OpenMP Tools Interface: Synchronization Information for Data Race Detection

Joachim Protze[1,2], Jonas Hahnfeld[1,2], Dong H. Ahn[3], Martin Schulz[3], and Matthias S. Müller[1,2]

[1] RWTH Aachen University, D-52056 Aachen, Germany
[2] JARA – High-Performance Computing, D-52062 Aachen, Germany
{protze, hahnfeld, mueller}@itc.rwth-aachen.de
[3] Lawrence Livermore National Laboratory, Livermore, CA 94550, USA
{ahn1, schulzm}@llnl.gov

Abstract. When it comes to data race detection, complete information about synchronization, concurrency and memory accesses is needed. This information might be gathered at various levels of abstraction. For best results regarding accuracy this information should be collected at the abstraction level of the parallel programming paradigm. With the latest preview of the OpenMP specification, a tools interface (OMPT) was added to OpenMP. In this paper we discuss whether the synchronization information provided by OMPT is sufficient to apply accurate data race analysis for OpenMP applications. We also present some implementation details for our data race detection tool called Archer which derives the synchronization information from OMPT.

1 Introduction

OpenMP is the de facto standard for parallel programming on shared memory machines. It is also becoming increasingly popular on extreme-scale systems as it offers a portable way to harness the growing degree of parallelism available on each node. However, porting large HPC applications to OpenMP often introduces subtle errors. Of these, data races are particularly egregious, as well as challenging to identify. Data races may remain undetected during testing, but nevertheless manifest during production runs by often resulting in confusing (and/or non-reproducible) executions that the programmer wastes considerable amounts of time debugging. In extreme situations, data races may simply end up silently corrupting user data. For all these reasons, data race detection remains one of the central concerns in parallel programming, in particular for shared memory programming models.

In previous papers [1,5], we presented the tool ARCHER, based on ThreadSanitizer [6,7], which is able to find data races in OpenMP applications, that are run with the LLVM/OpenMP runtime on x86 machines. The fact which makes this tool unique from other approaches of available data race detection tools for OpenMP applications is that we cover almost all host-side OpenMP directives as provided in the OpenMP 4.5 specification. To make the tool portable across OpenMP runtime implementations and hardware platforms, we want to base the annotation of OpenMP synchronization on OMPT events.

In this paper we investigate whether the information provided by OMPT is sufficient to derive all OpenMP synchronization semantics. We will describe OMPT based annotations of OpenMP synchronization. The annotations are provided as happened-before arcs, which can be understood by ThreadSanitizer, but also by the Valgrind based data race detection tool Helgrind. This approach is portable across OpenMP runtime implementations, as long as these implement and provide the necessary OMPT callback function invocations.

In Section 2 we look at OpenMP directives with synchronization semantics from a happened-before point of view. In Section 3 we describe the OMPT events, that we use to annotate the synchronization and how we specify the happened-before arcs. In Section 4 we discuss challenges we encountered on the way, implementing the tool and discuss information missing in the OpenMP tools interface.

2 Synchronization in OpenMP

According to the OpenMP specification [2]: "... if at least one thread reads from a memory unit and at least one thread writes without synchronization to that same memory unit [...], then a data race occurs. If a data race occurs then the result of the program is unspecified."

To enable a data race detection tool to identify a data race, complete understanding of synchronization is needed. In this section we provide a summary of the synchronization concepts in OpenMP, as they need to be understood by an analysis tool, to identify synchronized memory accesses. In this paper we focus on data races that happen between threads on a host device. Thus, we do not consider constructs for offloading to an accelerator device.

2.1 The `parallel` Construct

When a thread encounters a parallel construct, the thread creates a team of threads to execute the parallel region. Each thread of the team executes the structured block of the parallel region within an implicit task.

Encountering the parallel construct *happens before* the execution of all implicit tasks of the team.

There is an implicit barrier at the end of the parallel region, which *happens before* the master thread continues execution.

2.2 The `barrier` Construct

The barrier in OpenMP applies for the innermost parallel team. On encountering a barrier construct, a thread cannot continue executing the implicit task until all threads in the team reached the barrier.

For all threads in the team, encountering the barrier construct *happens before* they continue execution of the implicit task.

2.3 The `reduction` Clause

The reduction clause provides a mechanism to reduce results at the end of a work-sharing region into a single value. The clause takes a reduction identifier to specify the reduction operation, the synchronization of the reduction is provided by the OpenMP implementation.

If no **`nowait`** clause is used on the same construct, the reduction *happens before* the end of the region. Otherwise the reduction *happens before* the next barrier.

2.4 The `critical` Construct

The critical construct provides mutual exclusion for the critical region. The critical construct can have a name, that provides mutual exclusion only for critical regions with the same name. The critical region is equivalent to getting a lock at the begin of the region and releasing the lock at the end, with different locks for different names and an extra lock for all unnamed critical regions. Thus, the synchronization semantics are the same as for Locking routines.

2.5 Locking Routines

OpenMP provides routines to init, destroy, acquire and release locks and nested locks. Locks provide mutual exclusion for code between acquiring and releasing a lock.

As a strict measure, a lock-set algorithm can be used to express the synchronization of critical region and locking routines. But lock-set is in general too strict and can lead to false positives The reason is that an application might implement *happens before* semantics in the locked sections. The alternative is to express locks with a happens before relation:
Releasing a lock *happens before* acquiring the same lock.

This might over-estimate the synchronization semantics of the application and lead to omission of actual data races. This is a point, where large numbers of repetition and concurrency helps to stochastically execute the right interleaving of locked regions, so that the race can still be observed.

2.6 The `ordered` Construct

The ordered construct provides mutual exclusion for the ordered region. Additionally, the ordered construct also provides an ordering for the execution.

Thus, when observing the execution of an OpenMP program, the end of an ordered region *happens before* the begin of the next iteration of the same ordered region.

2.7 The `task` Construct

When a thread encounters a task construct, the thread generates a task from the associated structured block. The thread might execute the thread immediately, or defer the task for later execution.

Encountering the task construct *happens before* the execution of the task. The end of a task region *happens before* the next barrier of the team finished synchronization. Without further clauses or constructs, there is no more synchronization at the end of a task.

2.8 The `taskwait` Construct

The taskwait construct lets the encountering task wait for completion of all direct child tasks that this task created before encountering the taskwait.

Finishing all child tasks *happens before* the taskwait regions ends and the task can continue execution.

2.9 The `taskgroup` Construct

The taskgroup construct lets the encountering task wait at the end of the task group region for completion of all child tasks this task created in the taskgroup region and their descendants

Finishing all child and descendant tasks *happens before* the taskgroup regions ends and the task can continue execution.

2.10 The `depend` Clause

The depend clause provides synchronization for task as the provided *in*, *out*, and *inout* dependencies define constraints for the scheduling of tasks. A depend clause can have a list of storage locations, which describe *in* or *out* dependencies. The end of a task with an *in* dependency on a storage location `x` *happens before* the start of any task with an *out* or *inout* dependency on the same storage location `x`. The end of a task with an *out* or *inout* dependency on a storage location `x` *happens before* the start of any task with an *in*, *out*, or *inout* dependency on the same storage location `x`.

To summarize, only *in* dependencies with the same storage location `x` do not synchronize. All other dependencies with the same storage location `x` synchronize.

2.11 Untied Tasks

Deferring a task *happens before* scheduling the same task again. This is especially important for untied tasks, that can migrate from one thread to another thread after being deferred during execution.

2.12 The `flush` Construct

The flush construct makes a thread's temporal view of memory consistent with memory and enforces a specific ordering of memory operations. The flush construct takes an optional list of variables, the *flush-set*. With the right combination of loads, stores and flushes, an application programmer can achieve fine-grain synchronization. Modeling the semantics of flushes with plain happens-before relation introduces synchronization which possibly hides any data race. A better approach for handling flushes is discussed by Lidbury and Donaldson [4]. They extend ThreadSanitizer to understand and handle C++11 flush semantics.

3 OMPT Events for Synchronization

In this section we explain the synchronization events provided by the OpenMP tools interface as it is integrated into the preview of the OpenMP specification 5.0 [3]. Since

we implemented our prototype along with the LLVM/OpenMP runtime implementation, we used the version of OMPT, that is implemented there. The latest specification of OMPT describes events as points of interest in the execution of a thread. Tool callback functions are implemented in a tool and invoked by the runtime when a matching event happens. Multiple events might trigger the same callback; the tool can differ the events by some *kind* and *endpoint* arguments provided with the callback invocation. On tool initialization the OpenMP runtime implementation provides information to the tool, whether requested callback invocations are provided or not. For some groups of events invocation is mandatory, for some it is optional.

3.1 Team related OMPT Events

The following events mark the synchronization points for a team from the creation of the team to the end:

- *parallel-begin*
- *implicit-task-begin*
- *barrier-begin*
- *barrier-end*
- *implicit-task-end*
- *parallel-end*

On a *parallel-begin* event, we generate a new team information object and start a happened-before arc for the team.

On an *implicit-task-begin* event, we generate a new task information object and end the happened-before arc for the team. This synchronizes the team creation.

On a *barrier-begin* event, we start a happened-before arc on an address from the team's information object. This event is specified to happen before the actual synchronization of the barrier.

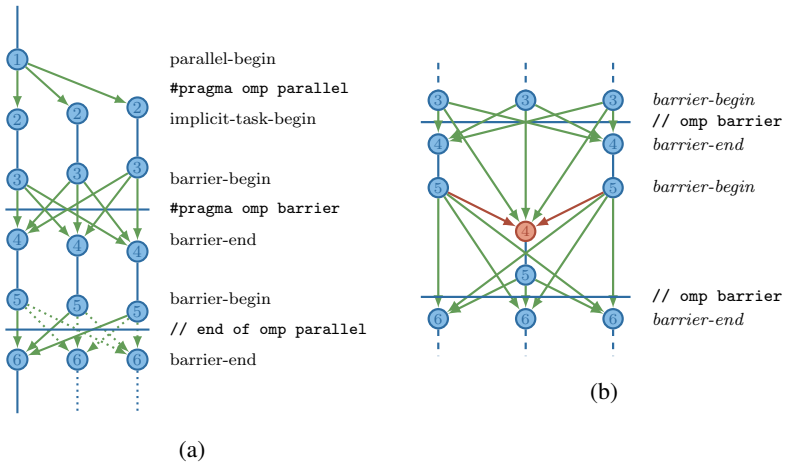

Fig. 1: a) Happens-before arcs in a parallel region with explicit barrier and implied barrier at the end. b) If a thread returns late from the barrier code (red *barrier-end* (4)), others might be already in the next barrier. In this case, we would add wrong happens-before arcs, if all barriers use the same token for the happened-before annotation

On the *barrier-end* event, we end the happened-before arc on the same address from the team's information object. Since there is no synchronization between the barrier end event and the next barrier begin event, it is possible as depicted in Figure 1b, that a thread of the team reaches the next barrier before another thread finished the previous barrier. Therefore, consecutive barriers should use distinct synchronization tokens. The OpenMP specification states that all threads in a team need to participate on each barrier, so we use two addresses for barriers in the team information object and each implicit task toggles between the two addresses.

The parallel region ends with an *implicit-task-end* event and a *parallel-end* event where we free the task and team information objects. The synchronization at the end of the region happens solely in the implied barrier at the end of the region. This is the second barrier in Figure 1a.

As a missing piece in OMPT we will discuss the OpenMP reduction clause in Section 4.

3.2 Task related OMPT Events

The following events mark the synchronization points for a task from the creation of a task to the end:

- *task-create*
- *task-dependences*
- *task-schedule*
- *taskwait-end*
- *taskgroup-begin*
- *taskgroup-end*

On a *task-create* event, we generate a new task information object and start a happened-before arc for the generated task. This synchronizes the task creation with the execution of the task. If this event is invoked before all data are copied to the task data structures, there might be some false data race alerts. Especially the copying of first-private data, which is then accessed by the task, might be a problem. See Figure 2 for an illustration of the task-related events and happened-before synchronization.

On a *task-dependences* event we save all dependences information into the task information object for later use.

On the first *task-schedule* event for a new task, we end the happened-before arc from the generation of the task. Further, we iterate over all task dependences and end happened-before arcs for all dependences. If the dependency is an *in* dependency, we only end happened-before arcs from *out* or *inout* dependencies on this storage location. If the dependency is an *out* or *inout* dependency, we end happened-before arcs from all dependencies on this storage location. See Figure 3 for an illustration of the dependencies-related events and happened-before synchronization. This also highlights the necessity to store the dependency information from task creation until task completion.

If the `prior_task_status` signals completion of the previous task, we start happened-before arcs for the completed task:

- towards a potential `taskwait` of the parent task
- if the task is in a `taskgroup` towards the end of the taskgroup
- if the task has dependencies, an arc per dependency.

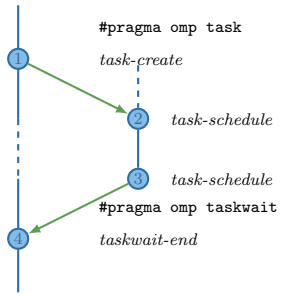

Fig. 2: Execution of a task happens after the task was generated from the parent; in case the parent task does a taskwait, the taskwait finishes after the generated task finished; end of taskgroup is similar

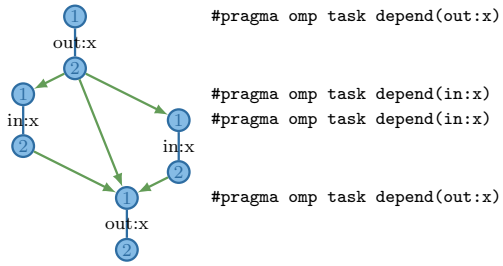

Fig. 3: This is the dependency graph for a set of tasks which were created with *out*, *in*, *in*, and *out* dependency on `x`; the end of a task with *out* dependency happens before all *task-begin* of tasks with a dependency on the same address. Tasks with the same *in* dependency run concurrently.

On a *taskwait-end* event, we end the happened-before arc from all child tasks. We use a common token for all child tasks, so this is a single operation.

On a *taskgroup-begin* event, we push a taskgroup information object on the taskgroup stack of the encountering task. The stack is necessary because multiple taskgroup regions might be closely nested within a task. All child tasks inherit the taskgroup stack on task generation, so they know about their enclosing taskgroup.

On a *taskgroup-end* event, we end the happened-before arcs of all child tasks, targeting to the taskgroup end. Then we pop the taskgroup from the stack of taskgroups.

3.3 Locking related OMPT events

The following events mark the begin and end of mutual exclusion:

- *acquired-lock*
- *acquired-nest-lock-first*
- *acquired-critical*
- *acquired-atomic*
- *acquired-ordered*
- *released-lock*
- *released-nest-lock-last*
- *released-critical*
- *released-atomic*
- *released-ordered*

The latest OMPT specification consolidates all above events into a single callback for acquired and released with a **`kind`** argument for the kind of synchronization. For the happened-before synchronization, we only use the **`wait-id`** argument, so the handling of events is symmetric for all kind of mutex events.

On an *acquired* event, we end a happened-before arc, that starts on a previous *released* event.

To represent the synchronization semantics of locks in a data race analysis, it is important to start and end the happened-before arc inside of the locked region. Otherwise, another thread might already enter a locked region, before the released information is available. To reduce the potential overhead of an OMPT tool, the *released* event is invoked after the lock was released and there is no *releasing* event in OMPT. We discuss in Section 4.1 how we worked around this issue.

3.4 OMPT flush event

The flush event doesn't fit into the semantics of the previously discussed event groups. As touched in Section 2.12, happened-before semantics are too strict. But omitting the handling of flush, we experience false reports on data races in applications that use flush for synchronization. Implementing the right semantics for flush in our tool is subject of future work. But for now, we found that the information provided by the flush event is not sufficient for data race analysis as we will discuss in Section 4.5.

3.5 Team and Task Information Structures

We create an information object for each team and each task, which we store in the runtime scope of this team or task using the **`parallel_data`** and **`task_data`** fields provided by OMPT. In this section we detail on the necessary members of these objects. Both kinds of objects contain tokens, that we use to annotate different synchronization points.

A team object contains

- two tokens for **`barriers`**, the tasks of the team use them alternating; we also use one of the tokens for the fork of the team.

A task object contains

- a token for the **`task`**, that is used for the annotation, task-create before task-execution and task-deferring before rescheduling,
- a token for **`taskwait`**, which is used to annotate synchronization between the end of all child tasks and the taskwait,
- a **`barrier index`**, that toggles between odd and even barrier count,
- a **`reference count`** for direct child tasks, the object is only freed when the task and all child tasks finished execution,
- a reference to the **`parent`** task object,
- a reference to the **`implicit`** task object in the stack next to this task,
- a reference to the currently active **`taskgroup`** object,
- a copy of the *list of dependencies* and a *dependency count*,

– address and size of task private memory.

A taskgroup object contains

– a token for the *taskgroup*
– a reference to the enclosing taskgroup

4 Implementation Challenges and OMPT Shortcomings

In this section we discuss challenges, potential pitfalls and open issues which we encountered implementing the synchronization annotations in an OMPT-based tool.

4.1 Annotation of Locking

For TSan a happened-before annotation consists of writing memory at the start of the happens-before arc and reading the memory at the end of the arc. If the memory access is not synchronized, expressing the happens-before arc fails, since the read possibly happens before the write. For the annotation of locking this means, that the annotation needs to take place, while the thread owns a lock, that prevents the other thread from entering the locked region.

OMPT only provides the events *acquiring* (i.e. asking for the lock), *acquired* (when the lock is acquired) and *released* (after the lock was released) of a lock. OMPT does not provide a *releasing* event to safe the potential overhead in the critical path of execution. As depicted in Figure 4a we would need to describe a happened before arc from a *releasing* event to the next *acquired* event. And an arc from a *released* event to the *acquired* event goes potentially backwards in time.

As work-around for this issue we set an own mutex in each *acquired* event, before we end the happens-before arc and release the mutex in the matching *released* event after we started the happens-before arc. This approach is depicted in Figure 4b. This way we can guarantee that we annotate the end of a happened before arc only after we annotated the begin of the happened before arc. Since the OpenMP runtime already acquired a lock, we don't expect lock contention. It just might be the case, that the previous locked region still holds the mutex to finish the *released* event.

4.2 Annotation of Task Dependencies

As discussed in 3.2, the synchronization behavior is different for *in* and *out* dependencies. The end of a task with an *in* dependency happened before a task begins with the same *out* dependency. The end of a task with an *out* dependency happened before a task begins with the same *in* or *out* dependency. That means, at the task begin with an *in* dependency, we need to differ the arcs that come from *in* or *out* dependencies.

So, we need two different tokens for starting the happens-before arc of *in* dependencies and *out* dependencies. This token need to be common knowledge of all task using the dependency and for TSan the requirement for a token is that it needs to be a valid memory address of the process. For this reason, it is natural to use the address of the dependency as the token to annotate the happened before arc. Since we need two tokens, we use the address provided as dependency and the address next to this address, assuming that applications will not use byte-sized variables as dependencies.

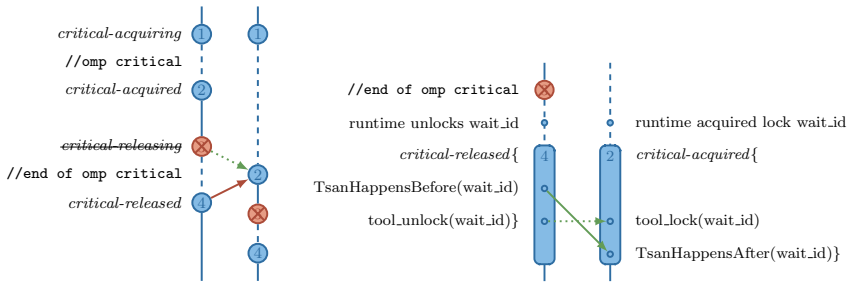

Fig. 4: a) OMPT doesn't provide a *releasing* event. Using the *released* event to start the happened-before arc potentially results in a happened-before arc backwards in time. b) We use an additional lock in the tool, to extend the exclusive region into the *released* callback. This way we can express the proper happened-before semantics.

4.3 Memory Behavior for Tasks

Another source of confusion for data race detection is the handling of OpenMP *data environment* for explicit tasks. For **`firstprivate`** variables, the OpenMP implementation needs to capture the value at task creation and provide the value at task execution. From what we have seen for various OpenMP implementations is, that the compiler generates code to copy the value at task creation into a task-specific data structure before adding the task to the queue. For the begin of execution of the task, the compiler generates code to copy the values to the stack of the task. These accesses are synchronized well. The problem arises from another implementation detail of the OpenMP runtime: The runtime takes the memory for the task structure from an internal memory pool. The access to the memory pool is internally locked. The tool can see memcpy to the task specific memory block at task creation and memcpy from the specific memory block at task execution. The tool can understand the synchronization between task creation and task execution. But there is no synchronization between end of execution—the free of the task specific memory block—and creation of another task—where the memory from the pool is reused. The tool needs to understand the new and free semantics for this memory provided by the runtime, but visible in the compile unit of the application.

Introducing a happened-before arc for this synchronization would be against the semantics of the OpenMP synchronization. Our proposal to solve this issue is a new OMPT inquiry function, where a tool can query for task specific memory blocks. The tool can then simulate the new and free semantics for this memory range at begin and end of task execution.

4.4 Annotation of Reductions

The current specification of OMPT provides no events for a reduction. The OpenMP specification does not require a specific point in the application execution, where the reduction needs to take place. Also an OpenMP implementation has a lot of freedom to implement the reduction algorithm, which results in various scenarios of memory

access patterns. Threads might accumulate the own value to another thread's reduction value, threads might fetch other thread's reduction value and accumulate at the own reduction value, a master thread might collect all reduction values. The reduction might also be implemented solely with atomic operations.

We propose the following events for the implementation of reductions:

- *release-reduction*: thread will not touch reduction variable after this event
- *reduction-begin*: begin of reduction operations
- *reduction-end*: end of reduction operations

We think, that *release-reduction* and *reduction-end* can share the same callback function. The callback function needs to provide information about the local copy of the reduction variable.

The LLVM/OpenMP runtime implements most reductions inside the synchronization of the barrier. So as a temporary workaround, we ignore memory accesses inside of OpenMP barriers. If a task is scheduled in the barrier, we turn of ignoring memory accesses and turn it back on, when the barrier gets active again. This works in most cases for this specific runtime, but we don't expect this to be a portable workaround.

4.5 Information on Flush-set

The current specification of the flush event as of TR4 only provides information on the source code of the flush (*codeptr_ra*) and the current thread, but no information on the provided *list* argument, which describes the flush-set of the flush operation. To derive the right flush semantics for data race detection, this information would be necessary.

We propose to extend the definition of **`ompt_callback_flush_t`** by an array of pointers and a size argument:

```
1 typedef void (*ompt_callback_flush_t) (
2   ompt_data_t * thread_data,
3   const void * list,
4   int list_length,
5   const void * codeptr_ra
6 );
```

5 Conclusions

In this paper we discussed whether OMPT provides sufficient information to derive all synchronization semantics needed for data race detection. We based the analysis on a happened-before based model. But we think, the observations would also apply for a different analysis model, based on lock-set or plain analysis of OpenMP flush semantics. We implemented a data race detection tool based on OMPT. With OMPT based annotations, the tool passes most of the tests in our test suite. We pointed out three missing pieces of information in the OMPT interface, that is information about reduction, information about runtime managed memory for tasks, and information on flush-set for flushes. We provide guidance on how to apply on-the-fly analysis for OpenMP mutual exclusion with the missing *releasing* event.

Further, we discussed the necessary OMPT events, to derive the synchronization information for data race analysis. To enable data race analysis based on these events, an OpenMP implementation needs to implement and provide callback invocation for these events. The issue here is that some of these callback invocations are optional according to current specification. This affects especially the events for taskwait, taskgroup, barrier and locks. If a data race detection tool cannot rely on these events, the advantage of portability across OpenMP implementations is gone. Therefore we suggest to make these callback invocations mandatory in the OpenMP specification.

Acknowledgments

Part of this work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344. (LLNL-PROC-730143). Part of this work was possible under funding by the German Research Foundation (DFG) through the German Priority Programme 1648 Software for Exascale Computing (SPPEXA).

References

1. Simone Atzeni, Ganesh Gopalakrishnan, Zvonimir Rakamaric, Dong H. Ahn, Ignacio Laguna, Martin Schulz, Gregory L. Lee, Joachim Protze, and Matthias S. Müller. ARCHER: effectively spotting data races in large openmp applications. In *2016 IEEE International Parallel and Distributed Processing Symposium, IPDPS 2016, Chicago, IL, USA, May 23-27, 2016*, pages 53–62, 2016.
2. OpenMP Architecture Review Board. OpenMP Application Program Interface. `http://www.openmp.org/wp-content/uploads/openmp-4.5.pdf`.
3. OpenMP Architecture Review Board. TR4: OpenMP Version 5.0 Preview 1. `http://www.openmp.org/wp-content/uploads/openmp-tr4.pdf`.
4. Christopher Lidbury and Alastair F. Donaldson. Dynamic race detection for C++11. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017*, pages 443–457, 2017.
5. Joachim Protze, Simone Atzeni, Dong H. Ahn, Martin Schulz, Ganesh Gopalakrishnan, Matthias S. Müller, Ignacio Laguna, Zvonimir Rakamaric, and Gregory L. Lee. Towards providing low-overhead data race detection for large openmp applications. In *Proceedings of the 2014 LLVM Compiler Infrastructure in HPC, LLVM 2014, New Orleans, LA, USA, November 17, 2014*, pages 40–47, 2014.
6. Konstantin Serebryany and Timur 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.
7. Konstantin Serebryany, Alexander Potapenko, Timur Iskhodzhanov, and Dmitriy Vyukov. Dynamic race detection with llvm compiler. In *Runtime Verification*, pages 110–114. Springer, 2012.