts}(4), \dots, \sigma_{ts}(4), \sigma_{b_e}(4) \}, \end{aligned} \quad (4.9b)$$

$$\begin{aligned} e_{p2} = \{ & \sigma_{b_b}(0), \sigma_{wb_b}(0), \sigma_{wb_e}(0), \sigma_{b_e}(0), \sigma_{ta_e}(0), \\ & \sigma_{b_b}(2), \sigma_{b_e}(2), \sigma_{ta_e}(2), \\ & \sigma_{b_b}(3), \sigma_{b_e}(3), \sigma_{ta_e}(3), \\ & \sigma_{b_b}(4), \sigma_{b_e}(4), \sigma_{ta_e}(4) \}, \end{aligned} \quad (4.9c)$$

$$\begin{aligned} e_{m2} = \{ & \sigma_{p_e}(0), \sigma_{th_e}(0), \\ & \sigma_{i_b}(2), \sigma_{i_e}(2), \sigma_{th_e}(2), \\ & \sigma_{i_b}(3), \sigma_{i_e}(3), \sigma_{th_e}(3), \\ & \sigma_{i_b}(4), \sigma_{i_e}(4), \sigma_{th_e}(4) \}. \end{aligned} \quad (4.9d)$$

In the first master epoch e_{m1} the parallel region is forked by thread 0. Since the program was compiled with the Intel compiler the threads 0,2,3,4 are OpenMP

⁴In the used version of the LLVM OpenMP runtime implementation the events **task_begin** and **task_end** were triggered as well. They are omitted here, because they neither exist anymore in revised technical report on OMPT (*rTR2*) nor in the technical report 4(*TR4*). The reason for that is that a **task_switch** event indicates the begin and end of a task implicitly. Furthermore, the **task_create** event is missing in the used implementation. Although the encountering of a task construct is an OpenMP scheduling point, none of these deviations from the specification/implementation are of conceptual importance for the epoch model.

threads, while the management thread 1 does not execute any user code and thus is not part of the OpenMP team. In the parallel epoch e_{p1} it can be seen that all tasks are executed by thread 2 and 4, where “...” denotes the execution of multiple tasks. The tasks are executed between the *barrier_begin* and the *barrier_end* event of the first (implicit) barrier of the *single* construct (refer to Listing 2.5), because this is a task scheduling point in OpenMP. In this case neither thread 0 nor 3 did execute any of the generated explicit tasks. This only depends on the used task scheduling strategy and the runtime behavior, of course. The second parallel epoch e_{p2} basically includes the execution of the second (implicit) barrier for the end of the parallel region and the *implicit_task_end* events. Since all tasks are executed in epoch e_{p1} , a correctness checking tool is able to detect defects like data races within this epoch, because no reliable happens-before relation between the tasks can be determined. As mentioned above, a correctness checking tool can also detect potential data races even if all tasks were scheduled to the same thread here. For instance, if the same data is accessed (and at least written once) by different tasks in the function `some_computation()` a (potential) data race exists. Here, one can also see the risk for false positives for those OpenMP programs using `threadprivate` data within explicit tasks (as mentioned above). If two tasks are executed by the same thread also the same memory is accessed. Since no happens-before relation is determined between the two periods (i.e., the tasks), a correctness checking tool would report this as a race condition, although each thread has its own copy of all variables declared as `threadprivate`.

For the evaluation of case (ii), Listing 2.6 shows the application of the parallel-producer, multiple-executors pattern to the given example. As expected this yields an equivalent result for the epochs. Nevertheless, e_{p1} looks slightly different now:

$$\begin{aligned}
 e'_{p1} = \{ & \sigma_{ta_b}(0), \sigma_{l_b}(0), \sigma_{l_e}(0), \sigma_{b_b}(0), \sigma_{ts}(0), \dots, \sigma_{ts}(0), \sigma_{b_e}(0), \\
 & \sigma_{ta_b}(2), \sigma_{l_b}(2), \sigma_{l_e}(2), \sigma_{b_b}(2), \sigma_{ts}(2), \dots, \sigma_{ts}(2), \sigma_{b_e}(2), \\
 & \sigma_{ta_b}(3), \sigma_{l_b}(3), \sigma_{l_e}(3), \sigma_{b_b}(3), \sigma_{ts}(3), \dots, \sigma_{ts}(3), \sigma_{b_e}(3), \\
 & \sigma_{ta_b}(4), \sigma_{l_b}(4), \sigma_{l_e}(4), \sigma_{b_b}(4), \sigma_{ts}(4), \dots, \sigma_{ts}(4), \sigma_{b_e}(4) \}, \quad (4.10)
 \end{aligned}$$

The absence of the `single` construct and the additional `loop` construct lead to a replacement of the $\sigma_{si_ \{b,e\}}$ and $\sigma_{so_ \{b,e\}}$ input symbols through corresponding $\sigma_{l_ \{b,e\}}$. Furthermore, the application of the parallel-producer pattern clearly shows that now all threads execute tasks. The reason for that is the use of thread-local queues in the LLVM OpenMP runtime and thus less task-stealing is necessary. As claimed above, this shows that the used pattern is irrelevant for a correctness checking tool, because the happens-before relations can be determined correspondingly. These examples ((i) and (ii)) show that the epoch model works as expected, also for those task-based OpenMP programs using only barriers to synchronize (i.e., non-recursive algorithms).

One example for the application of OpenMP task are algorithms which require a

4. Epoch Model for OpenMP

```
1  int fib(int i)
2  {
3      int x, y;
4      if (i < 2)
5          return i;
6
7      #pragma omp task shared(x)
8      {
9          x = fib(i - 1);
10     }
11
12     #pragma omp task shared(y)
13     {
14         y = fib(i - 2);
15     }
16
17     #pragma omp taskwait
18
19     return x+y;
20 }
```

Listing 4.2: A naive computation of the Fibonacci sequence with OpenMP tasks.

dynamic load balancing. The examples used in this section are the task-based versions of the CG method presented in Section 2.3, which apply both tasking patterns. Since they do neither use `threadprivate` nor tasks are generated within other tasks, all OpenMP-based versions of the benchmark can be analyzed without any limitations. This shows the applicability of the model to real-world applications, because CG kernels are one of the most compute-intensive parts of many PDE solvers.

Recursive Task Generation. Task synchronization with OpenMP barriers presupposes that the task hierarchy executed by current thread team is flat. However, this assumption is not valid in general, because tasks might create further child tasks and synchronize those with the `taskwait` construct. Furthermore, task dependencies exist since version 4.5. Typical use cases for hierarchical tasking programs are recursive algorithms, where the result of the current task depends on the results of the child tasks. Listing 4.2 shows a naive⁵ recursive computation of the Fibonacci sequence, which is characterized by the fact that every number is the sum of the two preceding ones:

$$F_i = F_{i-1} + F_{i-2}, \text{ where } F_1 = 1, F_2 = 2, i \in \mathbb{N}. \quad (4.11)$$

This formula represents a typical use case for task-based approaches: recursive algorithms. Figure 4.3 shows the task tree for the computation of F_5 with three threads. The execution was done with LLVM OpenMP runtime. Every node in the tree is marked with the observed task ID delivered by OMPT. The root of the tree

⁵In terms of performance, this is neither the best algorithm, nor the best OpenMP implementation to compute the Fibonacci sequence. Nevertheless, the example was chosen in order to keep it as illustrative as possible.

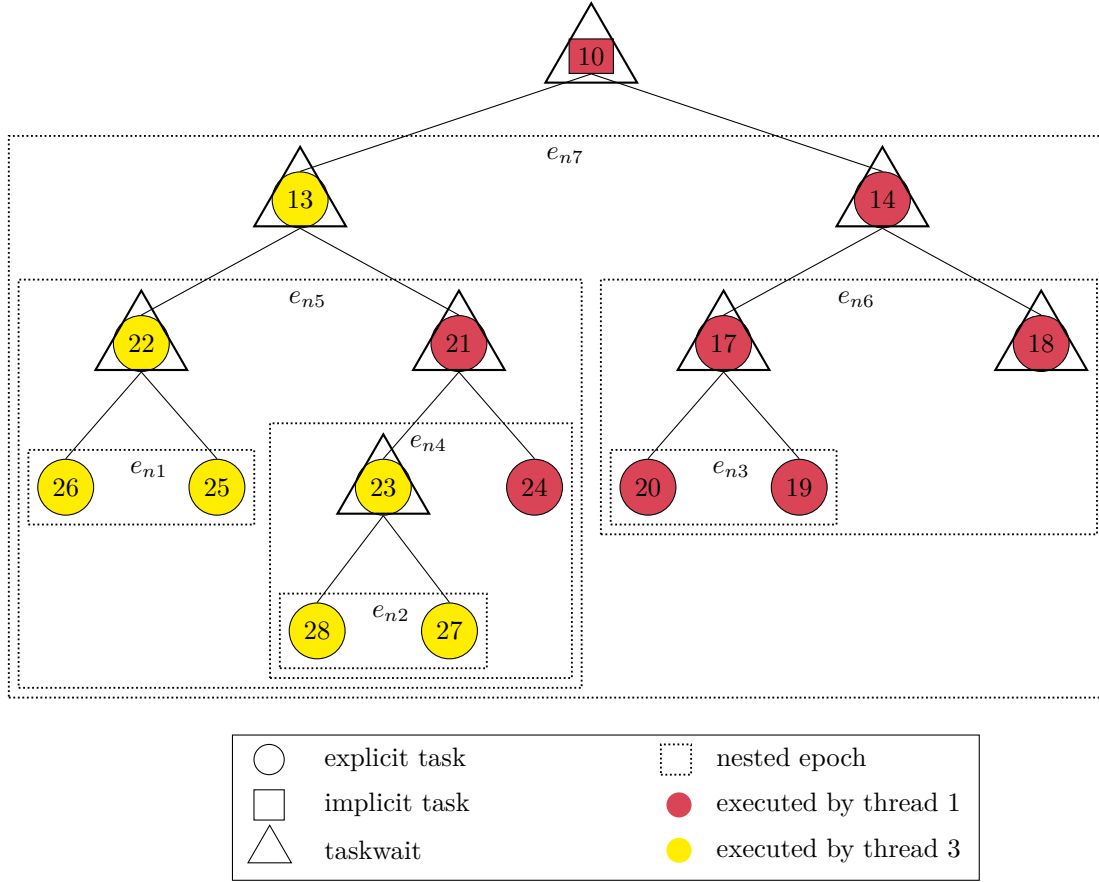


Figure 4.3.: A possible task tree for the computation of the Fibonacci sequence with $i = 5$. The generated (nested) epochs are shown within the dashed lines. The execution/scheduling of the tasks depends on the runtime behavior and might differ between multiple runs.

is an implicit task with ID 10 calling the `fib()` routine from the main routine. All child nodes are explicit tasks, which are synchronized by a `taskwait`. Although the program was started with three OpenMP threads, all tasks were executed by thread 1 and 3, while thread 2 did not execute any task. In OpenMP a `taskwait` is a scheduling point, which means that child tasks can be executed between the `task_wait_begin` and the `task_wait_end` event of the parent. In order to determine which tasks potentially are executed in parallel and which are not the concept of nested epochs can be applied. For that corresponding transition relations have to be added to the PDA, such that with every input symbol σ_{tw_b} a Y is pushed on top of the stack and with every σ_{tw_e} the epoch is generated and the top of the stack is removed again. Furthermore, the epoch merging algorithm has to be extended, such that the thread local epochs are assigned to a global one generated by the corresponding parent. A data race detection for instance has to be done from the inner most nested epoch to the most outer one, where the task executed in inner one are merged to the outer one. However, modeling this in two steps – thread local

4. Epoch Model for OpenMP

epoch generation and epoch merging – leads to fast increasing the state space when done as postmortem analysis. The reason for is, that the amount of nested epochs increases exponentially with the depth of the tree, such that the complexity is $\mathcal{O}(2^i)$, which correspond to the time complexity of the used algorithm itself. Furthermore, not only the algorithm and the determination of the happens-before relations is expensive, but also the epoch-based correctness validation.

In the previous section an OpenMP program with nested parallelism was analyzed (Section 4.4). The complexity remains linear in most real-world applications, because the depth of the parallel nesting is low on the one hand and the structure of the generated epochs is regular on the other hand (refer to Figure 4.2). This is especially important for the presented merge algorithm (Listing 3). The regularity of the epochs leads to clear happens-before relation. Unfortunately, neither the low nesting depth of the epochs nor the regularity is given for typical recursive task-based algorithms. The tasking example (Figure 4.3) shows that for many epochs no statement about their happens-before relations can be made and thus are concurrent epochs by definition. Although the figure shows for instance that $e_{n2} \prec e_{n4}$ or that $e_{n4} \prec e_{n5}$ and thus also $e_{n2} \prec e_{n5}$ (transitivity), no partial order can be determined between the leafs of the tree (e.g., $e_{n1} \not\prec e_{n2}$, $e_{n1} \not\prec e_{n4}$, $e_{n1} \not\prec e_{n3}$ or $e_{n1} \not\prec e_{n6}$). An optimization for a correctness checking algorithm could be to omit those epochs which were executed by the same thread (e.g., e_{n1} , e_{n2} or e_{n6}). However, this contradicts the given definition of the happens-before relations, because they are defined on a per-task basis (and not per thread). Furthermore, this increases the inaccuracy, because the knowledge of a potential data race is omitted as well. Last, it limits the universal validity of the epoch model, which is supposed to be a general approach for any kind of correctness checking tool. For instance, if task 25 and 26 in e_{n1} accessing the same data, the access is still unsynchronized (at least for non `threadprivate` data), although no data race occurred for this run. Neither the order of the execution is guaranteed, nor that the tasks are executed by the same thread in a second run of the program. Omitting the epochs which were executed by only one thread would prevent a corresponding analysis.

For a final assessment of the complexity, one has to keep in mind that the reason for this analysis complexity is not the presented epoch model approach, but it is the nature of the used algorithm. The analysis complexity increases with computational complexity and thus does not limit the scalability of the analysis in general. In order to avoid the cost for the epoch merging, it is beneficial to determine the epochs in a more dynamic approach instead of doing this in a postmortem analysis (i.e., on-the-fly). In this case a correctness analysis of the innermost nested epoch can be done as soon as the epoch is completed. After the check in this innermost nested epoch it can directly merged to the next higher level. This leads to a dynamically growing and shrinking epoch tree during the runtime without limiting the strength of the epoch model.

Further Limitations. Besides the limitations already mentioned above (false positives with thread private data, complexity for merge algorithm), two further restrictions exist: task dependencies and untied tasks. Both OpenMP features are not supported by the model. In general, OMPT supports the analysis of task dependencies by defining corresponding events. If the epoch model is used for pure application analysis this task dependencies can be taken into account by adding additional happens-before relation between the *task_begin* event of the first task and the *task_end* event of the second task. Thus, it is known that they must not be concurrently (i.e., in the same epoch), which has to be ensured by the runtime implementation. However, if the epoch model is used for the verification of the runtime implementation this information has to be ignored, because it might be a defect within the runtime. Furthermore, also untied tasks are not explicitly supported by the developed epoch model. OpenMP defines that the execution of untied task might be interrupted at a well-defined synchronization point and may be continued by another thread at another task scheduling point. In the given example (Figure 4.3) this would mean that a task leaf in the task tree can have multiple colors. Although the explicit support is missing, one can still execute and analyze the application with tied task (e.g., by setting the corresponding environment variable). This might influence the performance of the execution, but does not limit the applicability of the model in general.

4.6. Epoch Concept for Further Parallel Programming Paradigms

In the previous sections a generic epoch model for the determination of happens-before relations for OpenMP applications has been discussed in detail. Although the applicability has been shown for the main features of the programming standard (worksharing constructs, tasking, nested parallelism, target device offloading), it is still limited to OpenMP. Since the complete information required to determine the epochs (i.e., the concurrency information) is delivered by the OpenMP runtime system and (especially the input symbols for the PDA) are specific to the OMPT interface, a direct (unmodified) use of the concrete model is not possible (and was never designed for that). However, dividing execution phases into epochs in order to create a basis for correctness checking tools can still be a valid approach for other parallel programming paradigms. The intention of this section is not to present a well-defined model for any of these further parallel paradigms as it was done for OpenMP, but to discuss briefly the transferability of the general epoch concept.

OpenACC: OpenACC is a directive-based paradigm to simplify parallel programming of heterogeneous architectures such as GPUs. It does not intend to express parallelism on the host system directly (although one could develop corresponding implementations). The concept is comparable to target device constructs in OpenMP (refer to [91] for a detailed pattern-based comparison). These host-centric

4. Epoch Model for OpenMP

constructs allow offloading of data and compute-intensive code to an accelerator. Basically, this can be done either with the `kernel` construct, where the parallelization of the code in this scope is done by the compiler, or it can be done with the `parallel` construct, where the parallelism has to be expressed explicitly by the programmer (e.g., by using the `loop` construct). The data offloading can be done by using corresponding clauses on these constructs or by using one of the runtime library routines.

Since the code parts outside the corresponding directives are sequentially executed on the host in pure OpenACC programs, they would correspond to the concept of master epochs introduced in this thesis. As discussed in Section 3.1 OpenACC provides a profiling concept, which is quite similar to OMPT. Therefore, the information required as input for the PDA computation can be used from this callback interface as it was done for the OpenMP epoch model. In order to determine the begin and the end of an epoch it is essential that the interface provides information about the explicit and implicit synchronization. Here, it is important to know that in OpenACC no constructs exists to synchronize within a parallel kernel (like a `barrier` does in OpenMP). This is because of the fact that the hardware properties of common GPUs do not allow a direct global synchronization, but only local synchronization within one *Streaming Multiprocessor* (SM). In order to synchronize globally, the single offloaded kernels need to be synchronized using the host processor. This can be done either done by using a `wait` clause on a `kernel` or `parallel` construct or by using the `wait` directive for any asynchronous offload. To obtain the information about these synchronizations, the events `acc_ev_wait_start` and `acc_ev_wait_end` exist in the tools interface. Furthermore, events are available to determine the begin and the end of a kernel launch. Thus, it is possible to handle every kernel offload as parallel epoch as it is done with OpenMP epoch model. Here, it is important to apply the concept of nested epochs to asynchronous offloads.

Although the memory entities between host and device are separated in OpenACC, the device may share memory with the host. Furthermore, data can be transferred to or from the device implicitly or explicitly. Both data management methods are potentially error-prone and often lead to defective programs. Thus, a method for tracing the data transfers is required in order to detect these kinds of defects. The tools interface included in the OpenACC specification allows to register corresponding callbacks. Since these callbacks deliver information about the host and the device pointer of the transferred data, it is possible to track the data mapping and thus to detected data accesses to outdated data.

The epoch concept is transferable to OpenACC in general. Since the parallel expressiveness is lower compared to OpenMP the realization for the epoch generations is easier. Although the concept of nested epochs is required, a less complex finite state machine can be used instead of the PDA, because offloading within a kernel is not supported and thus only one nesting level exists. However, the question if an

epoch model for OpenACC due to the less complex semantics is as beneficial as for OpenMP is a topic for future investigations.

Cilk Plus: Cilk Plus [45, 77] is a parallel programming paradigm developed by Intel⁶. Similar to OpenMP it extends the base languages C and C++ with constructs to express loop- or task-level parallelism and the fork-join idiom. One of the differences compared to is that Cilk Plus only adds three new keywords to the base language: `cilk_for`, `cilk_spawn` and `cilk_sync`.

The `cilk_for` keyword allows expressing loop-level parallelism in Cilk Plus. This loop-level parallelism is comparable to the `taskloop` construct in OpenMP and not to a worksharing construct. Thus, tasks are generated in order to execute the loop iterations in parallel and for an adequate load balancing task stealing is used within the runtime system. This means that synchronization constructs like barriers in OpenMP does not exists to synchronize within the scope of a parallel loop. For the application of the epoch concept this means that the complete code region within the `cilk_for` scope is one parallel epoch as discussed in Section 2.5 (non-recursive task generation). In general, nested parallelism is possible with Cilk Plus. However, the amount of used threads for each of the parallel regions is not specified by the programmer explicitly, but optimized by the runtime system depending on the available hardware resources. On the one hand this implies a lower parallel expressiveness, on the other hand the code is less error-prone. As a consequence, the concept of nested epochs is not required for nested parallelism in Cilk Plus, because it is not directly influenced by the application programmer and thus is not needed for a correctness checking tool.

```

1  int fib(int i)
2  {
3      int x, y;
4      if (i < 2)
5          return i;
6
7      x = cilk_spawn fib(i - 1);
8      y = cilk_spawn fib(i - 2);
9
10     cilk_sync
11
12     return x+y;
13 }
```

Listing 4.3: A naive computation of the Fibonacci sequence with Cilk Plus.

⁶Cilk Plus has two predecessors: Cilk and Cilk++. The initial version Cilk was developed at the *Massachusetts Institute of Technology* (MIT) [10]. Intel acquired the company Cilk Arts, which developed a modern version of Cilk (i.e., Cilk++) later on and called it Cilk Plus. Although all three version slightly differ (e.g., in the spelling of the keywords), the main concept is the same and does not influence the general discussion here.

4. Epoch Model for OpenMP

Furthermore, Cilk Plus allows expressing parallelism in task-centric way as one can do with the `task` construct in OpenMP. For this the keyword `cilk_spawn` to create a new task and the keyword `cilk_sync` to synchronize the child tasks can be used. Listing 4.3 shows the naive computation of the Fibonacci sequence given in Equation (4.11) in Cilk Plus. Not only the code, but also the execution model is very similar to the OpenMP example used in Listing 4.2. For the transfer of the epoch concept it is important that the happens-before relations are defined on a per-task and not on per-thread basis (refer to Definition 4.2 and 4.3) in order to detect also potential error (e.g., potential data races) and keep the model as generic as possible. Thus, the application of the epoch concept to Cilk Plus is possible in general, but might lack on the overhead for the epoch merge and the amount of generated epochs. For the concrete realization of the model no tools interface exists as it is the case for OpenMP and OpenACC. However, an *Application Binary Interface* (ABI) exists in addition to the language specification. A correctness checking tool can wrap the corresponding runtime calls in order to keep track of the information required to generate the epochs by the PDA computation.

XMP: *XscalableMP* (XMP) [93] is a directive-based parallel programming paradigm using a *Partitioned Global Address Space* (PGAS) approach for distributed memory programming. In PGAS-based languages the memory models designate a shared global address space. This address space is partitioned between the distributed compute nodes. Thus, each node has a local portion of this global memory. Due to the distributed nature of the concept, bigger systems (i.e., complete compute cluster) can be used for the computation (in contrast to previous discussed paradigms OpenMP⁷, OpenACC or Cilk Plus). In contrast to MPI the PGAS memory model allows a transparent memory access instead of explicit message exchange. However, this implicit access makes the approach more error-prone with regard to race conditions. In order to avoid this, explicit or implicit synchronization is required. In XMP the programmer has to choose between two fundamental different memory models: the global-view memory model and the local-view memory model. In this thesis only the first one is discussed with respect to the applicability of the epoch concept.

The global-view memory model looks similar to worksharing concept of OpenMP. The parallelism is expressed through data parallelism and work mapping by adding

⁷With the introduction of the `target` constructs OpenMP can also be used to offload computation to devices which do not share the memory with the host system. In [49] Jacob et al. showed that an OpenMP implementation can support directive-based programming for distributed memory systems by interpreting the other (homogeneous) compute nodes as target devices. Although this is a standard-compliant interpretation, it is neither a common usage of OpenMP nor is the `target` directive designed for a scalable communication over a big amount of computed nodes. Furthermore, the company ScaleMP offers a visualization software (called vSMP Foundation), which provides shared memory on a big compute cluster. Both possibilities are out of scope of this thesis and are not further discussed here.

```

1  define N 10000
2  void vecDot(double* a, double* b, double* ab){
3      #pragma xmp nodes p(*)
4      #pragma xmp template t(0:N-1)
5      #pragma xmp distribute t(block) onto p
6      #pragma xmp align [i] with t(i) :: a, b
7
8      int i;
9      double local;
10     local = 0;
11
12     #pragma xmp loop on t(i) reduction(+:local)
13     for(i=0; i<n; i++){
14         local += a[i]*b[i];
15     }
16     *ab = local;
17 }

```

Listing 4.4: A simple XMP vector-dot kernel using the global-view memory model.

corresponding directives. In order to synchronize within such a construct collective constructs such as a barrier or a reduction clauses exit. Listing 4.4 shows a simple dot product of two vectors implemented in the global-view mode of XMP. In the example the `node` directive declares a named node array of a given size, where the “*” character is used in order to employ all available nodes. Here, a node is defined as an execution entity managed by the runtime system. Each execution entity has its own memory and can execute one or more threads concurrently. The `template` construct declares a dummy array that represents an index space, which can be used to distribute the data onto the node array. In this example the two vectors `a` and `b` are distributed block-wise by using the corresponding directives (line 5,6). The `loop` construct distributes the loop iterations among the nodes, so that the vector-dot product is executed in parallel.

For the evaluation of the applicability of the epoch concept to XMP it is important to consider that all node entities (i.e., the processes) are directly started as it is done with MPI. Thus, the concept of a master epoch does fit here. However, since defects like data races can occur in XMP, dividing the execution into different phases of concurrency is still beneficial in order to analyze the correctness of an application systematically. Here, the different (parallel) epochs are separated by the global-view communication and synchronization constructs. Besides the barrier and reduction synchronization, collective communication constructs like a broadcast to all execution entities or explicit data movement have to be taken into account. For a native XMP program in the global-view mode, nested epochs are not required, because neither nested parallelism, nor target offloading is supported. However, in order to parallelize an application within a node hybrid approaches can be used. For instance, using XMP in conjunction with a shared memory paradigm like OpenMP would allow to first distribute the loop iterations among the nodes and then divide each subset of loop iteration further and distribute the work packages to the OpenMP thread by using a worksharing construct. For the vector-dot example given

4. Epoch Model for OpenMP

in Listing 4.4 this means that a corresponding (combined) directive has to be added to line 12. For the application of the epoch concept the presented epoch model for OpenMP can directly be applied. For that the concept of nested epochs can be used for all OpenMP regions, which allows a recursive algorithm for the correctness checks.

In order to acquire the needed information for the epoch generation no standard-compliant tools interface exists for XMP at the moment⁸. However, the information required can be either acquired by an adequate source-to-source instrumentation or by wrapping the implementation dependent library call. The first method has the disadvantage that information quality is less accurate regarding the runtime behavior, as discussed in Section 3.2. Furthermore, an application needs to be recompiled before the correctness checking can be performed. The fact that the second method is vendor-specific and has to be done for each runtime implementation a tool wants to support, is not a disadvantage at the moment, because only one implementation exists⁹. In future this might be different, of course. For the PDA computation of hybrid programs (e.g., OpenMP and XMP) the input alphabet has to be extended by this information and the transition relations have to be changed correspondingly. Given that the epoch concept can be applied to XMP (or even hybrid approaches). Although further PGAS languages exists, XMP was chosen exemplary, because the directive-based approach allows a direct comparison to OpenMP.

Summary: This section briefly discusses the applicability of the epoch concept to further parallel programming paradigms like OpenACC, *Threading Building Blocks* (TBB), Cilk Plus and XMP. Although the present OpenMP epoch model is not directly applicable to any of these paradigms, the general epoch concept can be used in order to provide a basis for any kind of correctness checking tool. However, the overview given here has no claim to completeness. The concrete realization and detailed discussion of the transfer of the concept is out of scope of this thesis.

4.7. Summary and Conclusion

The increasing complexity of parallel programming paradigms leads to more powerful opportunities to express parallelism on the one hand and to more error-prone programs on the other hand. In order to overcome the latter one, powerful and scalable solutions for the validation of the correctness of an application are required as well. In this chapter the (to the best of my knowledge) first generic and formal OMPT-based epoch model for OpenMP was defined in order to provide a basis for

⁸In the project MYX (<https://doc.itc.rwth-aachen.de/display/CCP/Project+MYX>) employees from RWTH Aachen University and Japans PC Cluster Consortium are working on such a standard-compliant tools interface. However, this is neither published nor implemented at the moment writing this thesis.

⁹<http://www.xcalablemp.org>

any kind of dynamic OpenMP correctness checking tool. The goal of this epoch model is to split the execution of a given program into different phase to enable the determination of a partial order (or rather the concurrency of different program parts). The model differentiates three hierarchy level (events, periods and epochs) and defines the corresponding happens-before relations. With the help of an extended PDA the thread-local phases are determined, before they are merged with the developed algorithm. Although minor limitations (e.g., task dependencies, untied tasks, thread private data in tasks) are left, a broad range of features is covered (e.g., target offloading / loop, task and nested parallelism). The evaluation with the LLVM runtime demonstrates the functionality of the model and the applicability for correctness checking tools will be shown in Section 5.1.3. The model benefits from the use of OMPT as an information source, because it directly takes the OpenMP runtime and memory model into account on the one hand and the overhead of the interface is low on the other hand. Thus, the development of powerful and scalable correctness checking tools with respect to the OpenMP semantics is possible. The generic design of the model keeps it flexible and thus expandable for future OpenMP feature developments. Furthermore, this chapter discussed the transferability of the general epoch concept to further parallel programming paradigms. The concrete realization is future work.

5. Tool Supported Analysis

As seen in Chapter 2, today's supercomputers are based on highly-parallel architectures. Many of these modern systems consist of clustered shared memory nodes, where each node usually includes multiple NUMA domains. The complexity of such a system motivates the use of hybrid programming approaches by combining different standards. While MPI is still the de-facto standard for distributed parallel programming, the used paradigm for the second hierarchy level depends on the underlying hardware. For pure shared memory systems OpenMP became the de-facto standard on the second level, while for accelerated systems typically programming paradigms like CUDA, OpenACC or OpenCL have been used. With the introduction of the target offloading directives in OpenMP 4.x the latter approaches might become obsolete, because the paradigm allows application development for heterogeneous systems in a more convenient way. Nevertheless, tools are required to analyze the performance behavior and the correctness of an application – independent of the decision of the actual programming paradigm. This chapter will show that the developed approaches can be integrated into real-world analysis tools, which already partly support the combination of MPI, OpenMP, OpenACC or CUDA. Thus, a holistic tool supported analysis of hybrid applications becomes possible.

This chapter is structured as follows. Section 5.1 discusses how the developed concepts can be applied by correctness checking tools, where Section 5.1.1 first gives a brief overview of the related work in this area. In Section 5.1.2 a method based on binary instrumentation for the determination of the required memory events is presented. Section 5.1.3 demonstrates the application of the epoch model for the correctness checking of pure OpenMP and hybrid programs, while Section 5.1.4 discusses the epoch-based analysis of the OpenMP specification. In Section 5.2 the extension of the OMPT interface is evaluated in the real-world performance analysis tool Score-P.

5.1. Correctness Checking

Due to the complexity of parallel programming paradigms, system architectures and the nondeterministic scheduling behavior, parallel programming in general is error-prone. The combination of two or more approaches as well as the growing range of features of each paradigm increase the complexity additionally. The diversity of the error classes is comprised of the class of “traditional” OpenMP usage errors, like race conditions, the class of “traditional” MPI usage errors, like deadlocks, and a

5. Tool Supported Analysis

combinatorial class that results from the mix of the paradigms. Furthermore, for the programming of target devices new classes of errors have to be considered (refer to Section 5.1.3).

While tools like debuggers or techniques for static program analysis can be used to find and correct a broad range of defects, others cannot (or only with very time-consuming effort) be detected. In order to overcome this limitation, runtime error detection tools can be used. Here, different tools for different programming paradigms exist. For instance, *Marmot Umpire Scalable Tool* (MUST) [43] enables a developer to detect a wide range of MPI defects. Here, the tool does not only detect those usage errors that are visible in the current runtime environment, but also state information on the origin of an error. This is important, because the correction of these errors increases the stability and portability of an application. Thus, codes can be used across different systems or software stacks. Most of runtime error detection tools for OpenMP (e.g., [50, 34, 73, 84]) concentrate on one of the most notorious errors in parallel programming: data races. However, the range of potential errors massively increased with the features introduced in version 4.x. For instance, data races cannot only occur on the host processor or on the accelerator anymore, but also between both by incorrect usage of the `target` directives. Automatic runtime detection tools for these new classes of errors are hard to find. To the best of my knowledge, no tools exist for the validation of the target offloading features.

5.1.1. Related Work

In the area of HPC especially the performance of an application and the efficient usage of the available hardware are essential. Beyond this, the importance of correctness validation is increasing as well, because the software and the hardware becomes more and more complex and thus leaving more latitude for programming mistakes. Since the topic of correctness validation is not a new research area, a lot of work has been done in the past already. Especially, on-the-fly data race detection is of certain interest, because data races form one of the most notorious classes of concurrency bugs. However, existing state-of-the-art approaches [50, 34, 39, 75, 82, 60] mostly lack on the ability to react to high level runtime events. In contrast to error checks based on the presented epoch model (refer to Section 4) they have to rely on low level synchronization events instead (e.g., those delivered by the operating system). Basically, on-the-fly data race detection algorithms can be classified into three different methods: lockset methods, happens-before analysis or hybrid approaches.

Lockset methods track for each shared memory address if the operation was protected when multiple threads access the address and at least one access is a write operation. The first implementation of such a lockset algorithm was Eraser [78], where Savage et al. using a state machine in order to keep track on data accesses to shared variables and to suppress warnings until a second thread causes a race. Thus, the algorithm only respects explicit locks used by the application, but not

other synchronization primitives (e.g., fork/join, high-level/paradigm-specific primitives). This may lead to high amount of false alarms, which is the main disadvantage of this method, because it limits the usability for many applications.

The most widespread algorithm for happens-before analysis is the *FastTrack* algorithm, developed by Flagan and Freund [34]. In order to decrease the runtime and memory overhead, they use an adaptive epoch representation by aggregating memory accesses which results in an overall lower runtime overhead. This improves the former algorithm DJIT+ [75] and leads to a complexity of $\mathcal{O}(n)$ in terms of space and runtime overhead. In contrast, the epoch model presented in this thesis uses a different definition of epochs, because the focus is on the OpenMP semantics. One of the major differences here is that support for nested parallelism, target offloading or task-based parallelism exists. The information for the determination of the happens-before relations is directly taken from the OpenMP runtime (and thus respects the language-specific semantics) instead of relying on low-level synchronization events. The advantage is that the OpenMP memory model [15] is implicitly taken into account, because the exact information about the data scoping is delivered by the OpenMP runtime.

Hybrid approaches combine happens-before analysis with lockset methods in order to decrease the runtime overhead of a pure happens-before relation and to decrease the false alarm rate of pure lockset methods significantly. The most popular implementation of such a hybrid approach is Helgrind+ [50]. While Helgrind+ does not respect paradigm-specific primitives, Ha et al. [39] focus on OpenMP. Their *iFT* (*improved FastTrack*) algorithm uses only epochs in each access history by applying the *left-of-relation* developed by Mellor-Crummey [60]. This relation defines partial ordering of vertices in a fork-join graph and locates at least one data race, if any exists. The main idea here is to only maintain two read access in the access history in order to decrease the overhead. This might lead to a reduced result set, where not all existing data races were detected. However, Mellor-Crummey formally proved that at least one of the existing data races will be detected. Thus an incremental work-flow to data race free application can be applied.

In contrast to all these previous works on dynamic data race detection, this thesis focuses on the epoch model itself as a generic approach for any kind of correctness checking tool. The goal is to provide a solid basis for correctness checking tools in general, not only classical race detection. However, the examples following in the next paragraph show how the model can be used for data race detection and the potential strength of the approach.

5.1.2. Instrumentation-based Memory Access Tracing

Section 4 shows in detail the determination of the happens-before relations for epochs, periods and OMPT events by using the OMPT interface. However, the interface cannot be used to trace the memory events (i.e., memory loads or stores), because it is not part of the OpenMP semantics. Nevertheless, this information is required in order to detect typical OpenMP programming mistakes. One technique for memory access tracing is binary instrumentation. Unfortunately, the tracing of all memory event occurring during the execution of an application involves a huge overhead in terms of the memory footprint and the runtime. This section will briefly present a memory access tracer and discuss how the epoch model can be used to decrease the resulting memory overhead.

This thesis uses the binary instrumentation framework Intel PIN [57] for the determination of the happens-before relations of the memory events. The framework is limited to x86-64 instruction-set architectures. The developed tool for dynamic program analysis collects all relevant memory events based on memory operation types and virtual addresses. The number of collected memory accesses in HPC applications might become huge during the execution. However, these accesses are not completely random, because processed data is often organized in arrays or other well-defined data structures. Furthermore, especially iterative methods (e.g., the CG benchmark used in this thesis) access the same addresses repeatedly. Since for many correctness checks the order the memory accesses is only relevant between two (synchronization) events, the memory overhead can be reduced significantly by storing the address information of a memory access within a period only once. Here, the data can be aggregated to ranges of accesses instead of storing every single address, which reduces the memory overhead significantly. The used format basically stores the start address S , the end address E , the stride δ , the kind of the memory operation (i.e., load/store) O and an optional parameter L for the source code location:

$$\boxed{S \mid E \mid \delta \mid O \mid [L]}$$

This is defined as *Strided Memory Access (SMA)*, which will be denote as

$$SMA = \{S, E, \delta, O, L\} \quad (5.1)$$

in the following. The source code location is optional for two reasons. First, the information is only available if the application was built with the corresponding debug information. Second, the additional parameter might decrease the compression rate, if one or more addresses are accessed from different source code locations. On the one hand the source code location is not required for the detection of typical OpenMP programming mistakes, on the other hand this information is useful for an application developer to fix the error. Thus, defining the parameter as optional keeps the maximum freedom for a certain use case. A SMA has to be differentiated between load and store operations, because this information is required for

an adequate data race detection. During the execution of an application two SMAs $SMA_1 = \{S_1, E_1, \delta_1, O_1, L_1\}$ and $SMA_2 = \{S_2, E_2, \delta_2, O_2, L_2\}$ might become combinable if $(O_1 = O_2) \wedge (L_1 = L_2)$, and one of the following conditions holds:

$$(\delta_1 = \delta_2) \wedge (|S_1 - S_2| = \alpha \cdot \delta_1) \wedge ((S_2 \leq (E_1 + \delta_1)) \vee (S_1 \leq (E_2 + \delta_2))), \text{ with } \alpha \in \mathbb{N}$$

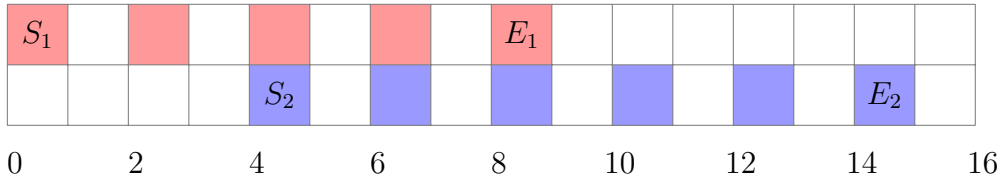
(aligned) (5.2a)

$$(\delta_1 = \delta_2) \wedge (|S_1 - S_2| = \frac{1}{2} \cdot \delta_1) \wedge (|E_1 - E_2| = \frac{1}{2} \cdot \delta_1)$$

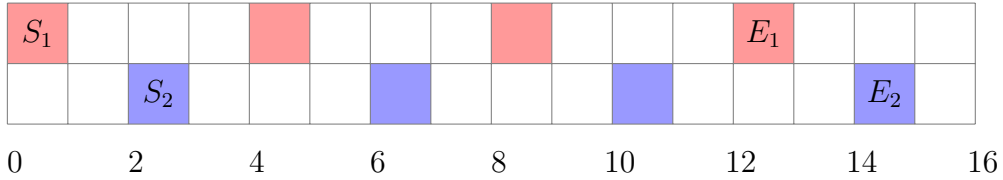
(interleaved) (5.2b)

$$(\delta_1 = \alpha \cdot \delta_2) \wedge (|S_1 - S_2| = \delta_1) \wedge (|E_1 - E_2| = \delta_1), \text{ with } \alpha \in \mathbb{N}$$

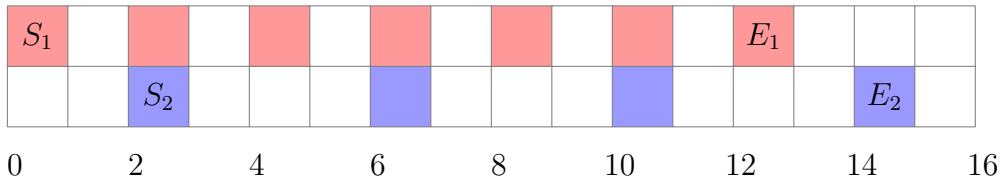
(multiple stride) (5.2c)



(a) Aligned aggregation with $SMA_1 = \{0, 8, 2, O, L\}$ and $SMA_2 = \{4, 14, 2, O, L\}$.



(b) Interleaved aggregation with $SMA_1 = \{0, 12, 4, O, L\}$ and $SMA_2 = \{2, 14, 4, O, L\}$.



(c) Multiple stride aggregation with $SMA_1 = \{0, 12, 2, O, L\}$ and $SMA_2 = \{2, 14, 4, O, L\}$.

Figure 5.1.: Examples for the aggregation of two SMAs.

5. Tool Supported Analysis

For each of the conditions a corresponding example is given in Figure 5.1. Equation 5.2a is fulfilled if the strides are the same and the offset of the start addresses differs by a multiple of the stride. In the example in Figure 5.1a the two SMAs $SMA_1 = \{0, 8, 2, O, L\}$ and $SMA_2 = \{4, 14, 2, O, L\}$ can be aggregated to a single consecutive one, which results in $SMA_3 = \{0, 14, 2, O, L\}$. Equation 5.2b is fulfilled if the strides are the same, but the offset of the start addresses differs by a half stride and thus the two SMAs are interleaved. This is depicted in Figure 5.1b, where the $SMA_1 = \{0, 12, 4, O, L\}$ and $SMA_2 = \{2, 14, 4, O, L\}$ can be aggregated to $SMA_3 = \{0, 14, 2, O, L\}$. Finally, Equation 5.2c describes the aggregation of two SMAs with different strides. Prerequisite is that one stride is a multiple of the other stride and the offset of the starting addresses is a multiple of the smaller stride. The example in Figure 5.1c depicts such a scenario, where $SMA_1 = \{0, 12, 2, O, L\}$ and $SMA_2 = \{2, 14, 4, O, L\}$ can be aggregated to the same SMA_3 as before.

By using this data aggregation the memory overhead is reduced significantly. However, applying the rules for each SMA might still not result in the optimal compression, because due to the application of one of the rules further optimization might become possible. Thus, the compression rate can be increased by applying the algorithm multiple times, which on the other hand causes additional computational overhead. Since the compression rate was appropriate, the corresponding compression algorithm was only executed once in this thesis. Another important point to keep in mind is the fact that the happens-before relations of the memory events get lost. In order to allow an adequate correctness analysis, the epoch model is used such that only memory events within the same period are aggregated. Due to the construction of the model no relevant information is lost for the use cases presented in Section 5.1.3. Depending on the concrete correctness check a tool developer wants to implement, further optimizations are possible. For instance the expensive memory tracing can be interrupted between two OMPT events, where no user code is executed (e.g., between an idle begin and an idle end event). However, this optimization is not possible if the model is used for the verification of the runtime implementation. Furthermore, one might consider a compression on epoch level instead of period level, which works because it is known that data accesses within an epoch are concurrent. This shows the further benefits of the developed epoch concept.

5.1.3. OpenMP-specific Error Detection: Methods and Algorithms

As discussed in Section 3, the OpenMP ARB published an interface for first-party performance analysis tools. Although the interface was designed for performance analysis, in [22] was shown that it is also sufficient for use in correctness checking tools. This section will discuss methods and algorithms for correctness checking based on the epoch model introduced in Section 4 and with respect to the error

classification presented in the next paragraph.



Figure 5.2.: Classification of common issues in OpenMP applications [65, 22].

OpenMP Target-specific Error Classification

A common classification of typical OpenMP errors was done by Münchhalfen et al. in [65]. In order to clarify ambiguous term *error*, they use the nomenclature given in [94], which also will be used in this work:

- **defect**: An incorrect program code (i.e., a programming error).
- **failure**: A visible manifestation of a defect (e.g., aborted execution, incorrect result, or deadlock).

Thus, only a failure is directly observable by the programmer. However, from the absence of a failure does not necessarily follow the absence of a defect. A simple example for this is the use of uninitialized variables, where a programmer assumes an initial value of zero. Especially during the development of a code where the programmer does not use any compiler optimization this assumption might be true by chance, which still leads to a correct program behavior. However, this not guaranteed for all hardware/software stacks and often changes as soon as the compiler

optimization is increased, because the uninitialized variable is put into a register instead of the top of stack, where the probability for a completely random initial value unequal zero is much higher. Thus, the code stability and portability will be increased by fixing the defect, although no failure can be observed in a certain environment.

Figure 5.2 shows the given error classification. It distinguishes between three main categories of issues: Syntactic defects, Semantic defects and Performance issues. In general, the first category covers violations concerning the grammar of a programming language. In case of OpenMP this can either be the use of wrong or non-existing compiler directives or the use wrong clause of the directive. This class of defects can easily be fixed, because the compiler will directly report this kind of issues. For the detection no runtime analysis tool is required. The third category will be discussed in the following paragraphs in detail. Since the work of Münchhalfen et al. focus on runtime detection tools it mainly discusses the second category of semantic defects. This category is divided into two subcategories. The first includes those defects which are a violation of the OpenMP specification or have a conceptual character. The second subcategory includes those defects, which manifests in one of the two typical failures in parallel programming: race conditions and deadlocks. The figure lists examples for each issue class at the lowest level. In Table 5.1 these examples are listed with respect to their detectability. Münchhalfen et al. differentiate here between the detection through the compiler, static code analysis, the OpenMP runtime system, debuggers or dynamic runtime tool. It can be seen that especially data races or races between a host device and a target device or deadlocks are hard to detect by using static code analysis or other methods. In the following it will be analyzed in detail how the developed epoch model can be applied in a dynamic runtime tool in order to automatically detect semantic defects.

Violation of the Standard

The first subclass of semantic defect defines all violations against the OpenMP specification which are not syntactically. This explicitly includes those defects which do not result in a failure (i.e., a visible manifestation of the defect).

Uninitialized locks: OpenMP provides several possibilities to guarantee exclusive access to a data region or a variable. Beside constructs like `critical` and `atomic` or the `reduction` clause, the library includes a set of general-purpose lock routines that can be used for synchronization. In OpenMP two types of locks exist – simple locks and nestable locks. In contrast to simple locks, it is valid to set a nestable lock multiple times by the same task before it is unset again. Each type of lock can also have a hint, which allows an optimization by the runtime implementation. All of these lock routines operate on specific lock variables, also defined in the OpenMP specification. In order to use them, the standard demands that it has to be initialized before it is accessed (e.g., acquired) for the first time. If this is not done in a given

#	Mistake	Compiler	Static Analysis	OpenMP Runtime	Debuggers	Runtime Tools
	<i>Syntactic mistakes</i>					
1.	wrong_directive		•			
2.	wrong_clause	•	•			
	<i>Semantic mistakes</i>					
	<i>Violation of the standard</i>					
3.	uninitialized_locks			•	•	•
4.	barrier_wo_all_threads		•	•		•
5.	violation_sese	(•)	•			
6.	worksharing_wo_all_threads					
7.	invalid_nesting	(•)	•	•		(•)
8.	lock_unlock_nonowner		•	•		•
9.	simd_aligned		(•)		•	
	<i>Conceptual defect</i>					
10.	parallel_inst_parallel_for					
11.	single_prod_wo_worksharing					
12.	number_of_threads					
13.	unallocated_memory	(•)	•	•	•	•
14.	missing_data_mapping	(•)	•			(•)
	<i>Race condition</i>					
15.	host			(•)		•
16.	accelerator			(•)		•
17.	host_accelerator					(•)
	<i>Deadlock</i>					
18.	multiple_locks		(•)			•
19.	single_lock		(•)			•

Table 5.1.: Defects from Figure 5.2 and their detectability: no mark means the type of tool is not able to detect the defect, (•) means that the type of tool is able to detect the defect but typically does not implement this functionality, • means that the tool is able to detect the defect and typically does implement the necessary functionality [65].

5. Tool Supported Analysis

application, the behavior is undefined (case #3 in Table 5.1). In order to detect this defect a correctness checking tool has to analyze the order of the events σ_{li} (lock init), σ_{ma} (mutex acquire), σ_{md} (mutex acquired), σ_{mr} (mutex release) and σ_{ld} (mutex destroy) in all epochs for a corresponding lock variable. With this event set all lock routines of the OpenMP specification are covered, including the variants for simple and nestable locks, as well as the locks with or without hint. Table 5.2 shows all lock routines and the events which are dispatched when the routine is called. The program is non-conforming if for a certain lock variable the following happens-before relation does not hold:

$$\sigma_{li} \prec \sigma_{ma} \prec \sigma_{md} \prec \sigma_{mr} \prec \sigma_{ld}, \quad (5.3)$$

where the event sequence $\sigma_{ma} \prec \sigma_{md} \prec \sigma_{mr}$ can occur several times between the lock initialization and the lock destroy.

Runtime Library Routine	Event(s)
omp_init_{nest}_lock	σ_{li}
omp_init_{nest}_lock_with_hint	σ_{li}
omp_set_{nest}_lock	$\sigma_{ma}, \sigma_{md}, \sigma_{nl}$
omp_unset_{nest}_lock	σ_{mr}, σ_{nl}
omp_test_{nest}_lock	$\sigma_{ma}, \sigma_{md}, \sigma_{nl}$
omp_destroy_{nest}_lock	σ_{ld}

Table 5.2.: List of events which occurs for OpenMP lock routines as defined in the technical report 4 (*TR4*) (refer to Table 4.1 for event names).

This rule allows detecting if a program tries to acquire or test a lock before it was initialized, which is the case when a σ_{ma} occurs without a corresponding σ_{li} . Although this case is reliable detectable, one has to keep in mind that – depending on the runtime implementation – the next event σ_{md} might not occur, because the runtime behavior for such a situation is not well-defined. For instance, accessing an uninitialized lock might result in a memory violation error detected by the operating system (i.e., a crash of the program). Furthermore, it is detectable if the lock was unset or destroyed without any initialization. However, this case is not reliable detectable for every runtime implementation and thus might fail, because for the events σ_{mr} and σ_{ld} the lock variable has to be accessed first. Thus, the application might crash before the events can be dispatched.

A comparison of the Intel 17.0 and GNU 6.3 runtime reveals that although the initialization of the lock variable is missing, the program is executed without any warnings or error messages with the GNU runtime, while it crashes with the LLVM runtime. This is because the initialization in the GNU runtime is done when the lock is acquired for the first time. This shows the benefit of a correctness checking tool, because an application developer using the GNU runtime can fix the defect before porting the program to another platform using the LLVM runtime.

```

1  MPI_Init(...);
2  #pragma omp parallel
3  {
4      if( omp_get_thread_num() % 2 ){
5          #pragma omp barrier
6          printf("Barrier 1\n");
7      } else {
8          #pragma omp barrier
9          printf("Barrier 2\n");
10     }
11 }
12 MPI_Finalize();

```

Listing 5.1: Program that fails to let all threads of a team reach the same OpenMP barrier construct.

Barrier not reached by all threads of a team: As mentioned before, barriers are another option for synchronization in OpenMP. Although they are less error-prone than lock routines, they offer potential for the violation of the *Single Entry, Single Exit* (SESE) principle of OpenMP (case #4 and #5 in Table 5.1). One consequence of this principle is, that all threads of a team have to execute the same barrier. Listing 5.1 depicts a MPI + OpenMP hybrid code example for such a violation. All threads with an even thread ID execute the first barrier (line 5) and all threads with an odd thread ID the second barrier (line 8) on alternating code paths. Depending on runtime implementation, the defect can manifest as deadlock, aborted execution, or may not manifest at all. In the first case, the threads wait for each other until they reach their respective barrier, which will never happen because of the alternating code paths. In the second case, the runtime may abort execution because of an internal runtime error or a timeout. In the last case, the two barriers are merged towards a single synchronization, such that the program terminates without any failures. Although this might be the intended behavior of the application, it is a violation of the OpenMP specification, which might cause execution failures on other platforms or with other runtime implementations. Such errors might be detectable with a static code analysis, if the code complexity is low. However, due to the dynamic behavior of the condition in the `if` statement, this is not possible for this example.

A dynamic analysis with the presented epoch model allows to analyze the barrier begin events σ_{b_b} within every parallel (or nested) epoch. The construction of the epoch model guarantees that in every epoch only one σ_{b_b} per thread exists. The OMPT interface does not provide unique identifiers for the barrier. This limitation in the interface can be overcome by resolving the return address of the call to the runtime routine. This can be done by evaluating the corresponding parameter which is handed over by the event. However, the technical report 4 (*TR4*) allows this parameter to be unset. In this case, a correctness checking tool has to determine the source code location of the barrier, for instance by resolving the first return address of the call stack, which is not within the OpenMP runtime. A corresponding detec-

5. Tool Supported Analysis

tion is shown in Algorithm 4, where Ω' contains all generated epochs (or rather all epochs collected up to a possible deadlock of the application). The worst case complexity of this algorithm is $\mathcal{O}(n)$, where n is the amount barriers in the applications.

Algorithm 4 Detection of multiple threads passing different barriers in pseudo-code

```

1: for  $e_i \in \Omega'$  do
2:    $old\_addr \leftarrow \emptyset$ 
3:   for  $\sigma_j \in e_i$  do
4:     if  $\sigma_j = \sigma_{b\_b}$  then
5:        $addr \leftarrow ResolveRetAddr(\sigma_j)$ 
6:       if  $addr = old\_addr \wedge old\_addr \neq \emptyset$  then
7:          $ReportError(addr)$ 
8:        $old\_addr \leftarrow addr$ 

```

MUST Output, starting date: Wed Sep 16 12:15:09 2015.

Rank(s)	Type	Message
0(27443)	Warning	You requested 4 threads by OMP_NUM_THREADS but requested MPI_THREAD_SINGLE from the ...
Details:		
Message		From
You requested 4 threads by OMP_NUM_THREADS but requested MPI_THREAD_SINGLE from the mpi library. This is ok as long as your application doesn't use any OpenMP between MPI_Init and MPI_Finalize.		Representative location: MPI_Init_thread (1st occurrence) called from: #0 main@omptest.c:15
0(27453)	Error	Error: Thread 3 of 4 (Parallelid: 2) passes a different barrier than other threads of the same team!
Details:		
Message		From
Error: Thread 3 of 4 (Parallelid: 2) passes a different barrier than other threads of the same team!		Representative location: main (1st occurrence) called from: #0 main@omptest.c:15
0(27452)	Error	Error: Thread 2 of 4 (Parallelid: 2) passes a different barrier than other threads of the same team!
0(27451)	Error	Error: Thread 1 of 4 (Parallelid: 2) passes a different barrier than other threads of the same team!
0(27451)	Error	Error: Thread 1 of 4 (Parallelid: 2) passes a different barrier than other threads of the same team!
0(27452)	Error	Error: Thread 2 of 4 (Parallelid: 2) passes a different barrier than other threads of the same team!
0(27453)	Error	Error: Thread 3 of 4 (Parallelid: 2) passes a different barrier than other threads of the same team!

MUST has completed successfully, end date: Wed Sep 16 12:15:11 2015.

Figure 5.3.: MUST error message for the incorrect usage of OpenMP barriers (refer to Listing 5.1).

In [22] this algorithm was implemented in the correctness checking tool MUST [43]. Figure 5.3 shows the MUST error report for the execution of the program presented in Listing 5.1. The report highlights two issues. First, it is reported that the initialization of MPI only requests `MPI_THREAD_SINGLE`, although multiple threads are used. This is a violation of the MPI specification and will lead to undefined behavior in most MPI implementations. The detailed discussion can be found in [22]. Second,

the report shows that the defect of the incorrect barrier invocation was successfully detected, including a hint to the source code line. The checks include the explicit barriers and the implicit barriers at the end of the `parallel` construct. Thus, multiple instances of the issue are reported. This example shows that the epoch concept for OpenMP can be integrated into tools already detecting other paradigm-specific defects such that hybrid program analysis becomes possible.

Invalid nesting of worksharing regions: Another example of a violation of the OpenMP specification is the invalid nesting of worksharing constructs (case #7 in Table 5.1). The behavior of a program is undefined if worksharing constructs are closely nested, because each worksharing requires the context of a single parallel region. If this nesting is done in the same source code file, a compiler is able to identify this kind of defects. However, in case of orphaned OpenMP regions or regions inside third-party libraries this might not be possible. Listing 5.2 and Listing 5.3 show a corresponding code example, where the loop construct of the function `foo` is closely nested into the outer loop construct of the main routine. The detection of this kind of errors is similar to Algorithm 4, because it has to be done within the same epoch as well. However, in contrast to Algorithm 4 this check has to analyze all worksharing event pairs Σ_{ws} instead of the barrier events, where the set of worksharing events is defined as

$$\Sigma_{ws} = \{\sigma_{l_b}, \sigma_{l_e}, \sigma_{se_b}, \sigma_{se_e}, \sigma_{si_b}, \sigma_{si_e}, \sigma_{so_b}, \sigma_{so_e}, \sigma_{ws_b}, \sigma_{ws_e}\}. \quad (5.4)$$

This definition directly follows from the worksharing constructs defined by the OpenMP specification: `loop` constructs, `sections` constructs, `single` constructs and `workshare` constructs. The algorithm for this check has to report an error as soon as a corresponding begin/end event pair is nested into another event pair $\in \Sigma_{ws}$. Since the check is done on the epoch level, it is ensured that the error is only reported if no new nested parallel region was encountered in between, because this would generate a new epoch. This shows another benefit of the application of the epoch model.

```

1  int main(){
2      #pragma omp parallel
3      {
4          #pragma omp for
5          for(int i=0; i < N; i++)
6              foo();
7      }
8      return 0;
9  }
```

Listing 5.2: Invalid nesting (main.c).

```

1  void foo(){
2      #pragma omp for
3      for(int i=0; i < N; i++)
4          printf("An inner iteration\n");
5      }
6
7
8
9
```

Listing 5.3: Invalid nesting (foo.c).

Conceptual Defects

The second subclass of semantic defects defines conceptual defects (refer to Figure 5.2). Regarding to Münchhalfen et al. [65] they occur when an OpenMP program

5. Tool Supported Analysis

has an unintended behavior, although it is conforming and without any violations of the specification.

Use of `parallel` construct instead of `parallel for`: An example for a conceptual defect is the unintended use of a directive (case #10 in Table 5.1). This might result in a syntactically correct program, which delivers wrong results because of a defective semantic. However, even a program with correct results might have conceptual defects. An example is given in Listing 5.4, which shows an initialization routine for an array of length `N`. In order to optimize the page distribution for a NUMA architecture, this initialization is done in parallel. However, a `parallel` construct instead of a combined `parallel for` construct was used. This means that the iterations are not distributed between the threads (worksharing), but every thread initializes every entry of the array. Thus, the computational results of the program are still correct, but the performance might not be as expected, because the desired effect of distributing the memory pages over the NUMA nodes was not achieved. From a correctness checking tool perspective these kinds of errors are hard to detect, since the intention of the user is unknown. By just using the OMPT interface this defect cannot be detected. However, the defect manifests as a race condition, which can be detected by combining the epoch model with memory access tracing. Data race detection is discussed in the next section in more detail.

```
1 void init(int* a) {
2
3     #pragma omp parallel
4     {
5         for (int i=0; i < N; i++)
6             a[i] = 42;
7     }
8
9     return;
10 }
```

Listing 5.4: Conceptual defect:
Semantically wrong
directive.

```
1 int sum = 0;
2 // no copy back of sum after
3 // execution of target region
4 #pragma omp target map(to:sum)
5 {
6     for (int i=0; i < N; i++)
7         sum += i;
8 }
9 // read outdated sum value on host
10 printf("%d", sum);
```

Listing 5.5: Conceptual defect: Missing
data mapping clause in
target region.

Missing data mapping: Another example for a conceptual defect is a missing data mapping clause for a `target` region. One effect of such a defect can be the use of outdated data on the host device (case #14 in Table 5.1). Listing 5.5 shows a program where a scalar value `sum` is calculated on a target device without mapping back the result. The behavior of the given example is undefined according to the OpenMP specification, even if it was the intention of the programmer to omit the mapping back to the host device. Thus, with one implementation the host could work on the old data, while on another it may operate on the new data. For the detection of this defect the presented memory tracer is used for the determination of all memory events $\sigma_m \in \Sigma_{mem}$. Here, a σ_m occurs in a target device region if a parent epoch with a σ_{tt_b} event exists. Furthermore, the data mapping for each

host device memory event σ_{m_host} to each target device memory event σ_{m_target} has to be determined:

$$\sigma_{m_target} \mapsto \sigma_{m_host}. \quad (5.5)$$

This mapping is known, because the presented extension of the OMPT interface (refer to Section 3.3) delivers a target and a host pointer of the mapped data with each OpenMP mapping clause. Combining this information, the check can report an error if a $\sigma'_{m_target} \prec \sigma'_{m_host}$, where σ'_{m_target} was a write event and no mapping back to the host exists:

$$\neg(\sigma'_{m_target} \mapsto \sigma'_{m_host}). \quad (5.6)$$

Race Conditions

The third class in the classification of OpenMP errors are formed by data races. They can occur on the host device, on the target device or between both (cases #15 - #17 in Table 5.1). In general, they are one of the most notorious defects in parallel programming, because they are not deterministic reproducible, hard to find manually and the existing techniques and dynamic data race detection tools are expensive in terms of performance and memory overhead. Furthermore, tools might produce false positives, which decrease their usability. One of the root causes for false positives is lack of information regarding the data synchronization primitives. Especially, programming paradigms based on dynamic runtime implementations (like OpenMP) might use own synchronization mechanism instead of low-level primitives. The developed epoch model (refer to Section 4) can be used in order to determine the corresponding happens-before relations with respect to the OpenMP memory and runtime model, because the corresponding information is delivered directly by the runtime system. Thus, the generated epochs do not depend on the information quality about the low-level synchronization primitives. The following paragraph will show how the information generated by the epoch model can avoid false positives for the analysis of a nested parallel OpenMP program. The goal here is not the development of a completely new and fully functional data race detector, but to show the potential benefits of an epoch-based approach.

As an example for the avoidance of false positive the OpenMP program in Listing 4.1 is examined again in a slightly modified version (Listing 5.6). Here, the programmer applied two performance optimization compared to the original version. In the first optimization, the data array is initialized in parallel by two threads. As discussed in Section 2.5 this improves the performance on systems using multiple NUMA domains (two in this case since two threads initialize the array) and an operating system with a first-touch policy. The outer parallel loop distributes the memory pages over both domains equally (assuming the size of the array is an even multiple of a page size). Given an adequate thread affinity and loop scheduling in the inner loop this optimization avoids expensive remote data accesses across the NUMA domains. The second optimization intends to avoid the implicit barrier at the end of the for-loop by using a `nowait` clause at the first worksharing construct.

5. Tool Supported Analysis

```
1  int i;
2  double* a = (double*)malloc(N*sizeof(double));
3
4  #pragma omp parallel num_threads(2)
5  {
6      // initialize complete array (parallel)
7      #pragma omp for private(i) nowait
8      for(i=0; i<N; i++){
9          a[i] = 41;
10     }
11     int tid = omp_get_thread_num();
12     #pragma omp parallel for num_threads(2)
13     for(i=tid*N/2; i<(tid+1)*N/2; i++){
14         a[i] += 1;
15     }
16 }
```

Listing 5.6: A simple OpenMP program with nested parallelism and parallel initialization.

Thus, the data can be processed in the nested parallel loop as soon as the corresponding master thread initialized the required part of data instead of waiting for the complete initialization. Although there is no implicit barrier between the data initialization and the data processing, this example is data race free, because the inner worker threads only access the data initialized by their master thread. Correctness checking tools which only detect those data races that actually occur during the execution will not report any false positives, because every thread access different parts of the data. However, epoch-based approaches can also detect potential race conditions and not only those which were encountered in the current execution. Due to the missing barrier an implementation of the presented epoch model generates one outer parallel epoch and two nested epochs, which looks similar to the case depicted in Figure 4.2. An epoch model which does not use the knowledge about the nested parallel region would report a false positive for this scenario, because the worker threads access parts of the data which were initialized by their master thread in the outer parallel loop. In contrast to that, the presented epoch model of this thesis allows to analyze the program beginning with the inner most nested epoch and ending with most outer loop. Thus, the detection of potential data races is possible without generating new false positives. This example shows the significance of the model and the potential benefit for a data race detection tool.

Another example for the usability is shown on an OpenMP program which approximates π by solving

$$\pi = \int_0^1 \frac{4}{1+x^2} dx, \quad (5.7)$$

where $x \in \mathbb{R}$. Listing 5.7 shows an OpenMP function which approximates the equation in parallel by simple numerical integration. Since the default data scoping in OpenMP is shared the variables `i` and `fx` have to be defined as private with the

```

1  double CalcPi (int n)
2  {
3      const double fH    = 1.0 / (double) n;
4      double fSum = 0.0;
5      double fX;
6      int i;
7
8      #pragma omp parallel for private(i, fX) //reduction(+:fSum)
9      for (i = 0; i < n; i++)
10     {
11         fX = fH * ((double)i + 0.5);
12         fSum += (4.0 / (1.0 + fX*fX));
13     }
14     return fH * fSum;
15 }

```

Listing 5.7: A data race in an OpenMP program calculating π .

corresponding clause. Furthermore, all threads sum up their partial result in `fSum` in parallel. However, in the given function the access to the variable is not protected (e.g., by using the `reduction` clause) and thus the code has a data race on `fSum`. The determination of the concurrency can be done with the presented epoch-based approach. By using the additional memory access information delivered by the Intel PIN tool (refer to Section 5.1.2) a successful detection of the race is possible. As soon as the `reduction` clause is added, the function behaves as expected and the implemented data race detection tool does not report any problems anymore.

5.1.4. Epoch-based Analysis of the OpenMP Specification

The intention of the developed epoch model (Section 4) is to provide a basis for any kind of correctness checking tool for OpenMP. The prerequisite for a reliable analysis is the correct event sequence delivered by the instrumented OpenMP runtime. Thus, the complete design is focused on correctness checking tools for applications. Nevertheless, it is worth to discuss whether the model can help to verify the correctness of a certain runtime implementation or even the OpenMP specification itself.

Verification of the Runtime Implementation

For the verification of an OpenMP runtime implementation three different scenarios are conceivable:

1. Verification of the runtime behavior based on a given code.
2. Verification of the tools interface.
3. Detection of data race within the runtime.

In the first scenario the correctness check has to define the expected result from the epoch generation for a given source code with a given number of threads. A very

5. Tool Supported Analysis

simple check without using the epoch model at all would be to compare the event sequence with the expectation. Nevertheless, in such a simple check one has to keep in mind that the order of events is not defined during a parallel execution. Thus, this has to be done on a per-thread basis. However, even for analysis on a per-thread basis, the event sequence is not well-defined, because of the dynamic program behavior. For instance, idle events may or may not occur during runtime, depending on the implementation, the timing, the number of threads, the thread affinity or the system load. Using the epoch model for checking against the expectation of a well-defined epoch sequence makes such a verification much more reliable. However, the effectiveness of those checks strongly depends on the definition of the expectation. The reason for that is the “as if rule” described in Section 3.3. For instance, one cannot expect a fixed amount of epochs, because a standard-compliant runtime or compiler might decide to generate more or less barriers than actually required without violating the specification. Although the huge flexibility for runtime developers makes the definition of reliable checks very complex, one can define a set of rules which at least have to be fulfilled in order to be standard-compliant. For example, the standard enforces at least one implicit barrier at the end of a parallel region. The absence is a clear indication of an error. The root of the error in this case can be either a missing synchronization or a missing barrier event. Both roots would be a violation of the standard as soon as OMPT becomes part of the OpenMP specification. However, this approach still cannot avoid all false negatives. Receiving OMPT events from the runtime does not necessarily mean that the runtime behaves correctly. It only shows that the corresponding event was triggered, but not that the action was executed in fact. For instance, a runtime could claim executing a barrier by calling the correctness checking tool without actually synchronizing the thread team. The triggered event might fulfill the defined rules, but the runtime behavior is still wrong in this case. In order to avoid such false negatives, an adequate verification also has to take the timing of the events into account (e.g., a barrier end event cannot occur before all barrier begin events of the all threads of the same team were triggered).

The second scenario is the validation of the implementation of the tools interface itself. Therefore, an application which is known as defect-free has to be used. If under this prerequisite the invalid sink state q_s is reached caused by an invalid event sequence, the deduction will be a defective runtime. However, the robustness of a tool with such an use-case is strongly limited, because the interface is used as source information in order to validate the interface itself.

As seen before, race conditions are one of the most common failures in parallel programming (especially in OpenMP). However, they might not only occur in a certain program, but also in the runtime implementation. Thus, the third scenario for a runtime validation is the data race detection within the runtime implementation. The combination of the defined epoch model with memory access tracing (refer to Section 5.1.2) allows efficient data race detection for OpenMP applications. Since binary instrumentation is used for the memory access tracing in this

work, analyzing the memory accesses within an OpenMP runtime is possible as well. However, the synchronization within the runtime is done with low-level mechanism (e.g., pthreads), but not with OpenMP constructs. As a consequence, the developed epoch model based on OMPT events cannot directly be used for the detection of data races within the runtime. Furthermore, the transition relations of the PDA would have to be redefined, because other synchronization functions might behave completely different compared to those given in OpenMP. The same is true for the set of states. Nevertheless, the principle idea of the epoch model is still adoptable, but would need a completely different event source (e.g., by wrapping low-level synchronization calls) and definition of the PDA.

Verification of the Specification

The semantic of OpenMP becomes more and more complex with every new version of the specification. This is not only reflected in the number of pages of the specification (the first version had a length of 85 pages while the latest version has almost 400), it can also be seen in many internal discussions within the Language Committee, where the OpenMP experts often discuss about the compliance of relatively short code snippets. The increasing complexity is not only a result of the increasing amount of features, but also the possibility of combining them. For instance, it is possible to use almost any OpenMP construct within a target region. This happens in a new target device data environment. An poor definition in the standard (e.g., which data is mapped to/from a device implicitly or explicitly) can easily lead to undefined program or runtime behavior. Furthermore, not all constructs within a target region lead to a well-defined state. One example is the use of nested target regions (also referred to as *reverse offloading*). This is not explicitly restricted by the specification, but it is claimed to have an unspecified behavior. This gives a vendor the flexibility to implement it, but might lead to non-portable code. Already in early versions of the specification the Language Committee realized the importance of well-defined models. This insight lead to a complete memory model [14, 15], for instance. Thus, it is worth to discuss whether a formal approach as used in the epoch model can help to avoid such definition lacks or mistakes in the specification.

Basically, the idea here is to detect those combinations of OpenMP constructs which lead to the sink state q_s by causing an invalid transition relation before actually implementing them into a certain runtime. Here, the advantage of the PDA is, that it allows to express all combinations of constructs with a more compact and well-defined method than it is possible with a textual specification. Definition lacks would be much faster and easier detectable by identifying undefined transition relations. However, since the epoch model, as defined in this work (including all transition relations), is based on the given specification and not vice versa, the approach at least is not suited for a formal verification of the OpenMP semantics. Furthermore, the model is based on runtime events, which means that at least a

prototype implementation including the new features which have to be validated has to be available. Nevertheless, given the latter prerequisite, the model still can be used to receive advices to potential issues of the new feature in an early phase of the specification process. Since the acceptance of new features through the ARB often presupposes the existence of a prototype implementation anyway, this co-design of specification and runtime implementation is a realistic scenario. In fact, this also happened with the proposal of the tools interface, where the combinations of the technical report [32] and the prototype implementation are the foundation for a potential standardization. The fact that improvements in the revised technical report [31] are also based on the experiences made during the implementation process, shows the tightly coupled process for in this case.

5.2. Performance Analysis

As seen in the previous section, standard-compliant tool support can be used for correctness checking with respect to the underlying runtime and memory model. However, the main intention of tools interfaces in HPC is the performance analysis. In order to enable the tool developer to create portable tools and thus the application programmer to analyze the complete set of programs covered by the used programming paradigm, it is essential to have a well-defined tools interface which covers the complete language semantics. Therefore, in Section 3.3 an extension for OMPT was defined, which represents an important step to fulfill this requirement for OpenMP. In the following the capabilities and the applicability of this extension in real-world performance analysis tools will be evaluated.

Based on the implementation in the LLVM OpenMP implementation (refer to Section 3.3), the interface was implemented in the performance measurement system Score-P by the Technical University of Dresden. The corresponding adapter developed in the context of [56] and [20] registers and handles all OMPT events including the added target-related events. In order to identify and record a program region or a function call, Score-P uses corresponding region handles. This region handle has to be passed to the begin and end event of the region. The correlation between the begin and the end is done by passing a region handle into the runtime with every begin event. The corresponding end event includes this handle in the callbacks signature. While the early implementation in [20] used a non standard-compliant mechanism to transfer the collected performance information back to the host, the latest version uses the proposed tracing interface. For the evaluation one benchmark from the preliminary SPEC ACCEL OpenMP Target Offloading suite [51] as a representative of a real-world application has been used. The measurements were done with a prototype implementation of Score-P provided by the Technical University of Dresden, including the integration to the visualization tool Vampir [52]. The goal of this evaluation is not a performance optimization of the benchmark, but the demonstration that an adequate performance analysis of applications using

`target` constructs is possible with the proposed OMPT extension. The tests have been performed on the Xeon Phi system presented in Section 2.2.

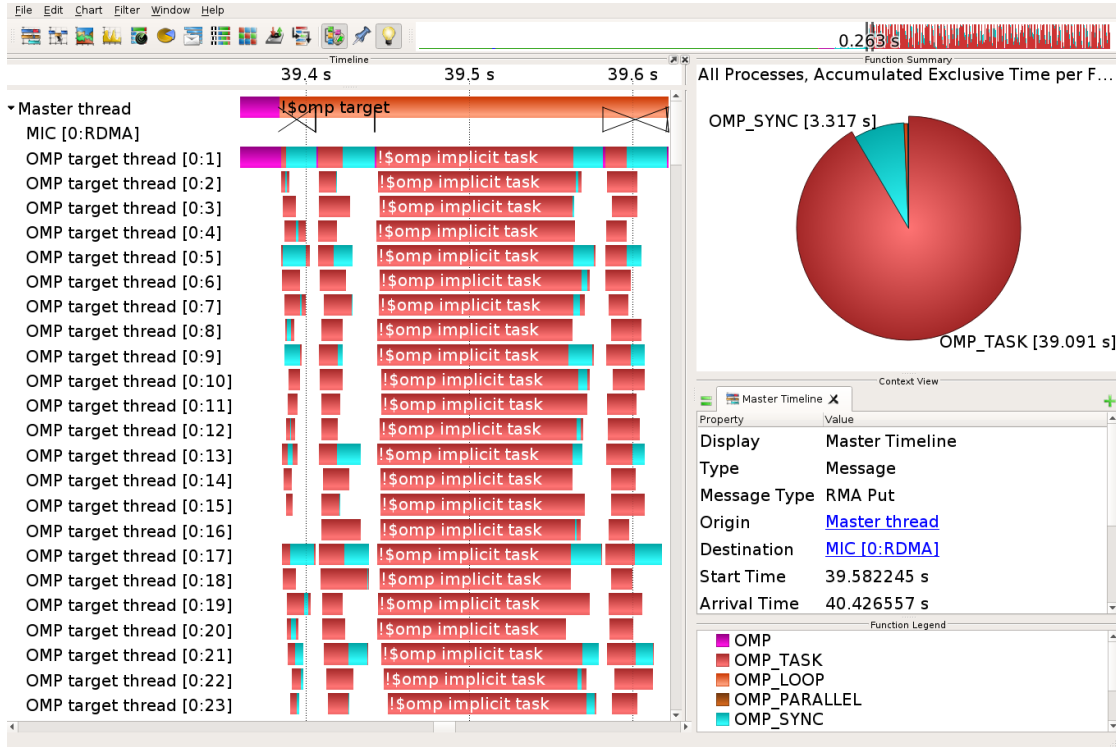


Figure 5.4.: Trace visualization of the SPEC ACCEL Benchmark 554.cg.

Figure 5.4 shows the visualization of the trace for the preliminary version of the SPEC ACCEL benchmark 554.cg. In order to recognize more detail, the figure only depicts a small part the execution time, which can be seen in the timeline on the top. Furthermore, only a small amount of the 236 target device threads are included. The algorithm used in the benchmark is a CG method similar to the benchmark presented in Section 2.3. However, the benchmark uses offloading directives in order to execute the iterations of the CG method on a target device. The execution of the target region is represented by the orange bar of the master thread. The black lines between the target region and the target device (named as “MIC”) represent the data transfers via *Remote Direct Memory Access* (RDMA) to or from the device. Since one of these lines is selected in the window, the *Context View* widget (in the middle right) shows more details to the message. For instance the start and the arrival times of the message are shown, which allows the determination of the bandwidth by combining this information with the message size. Furthermore, it can be seen that each target device thread executes the corresponding implicit tasks (red bars) and the synchronization is done by a `barrier` construct (blue bars). In the pie chart in the top right the share of the accumulated exclusive times grouped by the construct type is shown. Since the share of the synchronization time is small compared to the

5. *Tool Supported Analysis*

time spend in implicit tasks, the selected execution time frame is well balanced here.

This example shows that the proposed and implemented OMPT extension allows a detailed performance analysis for program using the OpenMP target offloading features. Furthermore, the OMPT tracing API supports tool developer to use tracing on target device in a convenient, portable and vendor-independent fashion.

6. Summary and Conclusion

Driven by the application requirements and the developments in computer architectures during the last decade, the expressiveness of parallel programming paradigms increased continuously. This increasing amount of functionality shifts some of the programming burden from the developer to the compiler and runtime time system by offering convenient methods for the programming of homogeneous or heterogeneous systems. However, also the complexity of parallel programs applying these new features increases. On the one hand, this leads to the necessity of a deep comprehension of the runtime behavior of an application in order to achieve the best possible performance. On the other hand, applications tend to be more error-prone. One key factor for the understanding of the performance and the correctness of a parallel program is reflected in the analysis of the memory accesses. Since modern supercomputers often consist of clustered shared memory systems or integrate many-core architectures, a parallel programming paradigm which enables expressing the parallelism within in a node is required in order to optimize the single node performance. With a focus on the de-facto standard for shared memory programming OpenMP, this thesis takes the hardware properties, the programming paradigm, its particular implementation and the interfaces for an appropriate tool support into account.

The user application studies of the two real-world FEM simulation codes ZaKo3D and iMOOSE (Chapter 1) from RWTH Aachen University showed the importance and benefits of interdisciplinary work and practice-oriented solutions. By combining the expertise of the domain researchers with the knowledge of computer specialists, significant improvements for the stability, correctness and performance of the in-house simulation packages have been achieved. Due to the more efficient usage of the available hardware, this enables deeper scientific insides for the domain researchers and motivated further investigations for this thesis in order to allow scalable, portable and sustainable program analysis.

Chapter 2 discussed the efficient memory access for OpenMP target devices and for task-based programming. The presented systematic methodology for the assessment of target devices includes five steps: The determination of the basic performance characteristics, the assessment of the paradigm-specific overheads, the scalability determination, the model-driven performance prediction and the evaluation with standard benchmark suites. The comparison of the single- and parallel-producer tasking patterns on large NUMA machines stressed the importance of adequate thread and data placement on the one hand, and the needs for a deep understand-

6. Summary and Conclusion

ing of the internal runtime behavior on the other hand. Both become even more important with the increasing amount of cores per node and more hierarchy levels of the NUMA domains or a bigger heterogeneity of the system.

Motivated by that, in Chapter 3 the support of standard-compliant target offloading support was improved. The extension for the OMPT interface includes host-sided events for the encountering of target region, data mappings and transfer to or from the target device. Based on this proposal and discussions within the OpenMP tool Subcommittee, a tracing API for the target-sided events was integrated into the official technical report from the OpenMP ARB and thus will be part of OpenMP 5.0. Furthermore, the complete extension has been implemented into the LLVM runtime, which is one of the most popular runtime implementations. Thus, even a prototype of the tracing API is available for a broad HPC community, which ensures the sustainability of this work.

For the determination of the happens-before relations of OpenMP programs, Chapter 4 introduces the first epoch model based on OMPT events. In order to respect the OpenMP semantics including the runtime and memory model of the paradigm, the different concurrency phases of an application are computed by an extended *Pushdown Automaton* (PDA). This PDA supports OpenMP worksharing, any parallel nesting level, target offloading and basic task-based programs. The generic design ensures the flexibility and extensibility of the approach for a solid foundation for any kind of correctness checking tool. The evaluation showed that the approach is applicable in general and the overhead for typical OpenMP programs is linear with respect to the generated epochs. Furthermore, the transferability to other parallel programming paradigms like OpenACC, Cilk Plus or XMP was discussed.

Finally, Chapter 5 evaluates the developed concepts with the real-world performance analysis tools Score-P and Vampir and the real-world correctness checking tool MUST. By combining a binary instrumentation-based memory access tracer with the OMPT interface, the epoch model is applied for the detection of a broad range of programming errors. This includes classical data race detection on a host device, as well as the detection of other semantic defects like a wrong API usage, violations of the standard, conceptual defects or race conditions between a host and a target device. Furthermore, the benefits of the epoch model for the analysis of the OpenMP specification or a certain runtime implementation have been discussed. The evaluation of the OMPT tracing API in Score-P and Vampir showed that the concept allows a vendor-independent performance analysis.

In conclusion, this thesis showed that the information gathered by the runtime can be used to ensure the performance and the correctness of parallel OpenMP applications. In order to ensure the sustainability of my work, the insights of the investigations have been contributed to the standard-compliant tools interface and

implemented in a popular open source runtime. This allows academic and commercial tool and runtime developers to create reliable and portable software components for the analysis of parallel programs. Furthermore, the epoch model enables the development of a broad range of correctness checking tools, which becomes more and more important with an increasing amount of programming paradigm features and increasing system sizes.

Statement of Originality

The insights presented in this thesis did profit from the close collaboration within the HPC team of the IT Center of RWTH Aachen University, headed by Professor Müller. The lively discussions, the involvement and the constructive criticism from and with the team improved many of the developed ideas and techniques. This thesis is partly based on several publications where members of teams or partners from other institutions or companies have been involved. In general, the order of the authors reflects the contribution in terms of novel approaches. In the following I provide a detailed overview on the collaborations and contributions of the (co-)authors for each individual chapter.

The user application studies in Chapter 1 are partly based on the following own publications:

- **Numerical simulation of electrical machines by means of a hybrid parallelisation using MPI and OpenMP for finite-element method [11] and Mesh Decomposition for Efficient Parallel Computing of Electrical Machines by Means of FEM Accounting for Motion [12].** This interdisciplinary work is a close collaboration with the Institute for Electrical Machines (IEM) of RWTH Aachen University. My contribution is the design and implementation of the OpenMP parallelization in the context of my diploma thesis, which was supervised by Enno Lange and evaluated by Professor Hameyer. The MPI part of the parallelization including the mesh decomposition has been developed by Stefan Böhmer in the context of his diploma thesis, which was supervised by Enno Lange, Martin Hafner and myself. This diploma thesis was evaluated by Professor Hameyer and Professor Bischof.
- **Praxisgerechte Strategien und Methoden zur effizienten FE-Berechnung des Zahnkontakts [21].** This interdisciplinary work is a close collaboration with the Laboratory for Machine Tools and Production Engineering (WZL) of RWTH Aachen University. My contributions are the analysis of the new PARDISO-based solver on large NUMA systems and the performance improvements based on a better data and thread affinity. The implementation and integration of the new solver was developed by Melanie Heidgen in the context of her bachelor thesis, which I supervised. The application use cases and the insights regarding the optimization of the gear tooth contact were contributed by Jannik Henser. Dieter an Mey contributed constructive criticism and practice-oriented ideas regarding the requirements for the solver.

6. Summary and Conclusion

Chapter 2 is partly based on the following own publications:

- **OpenMP Programming on Intel Xeon Phi Coprocessors: An Early Performance Comparison** [23]. My key contributions to this work are the comparisons between the BCS machine and the Xeon Phi with a focus on the CG and EPCC benchmarks. The model-driven investigations and the application of the Roofline Model are a close collaboration with Dirk Schmidl. Michael Klemm, who is a senior application engineer at Intel and the Chief Executive Officer (CTO) of the OpenMP ARB, contributed the technical details of the Xeon Phi and supported optimization of the benchmarks for the new Xeon Phi micro-architecture.
- **Assessing OpenMP Tasking Implementations on NUMA Architectures** [88] and **Task-Parallel Programming on NUMA Architectures** [89]. My contributions to this work are the analysis of the tasking patterns with the CG benchmark and the analysis of the data distribution on the BCS system. Christian Terboven contributed the proposal and discussion of the programming patterns. A benchmark for the examination of the task behavior on NUMA architectures was developed by Dirk Schmidl. Beyond this work, I analyzed one of the used runtimes regarding the task queue implementations in detail.
- **Assessing the Performance of OpenMP Programs on the Intel Xeon Phi** [81]. My key contributions to this work are the performance investigations with the spMV operation, the NAS Parallel Benchmarks and the iMOOSE suite. Dirk Schmidl had the initial idea to this publication and compared the architectures. This includes the investigations with the kernel and EPCC benchmarks. The application case studies with the FIRE and NestedCP code were contributed by Christian Terboven and Dirk Schmidl, the case study of the NINA code has been contributed by Sandra Wienke.
- **An OpenMP Extension Library for Memory Affinity** [80]. In this publication I supported the evaluation of the extension library by providing and analyzing the CG benchmark code. Dirk Schmidl developed the library, analyzed the overhead and proposed possible changes to the OpenMP specification. Christian Terboven supported the complete development process.

Chapter 3 is partly based on the following own publications:

- **Performance Analysis for Target Devices with the OpenMP Tools Interface** [20]. I developed the extension of the OMPT interface including the new events, signatures and inquiry functions in a close collaboration with Robert Dietrich from the Technical University Dresden. Furthermore, I contributed the double buffering benchmark, while Robert Dietrich contributed the investigations with an early preliminary version of one of the SPEC ACCEL

benchmarks. The evaluation with a different SPEC ACCEL benchmark in Section 5.2 has been done by me. The implementation in the LLVM OpenMP runtime was developed by me, the tool integration of the interface in Score-P and Vampir was developed by Robert Dietrich.

- **Evaluation of Tool Interface Standards for Performance Analysis of OpenACC and OpenMP Programs** [27]. My contribution to this work is the extension and evaluation of the OMPT interface. Robert Dietrich had the initial idea to this publication and contributed the extensions and analysis to the OpenACC tools interface. Furthermore, he integrated both interfaces into Score-P, supported by Ronny Tschüter and Guido Juckeland from the Technical University Dresden.
- **OMPT: An OpenMP Tools Application Programming Interface for Performance Analysis. Revised OpenMP Technical Report 2** [31]. The revised technical report is based on [32]. My key contribution is the extension of the target-related events. The initial idea to the OMPT interface is from Alexandre Eichenberger (IBM), John Mellor-Crummey (Rice University), Martin Schulz (Lawrence Livermore National Laboratory), Nawal Copti (Oracle), Robert Dietrich (TU Dresden), Xu Liu (Lawrence Livermore National Laboratory), Eugene Loh (Oracle), Daniel Lorenz (Jülich Supercomputer Center) and other members of the OpenMP Tools Working Group.

Chapter 4 is partly based on the following own publications:

- **An OpenMP Epoch Model for Correctness Checking** [24]. My key contribution to this work is the novel idea of a generic epoch model based on the OMPT interface, including the determination of the happens-before relations by using an extended PDA. Simon Schwitanski supported the development of the PDA and contributed the principle definitions of the events, periods and epochs in his bachelor thesis [83], which I supervised. Professor Müller and Professor Katoen evaluated this bachelor thesis. Felix Münchhalfen supported the development of the epoch merging algorithm. Beyond this publication, I extended the model by the target device and tasking capabilities in this thesis.

Chapter 5 is partly based on the following own publications:

- **Extending MUST to Check Hybrid-Parallel Programs for Correctness using the OpenMP Tools Interface** [22]. My key contribution to this work is the application of the epoch model to pure OpenMP and hybrid parallel programs. Felix Münchhalfen contributed the error classification and the thread-safe implementation of the correctness checking tool MUST, including the extended event model. Christian Terboven and Tobias Hilbrich supported the development process of the hybrid concept in the MUST implementation.

List of Figures

2.1.	OpenMP fork-join model.	10
2.2.	OpenMP nested parallelism. A $\bullet$ marks a fork or a join.	11
2.3.	Architecture overview of the BCS system, based on [38, 23].	13
2.4.	Architecture overview of the Intel Xeon Phi coprocessor, based on [40, 23].	14
2.5.	Sparsity pattern of the used matrix.	20
2.6.	STREAM memory bandwidth of the BCS system and the Intel Xeon Phi Coprocessor for different amount of threads and two binding strategies on the BCS.	24
2.7.	Scalability of the CG benchmark (1000 iterations) on the Xeon Phi coprocessor and the 128-core BCS machine for different versions and binding policies.	29
2.8.	Roofline Model applied to the Intel Xeon Phi and the BCS system.	32
2.9.	Performance of the spMV within the CG method on the Xeon Phi coprocessor and the 128-core BCS system.	33
2.11.	Page distribution over the NUMA nodes after the matrix initialization.	39
2.12.	Performance of the spMV kernel within the CG method for different initialization strategies and different amount of threads.	39
2.13.	Task scheduling in the LLVM runtime.	41
2.14.	Performance of spMV for different implementations.	42
3.1.	Sequence diagram for OMPT buffering API.	57
3.2.	Data transfer and control flow for target devices.	58
3.3.	Tracing API implemented with COI.	62
3.4.	Share of OMPT event types in selected SPEC ACCEL benchmarks.	65
4.1.	Pushdown Automaton for the epoch generation. The transition relations are labeled as following. First row: input symbol $\sigma \in \Sigma$ (refer to Table 4.1); Second row: stack operation $\text{pop}(\gamma_i \in \Gamma)$ / $\text{push}(\gamma_j \in \Gamma)$, where a $*$ can be any $\gamma \in \Gamma$; Third row (if exists): output symbol ω . If no ω is specified in the third row, the empty string ϵ is emitted implicitly. The symbol $\parallel$ separates different transitions, if there are multiple transitions between two states.	77
4.2.	Periods and epochs generated for a nested parallel OpenMP program (refer to Listing 4.1). Dotted edges mark a fork or a join. Italic nodes/gray edges mark idle events which can be assigned to different epochs as well. Dashed boxes mark the generated nested epochs.	84

List of Figures

4.3.	A possible task tree for the computation of the Fibonacci sequence with $i = 5$. The generated (nested) epochs are shown within the dashed lines. The execution/scheduling of the tasks depends on the runtime behavior and might differ between multiple runs.	93
5.1.	Examples for the aggregation of two SMAs.	107
5.2.	Classification of common issues in OpenMP applications [65, 22]. . . .	109
5.3.	MUST error message for the incorrect usage of OpenMP barriers (refer to Listing 5.1).	114
5.4.	Trace visualization of the SPEC ACCEL Benchmark 554.cg.	123

List of Tables

2.1.	Overhead in microseconds for the OpenMP constructs <code>parallel</code> , <code>for</code> , <code>parallel for</code> , <code>barrier</code> , <code>single</code> , <code>critical</code> , <code>lock/unlock</code> and <code>reduction</code> . The measurements were performed with the EPCC Microbenchmark <code>syncbench</code> on the BCS system and the Intel Xeon Phi Coprocessor.	26
2.2.	Execution time shares and the best solving times (1000 iterations) for the linear algebra kernels on a BCS system and a Xeon Phi Coprocessor.	33
2.3.	Runtime (in seconds) and speedup of the NAS parallel benchmarks on the Xeon Phi and a 2-socket SNB system.	35
2.4.	Percentage of tasks that were executed by a different thread than they were created from for the single- and parallel-producer pattern on the 4-sockets and 16-sockets systems using the CG kernel with 1024 tasks.	43
3.1.	Relevant information for the target-specific OMPT events.	53
3.2.	OMPT Record Sizes.	64
3.3.	SPEC ACCEL 557.pcsp: Tracing buffer sizes for all event types. . .	66
4.1.	Selected input symbols $\sigma \in \Sigma$ based on the OMPT events.	75
4.2.	PDA computation of the master thread for the nested parallelism example (refer to Table 4.1 for the input symbols).	85
5.1.	Defects from Figure 5.2 and their detectability: no mark means the type of tool is not able to detect the defect, (●) means that the type of tool is able to detect the defect but typically does not implement this functionality, ● means that the tool is able to detect the defect and typically does implement the necessary functionality [65]. . . .	111
5.2.	List of events which occurs for OpenMP lock routines as defined in the technical report 4 (<i>TR4</i>) (refer to Table 4.1 for event names). . .	112

Listings

2.1.	EPCC kernel to determine the parallel time of a for construct, based on [17].	16
2.2.	EPCC kernel to determine the reference time of a for construct, based on [17].	16
2.3.	EPCC kernel to determine the reference time of a target construct, based on [23].	27
2.4.	EPCC kernel to determine the parallel time of a target construct, based on [23].	28
2.5.	Single-producer, multiple-executors tasking pattern.	37
2.6.	Parallel-producer, multiple-executors tasking pattern.	37
3.1.	The union type for the OMPT tracing API [31].	59
4.1.	A simple OpenMP program with nested parallelism and serial initialization. The arrows on the right-hand side show the amount of active threads for each line of the code.	83
4.2.	A naive computation of the Fibonacci sequence with OpenMP tasks.	92
4.3.	A naive computation of the Fibonacci sequence with Cilk Plus.	97
4.4.	A simple XMP vector-dot kernel using the global-view memory model.	99
5.1.	Program that fails to let all threads of a team reach the same OpenMP barrier construct.	113
5.2.	Invalid nesting (main.c).	115
5.3.	Invalid nesting (foo.c).	115
5.4.	Conceptual defect: Semantically wrong directive.	116
5.5.	Conceptual defect: Missing data mapping clause in target region.	116
5.6.	A simple OpenMP program with nested parallelism and parallel initialization.	118
5.7.	A data race in an OpenMP program calculating π	119
A.1.	OMPT record types [31].	149

Bibliography

- [1] L. Adhianto, S. Banerjee, M. Fagan, M. Krentel, G. Marin, J. Mellor-Crummey, and N. R. Tallent. HPCToolkit: Tools for performance analysis of optimized parallel programs. In *Concurrency and Computation: Practice and Experience*, pages 685–701, 2010.
- [2] A. Aho, J. Hopcroft, and J. Ullman. Time and tape complexity of pushdown automaton languages. *Information and Control*, 13(3):186 – 206, 1968.
- [3] V. Aslot, M. J. Domeika, R. Eigenmann, G. Gaertner, W. B. Jones, and B. Parady. SPEComp: A New Benchmark Suite for Measuring Parallel Computer Performance. In *Proceedings of the International Workshop on OpenMP Applications and Tools: OpenMP Shared Memory Parallel Programming*, WOMPAT '01, pages 1–10, London, UK, 2001. Springer-Verlag.
- [4] V. Aslot and R. Eigenmann. Performance Characteristics of the SPEC OMP2001 Benchmarks. *SIGARCH Comput. Archit. News*, 29(5):31–40, December 2001.
- [5] E. Ayguadé, A. Duran, J. Hoeftinger, F. Massaioli, and X. Teruel. An Experimental Evaluation of the New OpenMP Tasking Model. In V. Adve, M. J. Garzarán, and P. Petersen, editors, *Languages and Compilers for Parallel Computing*, pages 63–77, Berlin, Heidelberg, 2008. Springer-Verlag.
- [6] D. Bailey, E. Barszcz, J. Barton, D. Browning, R. Carter, L. Dagum, R. Fatoohi, P. Frederickson, T. Lasinski, R. Schreiber, H. Simon, V. Venkatakrisnan, and S. Weeratunga. The Nas Parallel Benchmarks. *The International Journal of Supercomputing Applications*, 5(3):63–73, 1991.
- [7] N. Bell and M. Garland. Implementing Sparse Matrix-vector Multiplication on Throughput-oriented Processors. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, SC '09, pages 18:1–18:11, New York, NY, USA, 2009. ACM.
- [8] I. Bethune, F. Reid, and A. Lazzaro. CP2K Performance from Cray XT3 to XC30. In *Proceedings of the Cray User Group Conference*, CUG, 2014.
- [9] S. Blair-Chappell and A. Stokes. *Parallel Programming with Intel Parallel Studio XE*. ITPro collection. Wiley, 2012.

- [10] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou. Cilk: An Efficient Multithreaded Runtime System. *SIGPLAN Not.*, 30(8):207–216, Aug. 1995.
- [11] S. Boehmer, T. Cramer, M. Hafner, E. Lange, C. Bischof, and K. Hameyer. Numerical simulation of electrical machines by means of a hybrid parallelisation using MPI and OpenMP for finite-element method. *IET Science, Measurement and Technology*, 6:339–343(4), September 2012.
- [12] S. Boehmer, E. Lange, M. Hafner, T. Cramer, C. Bischof, and K. Hameyer. Mesh Decomposition for Efficient Parallel Computing of Electrical Machines by Means of FEM Accounting for Motion. *Magnetics, IEEE Transactions on*, 48(2):891 –894, feb. 2012.
- [13] C. Brecher, C. Gorgels, P. Kauffmann, T. Röthlingshöfer, A. Flodin, and J. Henser. 3D Tooth Contact Analysis: Simulation Possibilities for PM Gears. In *Proceedings of the EuroPM 2010: Production and Applications of Soft Magnetic Materials for Electric Motors*, pages 61–68. Shrewsbury: EPMA, 2010.
- [14] G. Bronevetsky and B. R. Supinski. Complete Formal Specification of the OpenMP Memory Model. *International Journal of Parallel Programming*, 35(4):335–392, 2007.
- [15] G. Bronevetsky and B. R. Supinski. Formal Specification of the OpenMP Memory Model. In M. S. Mueller, B. M. Chapman, B. R. Supinski, A. D. Malony, and M. Voss, editors, *OpenMP Shared Memory Parallel Programming: International Workshops, IWOMP 2006, Eugene, OR, USA, June 1-4, 2005, Reims, France, June 12-15, 2006. Proceedings*, pages 324–346, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.
- [16] F. Broquedis, N. Furmento, B. Goglin, P.-A. Wacrenier, and R. Namyst. Forest-GOMP: An Efficient OpenMP Environment for NUMA Architectures. *International Journal of Parallel Programming*, 38:418–439, 2010. 10.1007/s10766-010-0136-3.
- [17] J. M. Bull. Measuring Synchronisation and Scheduling Overheads in OpenMP. In *In Proceedings of First European Workshop on OpenMP*, pages 99–105, 1999.
- [18] J. M. Bull and D. O’Neill. A Microbenchmark Suite for OpenMP 2.0. *SIGARCH Comput. Archit. News*, 29(5):41–48, Dec. 2001.
- [19] M. E. Conway. A Multiprocessor System Design. In *Proceedings of the November 12-14, 1963, Fall Joint Computer Conference*, AFIPS ’63 (Fall), pages 139–146, New York, NY, USA, 1963. ACM.

- [20] T. Cramer, R. Dietrich, C. Terboven, M. S. Müller, and W. E. Nagel. Performance Analysis for Target Devices with the OpenMP Tools Interface. In *Parallel and Distributed Processing Symposium Workshop (IPDPSW), 2015 IEEE International*, pages 215–224. IEEE, May 2015.
- [21] T. Cramer, J. Henser, M. Heidgen, J. Pollaschek, D. an Mey, M. Brumm, M. S. Müller, and C. Brecher. Praxisgerechte Strategien und Methoden zur effizienten FE-Berechnung des Zahnkontakts. In *4. Kongress zu Einsatz und Validierung von Simulationmethoden für die Antriebstechnik (SIMPEP)*, pages 107–119, Sep 2014.
- [22] T. Cramer, F. Münchhalfen, T. Hilbrich, C. Terboven, and M. S. Müller. Extending MUST to Check Hybrid-Parallel Programs for Correctness using the OpenMP Tools Interface. In *Parallel Tools Workshop 2015*, September 2015.
- [23] T. Cramer, D. Schmidl, M. Klemm, and D. an Mey. OpenMP Programming on Intel Xeon Phi Coprocessors: An Early Performance Comparison. In *Proceedings of the Many-core Applications Research Community (MARC) Symposium at RWTH Aachen University*, pages 38–44, November 2012.
- [24] T. Cramer, S. Schwitanski, F. Münchhalfen, C. Terboven, and M. S. Müller. An OpenMP Epoch Model for Correctness Checking. In *2016 45rd International Conference on Parallel Processing Workshops (ICPPW)*, pages 299–308, August 2016.
- [25] T. A. Davis. University of Florida Sparse Matrix Collection. *NA Digest*, 92, 1994.
- [26] R. Dietrich, F. Schmitt, A. Grund, and D. Schmidl. Performance Measurement for the OpenMP 4.0 Offloading Model. In L. Lopes, J. Žilinskas, A. Costan, R. G. Cascella, G. Kecskemeti, E. Jeannot, M. Cannataro, L. Ricci, S. Benkner, S. Petit, V. Scarano, J. Gracia, S. Hunold, S. L. Scott, S. Lankes, C. Lengauer, J. Carretero, J. Breitbart, and M. Alexander, editors, *Euro-Par 2014: Parallel Processing Workshops*, pages 291–301, Cham, 2014. Springer International Publishing.
- [27] R. Dietrich, R. Tschüter, T. Cramer, G. Juckeland, and A. Knüpfer. Evaluation of Tool Interface Standards for Performance Analysis of OpenACC and OpenMP Programs. In *Parallel Tools Workshop 2015*, page TBD, September 2015.
- [28] J. Dongarra, M. A. Heroux, and P. Luszczek. High-performance conjugate-gradient benchmark: A new metric for ranking high-performance computing systems. *The International Journal of High Performance Computing Applications*, 30(1):3–10, 2016.

- [29] J. Dongarra, C. B. Moler, J. Bunch, and G. W. Stewart. *LINPACK users' guide*. Society for Industrial and Applied Mathematics, 1979.
- [30] A. Duran, X. Teruel, R. Ferrer, X. Martorell, and E. Ayguade. Barcelona OpenMP Tasks Suite: A Set of Benchmarks Targeting the Exploitation of Task Parallelism in OpenMP. In *2009 International Conference on Parallel Processing*, pages 124–131, Sept 2009.
- [31] A. E. Eichenberger, J. Mellor-Crummey, M. Schulz, N. Copt, J. Cownie, T. Cramer, R. Dietrich, X. Liu, E. Loh, and D. Lorenz. OMPT: An OpenMP Tools Application Programming Interface for Performance Analysis. Revised OpenMP Technical Report 2. <https://github.com/OpenMPToolsInterface/OMPT-Technical-Report>, August 2015. Accessed: 2015-09-08.
- [32] A. E. Eichenberger, J. M. Mellor-Crummey, M. Schulz, M. Wong, N. Copt, R. Dietrich, X. Liu, E. Loh, and D. Lorenz. OMPT: An OpenMP Tools Application Programming Interface for Performance Analysis. In A. P. Rendell, B. M. Chapman, and M. S. Müller, editors, *IWOMP*, volume 8122 of *Lecture Notes in Computer Science*, pages 171–185. Springer, 2013.
- [33] A. E. Eichenberger, C. Terboven, M. Wong, and D. an Mey. The Design of OpenMP Thread Affinity. In *Proceedings of the 8th International Conference on OpenMP in a Heterogeneous World, IWOMP'12*, pages 15–28, Berlin, Heidelberg, 2012. Springer-Verlag.
- [34] C. Flanagan and S. N. Freund. FastTrack: Efficient and Precise Dynamic Race Detection. *SIGPLAN Not.*, 44(6):121–133, June 2009.
- [35] K. Furlinger and M. Gerndt. ompP: A Profiling Tool for OpenMP. In *Proceedings of the 2005 and 2006 International Conference on OpenMP Shared Memory Parallel Programming, IWOMP'05/IWOMP'06*, pages 15–23, Berlin, Heidelberg, 2008. Springer-Verlag.
- [36] M. Geimer, F. Wolf, B. J. N. Wylie, E. Ábrahám, D. Becker, and B. Mohr. The Scalasca Performance Toolset Architecture. *Concurr. Comput. : Pract. Exper.*, 22(6):702–719, Apr. 2010.
- [37] B. Goglin and N. Furmento. Memory Migration on Next-Touch. In *Linux Symposium*, Montreal, Canada, July 2009.
- [38] D. Gutfreund. Mesca BCS Systems, October 2012. Bull SAS.
- [39] O.-K. Ha, I.-B. Kuh, G. M. Tchamgoue, and Y.-K. Jun. On-the-fly Detection of Data Races in OpenMP Programs. In *Proceedings of the 2012 Workshop on Parallel and Distributed Systems: Testing, Analysis, and Debugging, PADTAD 2012*, pages 1–10, New York, NY, USA, 2012. ACM.

- [40] A. Heinecke, M. Klemm, and H.-J. Bungartz. From GPGPUs to Many-Core: NVIDIA Fermi* and Intel® Many Integrated Core Architecture. *Computing in Science and Engineering*, 14(2):78–83, March–April 2012.
- [41] A. Heinecke, K. Vaidyanathan, M. Smelyanskiy, A. Kobotov, R. Dubtsov, G. Henry, A. G. Shet, G. Chrysos, and P. Dubey. Design and Implementation of the Linpack Benchmark for Single and Multi-node Systems Based on Intel Xeon Phi Coprocessor. In *2013 IEEE 27th International Symposium on Parallel and Distributed Processing*, pages 126–137, May 2013.
- [42] M. R. Hestenes and E. Stiefel. Methods of Conjugate Gradients for Solving Linear Systems. *Journal of Research of the National Bureau of Standards*, 49(6):409–436, December 1952.
- [43] T. Hilbrich, M. Schulz, B. R. de Supinski, and M. S. Müller. MUST: A Scalable Approach to Runtime Error Detection in MPI Programs. In M. S. Müller, M. M. Resch, A. Schulz, and W. E. Nagel, editors, *Tools for High Performance Computing 2009: Proceedings of the 3rd International Workshop on Parallel Tools for High Performance Computing, September 2009, ZIH, Dresden*, pages 53–66, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.
- [44] J. P. Hoefflinger and B. R. De Supinski. The OpenMP Memory Model. In *Proceedings of the 2005 and 2006 International Conference on OpenMP Shared Memory Parallel Programming, IWOMP’05/IWOMP’06*, pages 167–177, Berlin, Heidelberg, 2008. Springer-Verlag.
- [45] Intel Corporation. Intel Cilk Plus Language Extension Specification Version 1.2. [online] <https://www.cilkplus.org/specs>, September 2013. Accessed 2017-01-31.
- [46] Intel Corporation, Santa Clara, USA. *Intel Math Kernel Library. Reference Manual*, 2017.
- [47] B. M. Irons. A frontal solution program for finite element analysis. *International Journal for Numerical Methods in Engineering*, 2(1):5–32, 1970.
- [48] M. Itzkowitz and Y. Maruyama. HPC Profiling with the Sun Studio Performance Tools. In S. M. Müller, M. M. Resch, A. Schulz, and E. W. Nagel, editors, *Tools for High Performance Computing 2009: Proceedings of the 3rd International Workshop on Parallel Tools for High Performance Computing, September 2009, ZIH, Dresden*, pages 67–93, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.
- [49] A. C. Jacob, R. Nair, A. E. Eichenberger, S. F. Antao, C. Bertolli, T. Chen, Z. Sura, K. O’Brien, and M. Wong. Exploiting Fine- and Coarse-Grained Parallelism Using a Directive Based Approach. In C. Terboven, B. R. de Supinski, P. Reble, B. M. Chapman, and M. S. Müller, editors, *OpenMP: Heterogenous Execution and Data Movements: 11th International Workshop on OpenMP*,

- IWOMP 2015, Aachen, Germany, October 1-2, 2015, Proceedings*, pages 30–41, Cham, 2015. Springer International Publishing.
- [50] A. Jannesari, K. Bao, V. Pankratius, and W. F. Tichy. Helgrind+: An efficient dynamic race detector. In *Parallel Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on*, pages 1–13, May 2009.
 - [51] G. Juckeland, W. Brantley, S. Chandrasekaran, B. Chapman, S. Che, M. Colgrove, H. Feng, A. Grund, R. Henschel, W.-M. Hwu, H. Li, M. S. Müller, M. Perminov, P. Shelepugin, K. Skadron, J. Stratton, A. Titov, K. Wang, M. van Waveren, B. Whitney, S. Wienke, R. Xu, and K. Kumaran. SPEC ACCEL – A Standard Application Suite for Measuring Hardware Accelerator Performance. In *5th International Workshop on Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems, SC14*, November 2014.
 - [52] A. Knüpfer, H. Brunst, J. Doleschal, M. Jurenz, M. Lieber, H. Mickler, M. S. Müller, and W. E. Nagel. The Vampir Performance Analysis Tool-Set. In M. Resch, R. Keller, V. Himmler, B. Krammer, and A. Schulz, editors, *Tools for High Performance Computing: Proceedings of the 2nd International Workshop on Parallel Tools for High Performance Computing, July 2008, HLRS, Stuttgart*, pages 139–155, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.
 - [53] A. Knüpfer, C. Rössel, D. a. Mey, S. Biersdorff, K. Diethelm, D. Eschweiler, M. Geimer, M. Gerndt, D. Lorenz, A. Malony, W. E. Nagel, Y. Oleynik, P. Philippen, P. Saviankou, D. Schmidl, S. Shende, R. Tschüter, M. Wagner, B. Wesarg, and F. Wolf. Score-P: A Joint Performance Measurement Runtime Infrastructure for Periscope, Scalasca, TAU, and Vampir. In H. Brunst, S. M. Müller, E. W. Nagel, and M. M. Resch, editors, *Tools for High Performance Computing 2011: Proceedings of the 5th International Workshop on Parallel Tools for High Performance Computing, September 2011, ZIH, Dresden*, pages 79–91, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.
 - [54] S. Lankes, B. Bierbaum, and T. Bemmeler. Affinity-on-next-touch: An Extension to the Linux Kernel for NUMA Architectures. In *Proceedings of the 8th International Conference on Parallel Processing and Applied Mathematics: Part I, PPAM’09*, pages 576–585, Berlin, Heidelberg, 2010. Springer-Verlag.
 - [55] H. Löf and S. Holmgren. Affinity-on-next-touch: Increasing the Performance of an Industrial PDE Solver on a cc-NUMA System. In *Proceedings of the 19th Annual International Conference on Supercomputing, ICS ’05*, pages 387–392, New York, NY, USA, 2005. ACM.
 - [56] D. Lorenz, R. Dietrich, R. Tschüter, and F. Wolf. A comparison between OPARI2 and the OpenMP tools interface in the context of Score-P. In *Proc. of the 10th International Workshop on OpenMP (IWOMP), Salvador, Brazil*,

- September 2014, volume 8766 of *LNCS*, pages 161–172. Springer International Publishing, Sept. 2014.
- [57] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '05, pages 190–200, New York, NY, USA, 2005. ACM.
 - [58] J. McCalpin. STREAM: Sustainable Memory Bandwidth in High Performance Computers. <http://www.cs.virginia.edu/stream>, 1999. [Online, accessed 22-January-2017].
 - [59] G. H. Mealy. A Method for Synthesizing Sequential Circuits. *Bell System Technical Journal*, 34(5):1045–1079, 1955.
 - [60] J. Mellor-Crummey. On-the-fly detection of data races for programs with nested fork-join parallelism. In *Proceedings of the 1991 ACM/IEEE Conference on Supercomputing*, Supercomputing '91, pages 24–33, New York, NY, USA, 1991. ACM.
 - [61] B. Mohr, A. D. Malony, S. Shende, and F. Wolf. Design and Prototype of a Performance Tool Interface for OpenMP. *J. Supercomput.*, 23(1):105–128, August 2002.
 - [62] B. Mohr and F. Wolf. KOJAK - a tool set for automatic performance analysis of parallel applications. In *In Proc. of the European Conference on Parallel Computing (EuroPar)*, pages 1301–1304, 2003.
 - [63] MPI Forum. MPI: A Message-Passing Interface Standard, Version 3.0, September 2012.
 - [64] M. S. Müller, J. Baron, W. C. Brantley, H. Feng, D. Hackenberg, R. Henschel, G. Jost, D. Molka, C. Parrott, J. Robichaux, P. Shelepugin, M. van Waveren, B. Whitney, and K. Kumaran. SPEC OMP2012 – an Application Benchmark Suite for Parallel Systems Using OpenMP. In *Proceedings of the 8th International Conference on OpenMP in a Heterogeneous World*, IWOMP'12, pages 223–236, Berlin, Heidelberg, 2012. Springer-Verlag.
 - [65] J. F. Münchhalfen, T. Hilbrich, J. Protze, C. Terboven, and M. S. Müller. Classification of Common Errors in OpenMP Applications. In L. De Rose, B. R. de Supinski, S. L. Olivier, B. M. Chapman, and M. S. Müller, editors, *Using and Improving OpenMP for Devices, Tasks, and More*, volume 8766 of *Lecture Notes in Computer Science*, pages 58–72. Springer International Publishing, 2014.

- [66] C. J. Newburn, S. Dmitriev, R. Narayanaswamy, J. Wiegert, R. Murty, F. Chinchilla, R. Deodhar, and R. McGuire. Offload Compiler Runtime for the Intel Xeon Phi Coprocessor. In *Parallel and Distributed Processing Symposium Workshops PhD Forum (IPDPSW), 2013 IEEE 27th International*, pages 1213–1225, May 2013.
- [67] NVIDIA. CUDA C Programming Guide, Version 7.5, September 2015.
- [68] NVIDIA. *CUPTI User’s Guide*. NVIDIA Corporation, September 2015.
- [69] S. L. Olivier, A. K. Porterfield, K. B. Wheeler, and J. F. Prins. Scheduling task parallelism on multi-socket multicore systems. In *Proceedings of the 1st International Workshop on Runtime and Operating Systems for Supercomputers, ROSS ’11*, pages 49–56, New York, NY, USA, 2011. ACM.
- [70] OpenACC-Standard.org. The OpenACC Application Program Interface, Version 2.5, October 2015.
- [71] OpenMP Architecture Review Board. OpenMP Application Program Interface, Version 4.5, November 2015.
- [72] OpenMP Architecture Review Board. OpenMP Technical Report 4: Version 5.0 Preview 1, November 2016.
- [73] P. Petersen and S. Shah. OpenMP Support in the Intel®Thread Checker. In *Proceedings of the OpenMP Applications and Tools 2003 International Conference on OpenMP Shared Memory Parallel Programming, WOMPAT’03*, pages 1–12, Berlin, Heidelberg, 2003. Springer-Verlag.
- [74] C. A. Petri. *Kommunikation mit Automaten*. PhD thesis, Universität Hamburg, 1962.
- [75] E. Pozniarsky and A. Schuster. Efficient On-the-fly Data Race Detection in Multithreaded C++ Programs. *SIGPLAN Not.*, 38(10):179–190, June 2003.
- [76] D. J. Quinlan. ROSE: compiler support for object-oriented frameworks. *Parallel Processing Letters*, 10(2/3):215–226, 2000.
- [77] A. D. Robison. Composable Parallel Patterns with Intel Cilk Plus. *Computing in Science Engineering*, 15(2):66–71, March 2013.
- [78] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson. Eraser: A Dynamic Data Race Detector for Multi-threaded Programs. *SIGOPS Oper. Syst. Rev.*, 31(5):27–37, Oct. 1997.
- [79] O. Schenk and K. Gärtner. Solving unsymmetric sparse systems of linear equations with PARDISO. *Future Generation Computer Systems*, 20(3):475 – 487, 2004. Selected numerical algorithms.

- [80] D. Schmidl, T. Cramer, C. Terboven, D. an Mey, and M. S. Müller. An OpenMP Extension Library for Memory Affinity. In L. DeRose, B. R. de Supinski, S. L. Olivier, B. M. Chapman, and M. S. Müller, editors, *Using and Improving OpenMP for Devices, Tasks, and More*, volume 8766 of *Lecture Notes in Computer Science*, pages 103–114. Springer International Publishing, 2014.
- [81] D. Schmidl, T. Cramer, S. Wienke, C. Terboven, and M. S. Müller. Assessing the Performance of OpenMP Programs on the Intel Xeon Phi. In F. Wolf, B. Mohr, and D. an Mey, editors, *Euro-Par 2013 Parallel Processing*, volume 8097 of *Lecture Notes in Computer Science*, pages 547–558. Springer Berlin Heidelberg, 2013.
- [82] D. Schonberg. On-the-fly Detection of Access Anomalies. *SIGPLAN Not.*, 24(7):285–297, June 1989.
- [83] S. Schwitanski. Error Classification and Correctness Checking for Directive-based Offloading Programming on the Intel Xeon Phi. Bachelor thesis, RWTH Aachen University, 2015.
- [84] K. Serebryany and T. Iskhodzhanov. ThreadSanitizer: Data Race Detection in Practice. In *Proceedings of the Workshop on Binary Instrumentation and Applications*, WBIA '09, pages 62–71, New York, NY, USA, 2009. ACM.
- [85] S. S. Shende and A. D. Malony. The Tau Parallel Performance System. *Int. J. High Perform. Comput. Appl.*, 20(2):287–311, May 2006.
- [86] H. Sutter. The Free Lunch Is Over: A Fundamental Turn Toward Concurrency in Software. *Dr. Dobbs's Journal*, 30(3), 2005.
- [87] C. Terboven, D. an Mey, D. Schmidl, H. Jin, and T. Reichstein. Data and Thread Affinity in OpenMP Programs. In *Proceedings of the 2008 Workshop on Memory Access on Future Processors: A Solved Problem?*, MAW '08, pages 377–384, New York, NY, USA, 2008. ACM.
- [88] C. Terboven, D. Schmidl, T. Cramer, and D. an Mey. Assessing OpenMP Tasking Implementations on NUMA Architectures. In B. M. Chapman, F. Massaioli, M. S. Müller, and M. Rorro, editors, *OpenMP in a Heterogeneous World*, volume 7312 of *Lecture Notes in Computer Science*, pages 182–195. Springer Berlin Heidelberg, 2012.
- [89] C. Terboven, D. Schmidl, T. Cramer, and D. an Mey. Task-Parallel Programming on NUMA Architectures. In C. Kaklamanis, T. Papatheodorou, and P. Spirakis, editors, *Euro-Par 2012 Parallel Processing*, volume 7484 of *Lecture Notes in Computer Science*, pages 638–649. Springer Berlin Heidelberg, 2012.
- [90] D. van Riesen, C. Monzel, C. Kaehler, C. Schlensok, and G. Henneberger. iMOOSE - an open-source environment for finite-element calculations. *IEEE Transactions on Magnetics*, 40(2):1390–1393, March 2004.

- [91] S. Wienke, C. Terboven, J. C. Beyer, and M. S. Müller. A Pattern-Based Comparison of OpenACC and OpenMP for Accelerator Computing. In *Euro-Par 2014: Parallel Processing : 20th International Conference, Porto, Portugal, August 25-29, 2014, Proceedings / Silva, Fernando and Dutra, Inês and Santos Costa, Vítor (eds.)*, volume 8632 of *Lecture Notes in Computer Science*, pages 812–823, Cham [u.a.], 2014. Springer.
- [92] S. Williams, A. Waterman, and D. Patterson. Roofline: an Insightful Visual Performance Model for Multicore Architectures. *Commun. ACM*, 52(4):65–76, April 2009.
- [93] XcalableMP Specification Working Group. XcalableMP Language Specification, Version 1.2.1, November 2014.
- [94] A. Zeller. *Why Programs Fail: A Guide to Systematic Debugging*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2005.

A. Appendix: OMPT Record Types

Listing A.1 shows all OMPT record types as proposed in the Revised Technical Report 2 (rTR2) [31].

```
1  /* OMPT record type */
2  typedef enum ompt_record_type_e {
3      ompt_record_ompt                = 1,
4      ompt_record_native              = 2,
5      ompt_record_invalid             = 3
6  } ompt_record_type_t;
7
8  typedef enum ompt_record_native_class_e {
9      ompt_record_native_class_info  = 1,
10     ompt_record_native_class_event = 2
11 } ompt_record_native_class_t;
12
13 /* native record abstract */
14 typedef struct ompt_record_native_abstract_s {
15     ompt_record_native_class_t rclass;
16     const char *type;
17     ompt_target_time_t start_time;
18     ompt_target_time_t end_time;
19     uint64_t hwid;
20 } ompt_record_native_abstract_t;
21
22 /* record types */
23 typedef struct ompt_record_thread_begin_s {
24     ompt_thread_type_t thread_type; /* type of thread */
25 } ompt_record_thread_begin_t;
26
27 typedef struct ompt_record_idle_s {
28     ompt_scope_endpoint_t endpoint; /* begin or end */
29 } ompt_record_idle_t;
30
31 typedef struct ompt_record_parallel_begin_s {
32     ompt_task_id_t parent_task_id; /* ID of parent task */
33     const ompt_frame_t *parent_frame; /* frame data of parent task */
34     ompt_parallel_id_t parallel_id; /* ID of parallel region */
35     uint32_t requested_team_size; /* requested number of threads */
36     ompt_invoker_t invoker; /* who invokes master task? */
37     const void *codeptr_ra; /* return address of api call */
38 } ompt_record_parallel_begin_t;
39
40 typedef struct ompt_record_parallel_end_s {
41     ompt_parallel_id_t parallel_id; /* ID of parallel region */
42     ompt_task_id_t task_id; /* ID of task */
43     ompt_invoker_t invoker; /* who invokes master task? */
44     const void *codeptr_ra; /* return address of api call */
45 } ompt_record_parallel_end_t;
46
47 typedef struct ompt_record_task_create_s {
48     ompt_task_id_t parent_task_id; /* ID of parent task */
49     const ompt_frame_t *parent_frame; /* frame data for parent task */
50     ompt_task_id_t new_task_id; /* ID of created task */
51     ompt_task_type_t type; /* type of task being created */
```

A. Appendix: OMPT Record Types

```

52  _Bool has_dependences;          /* task has data dependences */
53  const void *codeptr_ra;         /* return address of api call */
54  } ompt_record_task_create_t;
55
56  /* note: not task dependences record since it points to data */
57  /*      rather than containing it */
58
59  typedef struct ompt_record_task_dependence_s {
60      ompt_task_id_t src_task_id;  /* ID of dependence source */
61      ompt_task_id_t sink_task_id; /* ID of dependence sink */
62  } ompt_record_task_dependence_t;
63
64  typedef struct ompt_record_task_schedule_s {
65      ompt_task_id_t prior_task_id; /* ID of descheduled task */
66      _Bool prior_completed;        /* true if prior task completed */
67      ompt_task_id_t next_task_id;  /* ID of scheduled task */
68  } ompt_record_task_schedule_t;
69
70  typedef struct ompt_record_scoped_implicit_s {
71      ompt_scope_endpoint_t endpoint; /* begin or end */
72      ompt_parallel_id_t parallel_id; /* ID of parallel region */
73      ompt_task_id_t task_id;        /* ID of task */
74      uint32_t thread_num;           /* OMP thread num */
75  } ompt_record_scoped_implicit_t;
76
77  typedef struct ompt_record_scoped_sync_region_s {
78      ompt_sync_region_kind_t kind;  /* barrier, taskwait, taskgroup */
79      ompt_scope_endpoint_t endpoint; /* begin or end */
80      ompt_parallel_id_t parallel_id; /* ID of parallel region */
81      ompt_task_id_t task_id;        /* ID of task */
82      const void *codeptr_ra;        /* return address of api call */
83  } ompt_record_scoped_sync_region_t;
84
85  typedef struct ompt_record_lock_init_s {
86      _Bool is_nest_lock;            /* nested lock or not */
87      ompt_wait_id_t wait_id;        /* wait ID */
88      uint32_t hint;                 /* OMP lock hint */
89      uint32_t kind;                 /* implementation kind */
90      const void *codeptr_ra;        /* return address of api call */
91  } ompt_record_lock_init_t;
92
93  typedef struct ompt_record_lock_destroy_s {
94      _Bool is_nest_lock;            /* nested lock or not */
95      ompt_wait_id_t wait_id;        /* ID of mutex being awaited */
96      const void *codeptr_ra;        /* return address of api call */
97  } ompt_record_lock_destroy_t;
98
99  typedef struct ompt_record_mutex_acquire_s {
100      ompt_mutex_kind_t kind;        /* kind of mutex */
101      uint32_t hint;                 /* based on OMP lock hint */
102      uint32_t impl;                 /* implementation of mutex */
103      ompt_wait_id_t wait_id;        /* ID of mutex being awaited */
104      const void *codeptr_ra;        /* return address of api call */
105  } ompt_record_mutex_acquire_t;
106
107  typedef struct ompt_record_mutex_s {
108      ompt_mutex_kind_t kind;        /* type of mutex */
109      ompt_wait_id_t wait_id;        /* ID of mutex being awaited */
110      const void *codeptr_ra;        /* return address of api call */
111  } ompt_record_mutex_t;
112
113  typedef struct ompt_record_scoped_nested_lock_s {
114      ompt_scope_endpoint_t endpoint; /* begin or end */
115      ompt_wait_id_t wait_id;        /* ID of mutex being awaited */
116      const void *codeptr_ra;        /* return address of api call */
117  } ompt_record_scoped_nested_lock_t;

```

```

118
119 typedef struct ompt_record_target_data_s {
120     ompt_target_id_t host_op_id;          /* host side ID for operation */
121     ompt_target_data_op_t optype;         /* type of operation */
122     void *host_addr;                     /* host address of the data */
123     void *device_addr;                   /* device address of the data */
124     size_t bytes;                         /* number of bytes mapped */
125     ompt_target_time_t end_time;          /* end time */
126 } ompt_record_target_data_t;
127
128 typedef struct ompt_record_target_kernel_s {
129     ompt_target_id_t host_op_id;          /* host side ID for operation */
130     uint32_t granted_num_teams;           /* number of teams granted */
131     ompt_target_time_t end_time;          /* end time */
132 } ompt_record_target_kernel_t;
133
134 typedef struct ompt_record_scoped_master_s {
135     ompt_scope_endpoint_t endpoint;        /* begin or end */
136     ompt_parallel_id_t parallel_id;        /* ID of parallel region */
137     ompt_task_id_t task_id;               /* ID of task */
138     const void *codeptr_ra;               /* return address of api call */
139 } ompt_record_scoped_master_t;
140
141 typedef struct ompt_record_scoped_worksharing_s {
142     ompt_worksharing_type_t wstype;        /* loop, sections, single ... */
143     ompt_scope_endpoint_t endpoint;        /* begin or end */
144     ompt_parallel_id_t parallel_id;        /* ID of parallel region */
145     ompt_task_id_t task_id;               /* ID of task */
146     const void *codeptr_ra;               /* return address of api call */
147 } ompt_record_scoped_worksharing_t;
148
149 typedef struct ompt_record_flush_s {
150     void *codeptr_ra;                     /* return address of api call */
151 } ompt_record_flush_t;

```

Listing A.1: OMPT record types [31].