Classification of Common Errors in OpenMP Applications

Jan Felix Münchhalfen[1,3,4], Tobias Hilbrich[2], Joachim Protze[1,3,4], Christian Terboven[1,3,4], and Matthias S. Müller[1,3,4]

[1] IT Center, RWTH Aachen University, D - 52074 Aachen
[2] ZIH, Technische Universität Dresden, D - 01062 Dresden
[3] Chair for High Performance Computing, RWTH Aachen University, D - 52074 Aachen
[4] JARA – High-Performance Computing, Schinkelstraße 2, D – 52062 Aachen
{muenchhalfen,protze,terboven,mueller}@itc.rwth-aachen.de,
tobias.hilbrich@tu-dresden.de

Abstract. With the increased core count in current HPC systems, node level parallelization has become more important even on distributed memory systems. The evolution of HPC therefore requires programming models to be capable of not only reacting to errors, but also resolving them. We derive a classification of common OpenMP usage errors and evaluate them in terms of automatic detection by correctness-checking tools, the OpenMP runtime and debuggers. After a short overview of the new features that were introduced in the OpenMP 4.0 standard, we discuss in more detail individual error cases that emerged due to the `task` construct of OpenMP 3.0 and the `target` construct of OpenMP 4.0. We further propose a default behavior to resolve the situation if the runtime is capable of handling the usage error. Besides the specific cases of error we discuss in this work, others can be distinctly integrated into our classification.

1 Introduction

With the advent of multi- and manycore architectures, node-level parallelization has become increasingly important in the field of high performance computing (HPC). In fact, OpenMP has emerged as the most widely used standard for shared memory parallel programming in HPC. Although the nature of OpenMP programming greatly enhances the development of parallel applications, parallel programming with OpenMP is still error prone to generic mistakes in parallel programming and those specific to OpenMP. In addition, resilience capabilities will become of greater importance with the availability of exascale supercomputers. Parallel programs must be able to detect and respond to certain events that may lead to program termination or incorrect results, e.g., runtime failures. In this context, the development of an error model is a high priority in the OpenMP language committee.

In this work, we propose a classification for OpenMP usage errors (*defects*) to summarize known types of syntactic and semantic mistakes. We further distinguish these from performance issues. Particularly, this includes defect classes that involve the OpenMP 3.0 task and OpenMP 4.0 target constructs. This classification may serve tool developers and the OpenMP community as a framework and overview of the common defects introduced when parallelizing with OpenMP.

L. DeRose (Eds.): IWOMP 2014, LNCS 8766, pp. 58–72, 2014.

Additionally, our examples provide input for test suites that evaluate the correct operation of OpenMP compilers and runtimes. We also investigate the failures that these defects may induce on an application, the possibilities towards automatic detection, and the correctness-checking tools that are capable of detecting them. This classification extends and incorporates existing studies [8,10] and our long term experience in application developer support.

Our investigation first discusses related work (Section 2) and an overview of the latest standard specification released by the OpenMP language committee — OpenMP 4.0 [4] (Section 3).

In Section 4 we discuss our classification of defects, with respect to syntax, semantics and performance. Sections 4.1, 4.2 and 4.3 discuss the individual defect classes in detail and the possibilities for tool developers to detect them automatically. Finally we draw our conclusions in sec. 6.

2 Related Work

Suess et al. discussed common mistakes in OpenMP [10]. They conducted a study over two years and observed which mistakes their students made when parallelizing with OpenMP. A classification and coarse relationship between correctness and performance achievement is made in the paper, and best practices into avoiding common mistakes are presented. Additionally, different compilers and tools are evaluated for their ability to detect certain defects. At the time of this study, OpenMP 2.5 was the most recent standard specification. Thus, it does not include the defect classes that we introduce for the task and target constructs of later OpenMP versions.

Our classification subsumes the defect classes of this study and extends them by syntactic as well as semantic defects the students did not encounter, besides covering newer OpenMP constructs.

P. Petersen and S. Shah also published a classification of threading errors [8] (Section 4, Figure 1). In contrast to our classification, they further distinguish the class of semantic defects, especially the logical defects, into *stalls* and *live-locks*. As we do not see an increased potential of detection out of this differentiation, we summarize these defects in a class of *Conceptual defects*. Apart from that, we subsumed this classification and extend it by classes for syntactic defects and performance issues.

A. Duran et al. proposed an error handling clause [7]. Their proposal draws a callback based mechanism that defines a set of different failure severities and predefines actions to handle specific failures. In case of a failure, the predefined action, or a specified callback is executed and may decide how to handle the failure. This callback can receive additional information about the region- or source code location where the failure occurred. The proposal was evaluated using a modified NANOS [2] OpenMP runtime library. Two benchmark suites, the EPCC micro-benchmarks [6] and the NAS parallel benchmarks [3] (v3.0) are further evaluated to determine which overhead the error handling functionality adds to the OpenMP runtime.

Another work from Wong et al. [11] evaluates three different possibilities to make the OpenMP standard capable of handling erroneous situations. The proposal motivates extended error handling capabilities by demonstrating how C++ exception based

error handling can be used to catch unexpected failures within the OpenMP runtime. Wong et al. then discuss requirements that proposed error handling addition to the standard should conform to. They introduce three different approaches for error handling in OpenMP: a `done` construct, which was included (see *cancellation points*) in the latest OpenMP standard: an error-code dependent approach that would be compatible with languages that are exception unaware; and a callback based approach which slightly extends the one proposed by Duran et al. [7]. For the failure class "*SIMD aligned with unaligned data*" we present in the following, the error handling approaches with error-codes and the `done` construct would be insufficient, the latter at least in exception unaware languages.

Motivated by these works [7][11] there are efforts in the OpenMP language committee to establish error handling capabilities in the OpenMP standard. To support the work of the OpenMP language committee we provide a classification of defects and evaluate possibilities for their automatic detection as shown in several examples. When appropriate, we also include a recommendation on how the failure should be handled by the OpenMP runtime.

3 Overview of OpenMP 4.0

The OpenMP standard is a directive based approach to express parallelism. It uses a strict fork-join model in which multiple threads process *tasks*, which can be either implicit or explicit. The terms implicit and explicit are used to distinguish between a task originating from programmer effort by using the `task` directive, and a task created implicitly when a parallel region is encountered.

Every OpenMP program can be compiled either as a parallel program by interpreting the OpenMP directives and clauses, or as purely sequential by ignoring all OpenMP directives.

The standard does not impose requirements on the behavior of an OpenMP application if it is compiled as a sequential program. The execution results of such a binary can deviate from the parallel results. In other respects there are several requirements to applications that are parallelized with OpenMP, e.g., OpenMP strictly holds on to the Single Entry, Single Exit principle, which means no branch or jump statements from the base language may be used to enter or leave OpenMP regions. Programs which do not adhere to the OpenMP standard's requirements are called *nonconforming*. An OpenMP program starts with one thread that executes the main program on the *host* and subsequently is able to spawn other threads using the corresponding OpenMP directives. Several worksharing constructs then distribute workload among multiple threads, e.g., the *do*, and *for*-loop, and `sections` constructs assign workloads. Whenever a thread is not idle, it can process a task.

In version 4.0, the OpenMP standard distinguishes between *host* and *target* devices. The two terms correspond to host computer and accelerators respectively. Code that is enclosed in a `target` region, is compiled to be executed on accelerators. This extension enables OpenMP to reach out to heterogeneous architectures that previously required different programming models like CUDA, OpenCL, or OpenACC. The new `simd` construct controls execution in the vector units. Other additions to the standard include user defined reductions, task dependency and thread affinity control.

Currently the Intel compiler, the Cray compiler, and the GNU Compiler Collection (GCC) support OpenMP 4.0, although GCC has no functionality implemented yet for the `target` directive and always executes this code on the host.

4 Error Classification

We collected common defects in OpenMP applications from our own experiences, long lasting support activities from various HPC users and our projects, and summarized them in a classification. Figure 1 presents this classification in which we distinguish between syntactic and semantic defects, as well as performance issues. The latter covers performance flaws that do not actually lead to wrong results or nonconforming applications, but affect the efficiency of the application. We will discuss the defects and failure handling, as well as tools that are able to detect them below.

We carefully assembled this classification based on our experiences and available studies [8,10], in order to incorporate all types of defects that we are aware of, as well as novel defects that extensions of OpenMP 3.0 and OpenMP 4.0 introduced. This classification provides a framework for common OpenMP programming defects and forms a basis for future extension.

The naming *error* is ambiguous; therefore we use the notion from [12]:

- *defect* to address programming errors, i.e., incorrect source code, and
- *failure* to address error manifestation, e.g., execution abortion or deadlocks.

To remove a failure from a program a trace-back is needed to the defect in the source code. But there is no distinct correlation between classes of failures and defects, and an error could be assigned to both a failure and a defect class. Compilers and static analysis tools typically detect defects. Runtime analysis tools typically spot failures, but may also spot defects. The same holds for debuggers. When we expect an error to be spotted as a defect we assign it to a defect class. We assign an error to a failure class if we think that this error is only detectable as a failure.

4.1 Syntactic Defects

A syntactic defect in general is code that is not compliant by the grammar of a programming language. With respect to OpenMP parallelization we limit syntactic defects to OpenMP compiler directives. We distinguish between mistyped and correct OpenMP prefixes (e.g. `#pragma omp` in C++); mistyped prefixes are not recognized as OpenMP compiler directives, the compiler will ignore them by default. We do not classify syntactic errors caught by compilers, and refrain from examining them further here. For misspellings, the compiler could recommend similar keywords from the OpenMP grammar.

4.2 Semantic Defects

We recognize programming mistakes that are semantic defects. These are compiled into executable code, but otherwise cause failures within the OpenMP runtime, which may

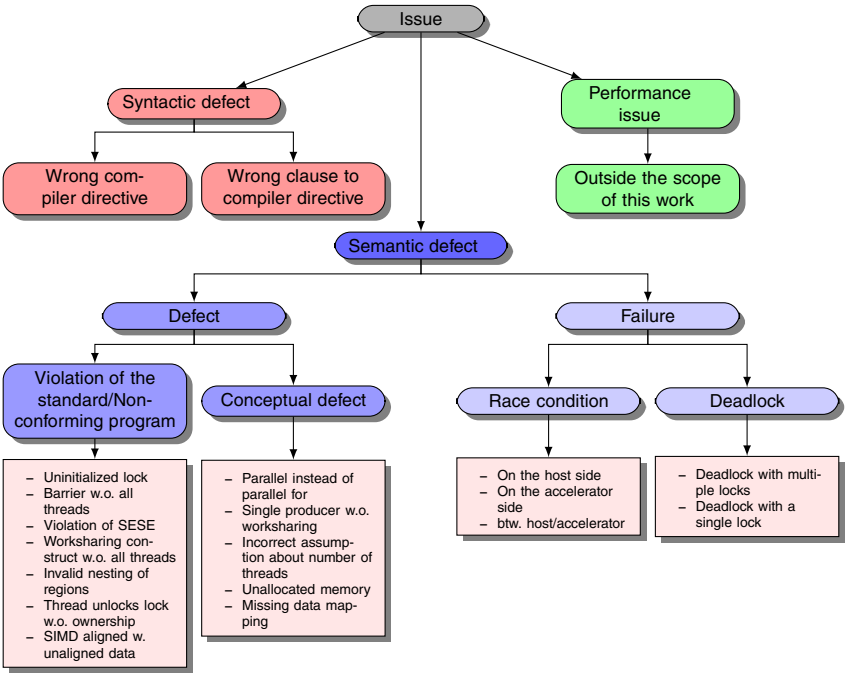

Fig. 1. Classification of Common Issues in OpenMP Applications

manifest themselves as execution aborts, deadlocks, or incorrect results. Most importantly, the exact behavior of a semantic defect depends on the runtime at hand. Thus these defects may introduce portability problems that only become visible with specific runtimes. Semantic defects form the biggest class of possible mistakes — not only when parallelizing with OpenMP — but with programming in general. In the following, we will discuss the four sub-classes of semantic defects in Figure 1. Some of these sub-classes involve additional child classes.

Violations of the Standard/Nonconforming Program: In this class we integrate defects that are a clear violation to one or more of the definitions of the OpenMP standard. The standard strictly prohibits developers from using certain combinations of OpenMP compiler directives or the usage of runtime functions in specific contexts. We do not claim completeness at this point. The amount of possible standard violations simply exceeds the scope of this work, thus we focus on a smaller, respresentative portion of defects.

Uninitialized locks: OpenMP offers different mechanisms to synchronize threads. The most commonly used are *barriers* and *critical sections*. It is more difficult to induce a defect using the latter two high level approaches than doing so with *locks*. Locks are a low level mechanism for synchronization and enable the programmer to coordinate threads with finer and increased control, but also with increased potential to create

defects. OpenMP features two types of locks: regular and nested locks. The difference between the two types is that nested locks may be locked repeatedly by the same thread without blocking, while regular locks would block if the thread that currently owns them tries to acquire the lock again. In both cases a lock needs to be released the same number of times it was locked. Additionally, locks, need to be initialized before they are used. The standard does not specify how programs will behave if programmers make use of an uninitialized lock. Detection of this is either possible in the OpenMP runtime, or by keeping track of initialized locks, e.g., in a correctness tool. The OpenMP runtime could issue a mild severity level, i.e., a warning and handle the failure appropriately by e.g. just initializing the lock to resolve the situation without terminating the application. We do not propose to terminate the application in this case, despite this being a violation to the standard, because the defect is a minor violation and can be resolved quite easily. In our experiments the defect could be found by using Valgrind (memcheck), the Intel Inspector XE or a debugger.

Barriers not reached by all threads of a team: As mentioned previously, barriers are less error prone than locks. Nevertheless they still offer potential for defects as barriers must always be encountered by all threads of a team. Some OpenMP regions have implicit barriers at their end unless a `nowait` clause is present. Thus, this defect may or may not be a special case of *Violation of the Single Entry, Single Exit principle (SESE).* It is possible to induce this situation by using barriers inside conditional statements which depend on a condition that is special to a thread, or by implementing irregular execution patterns using statements such as `goto`. An example of this situation for a barrier inside a conditional statement, is given in code Example 1.

If the code is of low complexity and only includes a few conditional statements that determine if a thread reaches a barrier or not, the defects of this class may be detectable by static code analysis. In more complex codes, e.g., where the barrier is hidden in a third party library, the detection proves far more difficult. A runtime analyzer may identify this situation if the waiting states of all threads are clearly known, e.g. if all threads wait at different barriers. More exhaustive approaches could utilize model checking techniques to detect defects of this class. If this situation can be detected, a runtime may decide to terminate the application, since the program exhibits a serious nonconformance to the standard.

Violation of the Single Entry, Single Exit principle (SESE): The OpenMP execution model requires each thread encountering an OpenMP region to also exit the region in a regular fashion (without skipping the end of the region). A team of threads that is spawned at the start of a `parallel` region must correctly exit the parallel region. Parallel execution in OpenMP strictly follows a fork-join-model and consequently the generated team threads must join at the end of a parallel region. Any deviation from this principle violates the OpenMP standard and will cause undefined behavior ranging from program termination to deadlock, as well as partially executed worksharing directives. Detection is possible through static code analysis or by keeping track of individual threads and their paths of execution during runtime. The default behavior of the runtime should be immediate termination of the application or cancellation of all `parallel` regions because the program behavior is undefined from this point on.

Worksharing constructs not reached by all threads of a team: Special constructs like worksharing constructs must be encountered by all threads of a team or none at all. As described in *Violation of the Single Entry, Single Exit principle (SESE)*, these situations are created by using the `goto` keyword or by placing worksharing constructs inside conditional code paths. An example is given in code Example 1. The most promising approach to detect this kind of defect is static code analysis. If the runtime is able to detect this automatically, it should terminate the innermost `parallel` region with an appropriate error level because the successful parallel execution of this region is then rendered impossible. This would enable developers to implement different strategies and, if necessary, rely on (possibly serial) fallback algorithms to deliver correct results.

```
1 #pragma omp parallel
2     if( omp_get_thread_num() % 2 ){
3         #pragma omp for
4         for( int i=0; i < N; ++i )
5             ...
6         #pragma omp barrier
7     }else{
8         #pragma omp barrier
9     }
```

Example 1. Worksharing construct/barrier is not reached by all threads of a team

```
1 double a[N], b[N], c[N];
2 ...
3 #pragma omp parallel
4     for(int i=0; i < N; i++)
5         a[i] = b[i] * c[i];
```

Example 2. Parallel for should be used

Invalid nesting of regions: The current OpenMP standard [4] covers the nesting of regions and places a set of restrictions on which regions may not be directly nested. For example it is strictly forbidden to directly nest worksharing constructs, because each worksharing construct requires the context of a single `parallel` region. Any violation to this restriction fits into this defect class. The compiler is in many cases able to identify this kind of defect. An exception to this is when OpenMP regions are *orphaned* and hidden inside third-party libraries or subroutines in other source files. This effectively prevents some compilers from identifying a relationship between the OpenMP region and its parent region. In this case, identifying the violation to the standard is far more difficult and requires more sophisticated approaches such as modifications to the OpenMP runtime to keep track of which regions are reached by individual threads. Termination of the innermost parallel region may in some cases suffice to handle the failure, but that depends on the combination of regions which violate the standard and it may as well be required to terminate the application if e.g. `critical` regions of the same name are nested (see *Deadlocks*). While this would render further execution impossible, termination of the innermost parallel region on the other hand may enable developers to implement recovery mechanisms, e.g., a fallback approach. Compilers with interprocedural analysis capabilities will most probably be able to even detect invalid nesting of *orphaned* constructs properly.

Unlocking locks that are not owned by the current task: The OpenMP standard states that calls to `omp_unset_lock` and `omp_unset_nest_lock` with a lock as argument that is neither locked nor owned by the current *task* is nonconforming. It is easy to avoid this kind of situation when using OpenMP locks: Using higher level approaches, e.g., C++ classes that lock in the constructor and unlock in the destructor, or always placing the lock and unlock in pairs, and near to each other in the source code. When tasks are tied to threads in OpenMP (default behaviour), the constraint that the lock needs to be owned by the current task is equivalent to the constraint of lower-level mechanisms like pthreads, that the current threads need to own the lock. So far, none of the major compilers mentioned in Section. 3 have implemented this clause, but this possibly might create problems when locks are used inside untied tasks. Unlocking attempts by threads/tasks which don't own the lock can easily be detected by keeping track of the locking and unlocking operations in the runtime. As the program state is undefined after this operation, the runtime should either terminate the program if this failure occurs or exit all parallel regions with a descriptive error code.

SIMD aligned with unaligned data: OpenMP 4.0 includes a `SIMD` directive that enables programmers to manually instruct the compiler to translate a loop into vectorized code. An `aligned` clause can be used to specify the alignment of loop data in bytes. This may be beneficial because some instruction sets, e.g., the x64 instruction set, include SIMD instructions for aligned and unaligned data, of which the former generally needs less loads to complete. The developer can align data either by using special `malloc`-variants like `posix_memalign` or by manually allocating $(N + \textit{alignment})$ bytes and then adjusting the pointer appropriately.

The task of detecting the incorrect use of the `aligned` clause can be very difficult at compile time as it is not known then how the data will be aligned when the program is run. A debugger will certainly detect this (after execution) as the CPU will raise an interrupt when this defect occurs. Because the runtime will most probably not be able to detect this kind of defect before an interrupt occurs, we do not propose a default behavior here.

Conceptual Defect: A *conceptual defect* occurs when code does not explicitly violate the OpenMP standard and may as such be called a conforming OpenMP program, but nevertheless results in unwanted or unintended program behavior. Cases of this class represent correct applications in terms of the restrictions of the OpenMP standard, but fail to meet intended specifications, due to another software defect or due to an incorrect specification. As an example, an application meant to calculate π could be correctly parallelized, but simply have a defect in its formulas or logic. It may be either caused by a program condition that only occurs under specific circumstances that were not considered by the developer or simply by lack of knowledge on the developer. We explicitly exclude race conditions (see *Race Conditions*) and deadlocks (see *Deadlocks*) from this class.

Parallel instead of parallel for: The mistake of using a `parallel` directive when a `parallel for` directive was intended is quite common and can lead to different

failures, depending on the pre-parallized code. This defect may not have any impact on compilation at all and thus go unnoticed. This is the case, e.g., when all threads execute the same code in parallel, without any worksharing or data races. The result will be correct, but there will be no benefit in runtime because each thread does the same work as the serial program. See Example 2 for this case. Because there are situations where it actually may make sense to use `parallel` over `parallel for`, this defect is hard to detect. If data races occur during execution of this defect, the defect is detectable by applying detection methods for race conditions (see *Race Conditions*). If no race conditions occur during execution of this defect we do not think a tool would be able to detect it.

Single producer without worksharing: An often used concept in parallel programming is the *single producer*. In the single producer pattern one thread creates one or multiple tasks inside a `single` construct. Subsequently those tasks are processed by the threads of the underlying parallel region. Because the absence of the single construct does not violate the standard or renders the code syntactically incorrect, it is easily overlooked. Another well known design pattern is the *parallel producer*. As the name suggest, here tasks are created in parallel and possibly without a `single` construct. Hence it is not easy to determine the correctness of applications which utilize these parallel programming patterns. They might however trigger other defects which are covered in this work, e.g. *Race Conditions*. We do not consider this sort of mistake to be detectable by correctness checking tools because the intention of the application remains ambiguous without deeper insight.

Locks as barriers: An OpenMP lock that was locked by one thread will block all other threads that call `omp_set_lock` until the thread which owns the lock executes a call to `omp_unset_lock`. Therefore, a programmer with a lack of understanding of OpenMP constructs might try to use locks as a concept for barriers, unaware of the `barrier` directive and assuming that a single call to `omp_unset_lock` enables the continuation of all threads. We do not expect that the intention to create a barrier is detectable by a tool when this situation is encountered. Potentially, a deadlock or a possible deadlock can be detected and does not differ from the situation in the subsequent deadlock defect class.

Incorrect assumptions about the number of threads: The `OMP_DYNAMIC` environment variable may raise unexpected situations in OpenMP programs if set to true and the developer relies on the number of threads being set by other methods, such as a previous argument to `omp_set_num_threads` or the value of the environment variable `OMP_NUM_THREADS`. While this situation does not interfere with dynamic loop scheduling, it is probably best to allow the runtime to choose the number of threads arbitrarily to optimize the utilization of system resources. Therefore the programmer may not rely on the number of threads being equal to what was requested. Even if a certain number of threads was requested and the value of `OMP_DYNAMIC` equals to false, a different number of threads may be provided by the runtime, e.g., if less threads are available than what was requested. This kind of defect is logical nature and thus very

difficult to spot automatically because an analyzing application needs to know what is to be accomplished in the first place. The runtime therefore cannot detect it.

Unallocated memory (host/accelerator): If an OpenMP application fails to allocate memory and thus uses a `NULL`-pointer in a subsequent OpenMP clause, the behavior of the runtime library is undefined. This sort of defect could be handled by a mechanism like the one proposed in [7,11]. The runtime should take care of the error handling in this case. To resolve the failure, the region should either be completely skipped with an adequate error code given back to the application developer — or transfer execution to a callback function that could be specified by the developer. While `NULL`-pointers are quite easy to detect, a pointer to a memory region that is not allocated or only partially allocated may prove more difficult to spot. To achieve this, the runtime would need to track the application heap, or all calls to `malloc`. This erroneous situation is also detectable using a debugger: An application raises a `SEGFAULT`-interrupt when it accesses unallocated memory in any way. The proposed way to handle this failure is either program termination or termination of the innermost parallel region with a suitable error code.

Missing data mapping to accelerator: In OpenMP 4.0 the `target` and `target data` directives were introduced to that enable the programmer to offload computations to accelerator devices. The directives support a `map` clause that is responsible for the mapping (or migration) of data from the host to the accelerator and vice versa. The developer has to name each variable that was declared outside the scope of the `target` region in a corresponding `map` clause. Depending on the type of accelerator, host and device may either share their memory, portions of their memory or have separate memories. Therefore it is important to specify the type of mapping in the `map` clause which may either be `alloc`, `to`, `from` or `tofrom`. Global variables inside a `target` region must be declared with the `declare target` directive. If the accelerator and host do not share a common memory, on the device side the data will be allocated, received from the host and copied back, corresponding to their mapping. If a data mapping for the `target` region is missing, the compiler will raise an error and most likely interrupt the compilation process.

Race Conditions: A race condition is defined as a situation during program runtime in which two or more executing units, usually threads, access a shared resource simultaneously in such a way that at least one unit is modifying the resource. This may lead to failures, if the resource does not provide sufficient internal synchronization to allow for multiple executing units/threads accessing it at the same time, e.g. a memory location. This usually requires the programmer to employ suitable synchronization. To this class we assign all defects which can lead to race conditions. Detecting this class of defects usually induces a significant overhead on application runs. We therefore do not consider this class recommendable to be handled by the OpenMP runtime automatically.

Race conditions on the host side: Race conditions are a problem not only specific to OpenMP but to parallel programming in general. OpenMP parallelized applications which do not make use of the `target` directive to offload data for computation to an accelerator, may execute with false assumptions about the default data sharing behavior of OpenMP or forgotten data sharing clauses when entering a parallel region. There are several works in progress on the detection of data races on the host side, e.g. the thread sanitizer that is part of the latest version of GCC [9] and commercial tools like the Intel Inspector XE [1] and Oracle Solaris Studio Thread Analyzer [5]. Most of them are able to properly detect those types of data races.

Race conditions on the accelerator side: If an application with target directives is compiled to offload code to an accelerator, OpenMP directives and runtime functions may be utilized on the device as well as on the host with few limitations [4]. The developer may spawn a team of threads on the accelerator using the `parallel` directive — same as on the host side — and forget to specify adequate data sharing clauses for the variables used inside the parallel region. In principal the problem is very similar to the one described in *Race conditions on the host side*, but its detection can be much more complicated because not all accelerator systems offer the same repertoire of debugging/tracing capabilities as most host processors. Additionally, detecting race conditions usually requires a lot of device memory because each memory access needs to be traced for later analysis. On accelerator devices this may prove challenging, as memory is generally very limited. An approach for GPGPU accelerators [13] provides insight into such data race detection capabilities, but has not yet been applied to OpenMP applications.

Race conditions between host/accelerator: With the introduction of the `target` directive it is possible to offload computations to an accelerator device. As explained in *Missing data mapping to accelerator* it is further possible that the host and the accelerator truly share memory. In this case, race conditions will occur if both the host and the accelerator device operate on the same data without synchronization. As a consequence, data is copied from and to the accelerator as `target` or `target data` regions are encountered. While the OpenMP runtime should take care that no data is overwritten when data is copied to the accelerator, it is very well possible that the host changed the data that was transferred to the accelerator in the meantime. A copy-back operation from the accelerator may overwrite this data and lead to a data race if the user has not synchronized the access. If host and accelerator share a common memory, the detection of this requires tracing both host and device memory accesses because both can modify the same memory transparently. If host and accelerator have seperate memories, it is sufficient to trace only host memory accesses because one thread will always block until the target region is completed and only then, copy back data (see *Race conditions on the host side*).

Deadlocks: A deadlock is a situation in which a program is in a waiting state for an indefinite amount of time. In this class we categorize defects which can lead to deadlocks. If the OpenMP runtime is able to identify a deadlock situation, it should handle the failure by either terminating the application or, more preferably, by terminating all

`parallel` regions with an error code. Unfortunately, not all deadlock situations will be detectable by the runtime. OpenMP 4.0 introduced the feature of task dependencies which we evaluated with regard to their potential of introducing deadlocks. Task dependencies by design prevent programmers from creating circular dependencies and therefore it is effectively impossible to run into a deadlock by only using task dependencies.

Deadlock with multiple locks: The explicit use of the locking API is in general error-prone and susceptible to deadlocks. Especially when more than one lock is involved, deadlocks can occur if locking operations overlap. In a real world application this may be difficult to detect at compile time if calls to `omp_set_lock` and `omp_unset_lock` are concealed in subroutines (*orphaned*). If all threads are blocked due to a call to `omp_set_lock`, the deadlock situation is clear and can be identified by the OpenMP runtime or correctness-checking tools.

Deadlock with a single lock: This type of defect describes a situation where a deadlock occurs due to the order in which locks are accessed. The deadlock situation might not always occur but only under specific circumstances, e.g., specific scheduling of threads/tasks. In Example 3 it very much depends on the scheduling of tasks if the application will deadlock or not. If two or more tasks operate on locks concurrently and there is a task scheduling point after a call to `omp_set_lock` without a preceding `omp_unset_lock`, a deadlock might occur. This situation is also called a *potential* deadlock. A tool could spot this situation via a model checking approach or at runtime by keeping track of locks and their owners and `taskwait` constructs. We are not aware of any tools which are currently able to spot such failures.

```
1  omp_lock_t lock; omp_init_lock( &lock );
2  #pragma omp parallel
3      #pragma omp master
4          #pragma omp task
5          {
6              #pragma omp task
7              {
8                  omp_set_lock( &lock );
9                  omp_unset_lock( &lock );
10             }
11             omp_set_lock( &lock );
12             #pragma omp taskwait
13             omp_unset_lock( &lock );
14         }
15 omp_destroy_lock( &lock );
```

Example 3. Possible deadlock due to a race on a lock.

4.3 Performance Issues

Because the performance issues named in Figure 1 are not specific to OpenMP programming, we will not include a more detailed specification of them in this work. We

do not discourage the usage of critical sections and locks, but recommend that developers consider the amount of work that is done inside synchronization constructs. Depending on this, the other threads will be blocked during that time, or the overhead of synchronization might have a significant impact on the overall runtime. Furthermore, pointless flushing, as well as memory access patterns which will lead to cache trashing or false sharing, will probably lead to bad application performance and are thus best to avoid.

5 Summary

We evaluated each error class in terms of detectability in the above table (2). In this evaluation we distinguished between different methods to detect defects, such as compilers, compiler based static analysis (involving interprocedural analysis), the OpenMP runtime, debuggers and tools which conduct correctness checks at runtime. A cross ($\boxed{\bullet}$) marks that the tool class is able to detect the defect, a cross in round brackets ($\boxed{(\bullet)}$) means that the tool class is able to detect the error under special circumstances.

#	Mistake	Compiler	Static Analysis	Runtime	Debuggers	Tools
	Syntactic mistakes					
1.	wrong_directive		•			
2.	wrong_clause	•	•			
	Semantic mistakes					
	Violation of the standard					
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		(•)		•	
	Conceptual defect					
10.	parallel_inst_parallel_for					
11.	single_prod_wo_worksharing					
12.	number_of_threads					
13.	unallocated_memory	(•)	•	•	•	•
14.	missing_data_mapping	(•)	•			(•)
	Race condition					
15.	host			(•)		•
16.	accelerator			(•)		•
17.	host_accelerator					(•)
	Deadlock					
18.	multiple_locks		(•)			•
19.	single_lock		(•)			•

Fig. 2. Defects and their detectability

6 Conclusion

We developed a classification of OpenMP defects for the OpenMP 4.0 standard. In this classification we distinguished errors by both the defect and the failure. We further summarized many defects we consider common to OpenMP programming and evaluated them in terms of their potential for automatic detection by analysis tools. If we considered the runtime able of handling a failure, we propose a default error handling mechanism that is to be executed when application developers do not specify error handlers of their own. We found that the number of common defects in OpenMP did not decrease with newer versions of the standard, but slowly increased due to new features that were added to the standard. An example of this is the `target` construct which was introduced with OpenMP 4.0.

Future work will explore possible collaborations with developers of specific scientific domains to collect their mistakes and to quantify the frequency of the error classes we listed in this work.

Acknowledgement. Parts of this work were funded by the German Federal Ministry of Research and Education (BMBF) under Grant Number 01IH13008A (ELP).

References

1. Intel Inspector XE (2013), `https://software.intel.com/en-us/intel-inspector-xe`
2. NANOS Project, `http://www.cepba.upc.edu/nanos/`
3. NAS Parallel Benchmarks, `https://www.nas.nasa.gov/publications/npb.html`
4. OpenMP 4.0 specification (July 2013), `http://openmp.org/wp/openmp-specifications/`
5. Oracle Solaris Studio, `http://www.oracle.com/technetwork/server-storage/solarisstudio/documentation/index.html`
6. Bull, J.M.: Measuring Synchronisation and Scheduling Overheads in OpenMP. In: Proceedings of First European Workshop on OpenMP, pp. 99–105 (1999)
7. Duran, A., Ferrer, R., Costa, J.J., Gonzàlez, M., Martorell, X., Ayguadé, E., Labarta, J.: A Proposal for Error Handling in OpenMP. In: Mueller, M.S., Chapman, B.M., de Supinski, B.R., Malony, A.D., Voss, M. (eds.) IWOMP 2005/2006. LNCS, vol. 4315, pp. 422–434. Springer, Heidelberg (2008)
8. Petersen, P., Shah, S.: OpenMP Support in the Intel® Thread Checker. In: Voss, M.J. (ed.) WOMPAT 2003. LNCS, vol. 2716, pp. 1–12. Springer, Heidelberg (2003)
9. Serebryany, K., Iskhodzhanov, T.: ThreadSanitizer: Data Race Detection in Practice. In: Proceedings of the Workshop on Binary Instrumentation and Applications, WBIA 2009, pp. 62–71. ACM, New York (2009)
10. Süß, M., Leopold, C.: Common Mistakes in OpenMP and How to Avoid Them: A Collection of Best Practices. In: Mueller, M.S., Chapman, B.M., de Supinski, B.R., Malony, A.D., Voss, M. (eds.) IWOMP 2005/2006. LNCS, vol. 4315, pp. 312–323. Springer, Heidelberg (2008)

11. Wong, M., Klemm, M., Duran, A., Mattson, T., Haab, G., de Supinski, B.R., Churbanov, A.: Towards an Error Model for OpenMP. In: Sato, M., Hanawa, T., Müller, M.S., Chapman, B.M., de Supinski, B.R. (eds.) IWOMP 2010. LNCS, vol. 6132, pp. 70–82. Springer, Heidelberg (2010)
12. Zeller, A.: Why Programs Fail: A Guide to Systematic Debugging. Morgan Kaufmann Publishers Inc., San Francisco (2005)
13. Zheng, M., Ravi, V.T., Qin, F., Agrawal, G.: GMRace: Detecting Data Races in GPU Programs via a Low-Overhead Scheme. IEEE Trans. Parallel Distrib. Syst. 25(1), 104–115 (2014)