Low overhead data race detection in large structured parallel applications

Simone Atzeni
simone@cs.utah.edu

1 Introduction

Multithreaded programming has become widespread in use, given the need to employ multicore CPUs to gain higher performance at a given energy budget. In the High Performance Computing (HPC) world, this has lead to an increased adoption of on-node parallelism in large software applications. This trend is confirmed by work that is in progress at national research facilities [22, 23, 25].

Multithreaded programming is achieved through different programming models (e.g. Pthreads); however, the predominant paradigm of choice in HPC is OpenMP [18], which guarantees portability and ease of use. In fact, we are collaborating with computational scientists at the Lawrence Livermore National Laboratory (LLNL), one of the world's largest computing facilities, where one of the main ongoing tasks is the porting of critical multiphysics application [13] to OpenMP. In this community, OpenMP is of paramount importance to enable shared memory parallel programming; yet, porting large HPC applications to OpenMP is error-prone. The correctness of these applications is crucial to the reliability of critical simulations pertaining to many real world phenomena of fundamental importance such as modeling of nuclear explosions, weather simulations, hydrodynamics modeling, etc. One of the most common of error types in OpenMP applications is data races [24]. Data races are often hard to locate with traditional debugging methods. Precise checking tools to detect data races are needed now more than ever. While data race is a well-known problem and many Pthreads data race detection tools have been proposed over the past twenty years, none or just a few of them are actually able to analyze OpenMP programs. Our work targets this critical need.

Data race detection research has focused both on static and dynamic analysis techniques. Static analysis techniques allow to reason about all the inputs of the program and the interleavings of the threads, and they are fairly scalable and fast since no runtime overhead is incurred. However, the lack of information that exists only at runtime makes these techniques very imprecise; in fact they often miss races and generate many false positives. Runtime techniques are precise, as they do not report any false positives, and only report races in the branches of programs that are actually executed. On the other hand dynamic analysis for data race detection is known to generate a very high runtime and memory overhead due to the operations it needs to perform and the states it needs to maintain during the execution.

The runtime overhead of even the best of dynamic tools, such as the ThreadSanitizer (Tsan) and Intel Inspector XE, can cause between $5x-20x$ slowdown and the memory overhead can be between $2x-10x$ of the memory used by the normal execution of the programs. For very large programs, such as HPC applications, the runtime and memory overheads can be even bigger. For instance, our experiments show that this slowdown can be $100x$ and the memory consumption can be $50x$ larger. The high runtime slowdown and memory usage make such tools useless from the point of view of actual developers, who probably would not be keen on waiting a long time to check their programs; or they may not even have enough machine resources to run the tools. We definitely need better techniques – static, dynamic, or combinations – to detect data races in large structured parallel applications, while guaranteeing precision and accuracy while incurring modest overheads.

1.1 Thesis statement

The goal of this dissertation is to provide a sound and precise data race detection tool for large OpenMP applications that combine static and dynamic analysis while maintaining a low runtime and memory overhead. *Our thesis statement is that by combining the best of these techniques and tailoring the implementation to the actual concurrency structure of structured parallel languages such as OpenMP, we can make data race checking of HPC applications practical.*

Our approach is now briefly summarized. First, we combine static and existing dynamic techniques to reduce the amount of code to analyze at runtime, thus lowering the overheads. Second, we will develop a new runtime technique for data race detection that exploits the structured parallelism of OpenMP to reduce runtime and memory overhead, while guaranteeing high precision and accuracy. Our project goals are now listed, as conceived at the beginning of our research.

- **Sequential Blacklisting:** We will design a static analysis to identify the code executed sequentially in a OpenMP program.
- **Data Dependency Analysis:** We will design a static analysis technique that identify race free code in a OpenMP program to exclude it from the runtime analysis.
- **Archer v1:** We will combine the static analysis with the existing Clang/LLVM Tsan data race detection tool and we will make Tsan capable of detect data races in OpenMP programs, as part of a first version of a data race detector called Archer.
- **Clock-less runtime algorithm:** We will design and implement a new runtime analysis technique that exploit the structured parallelism of the OpenMP paradigm in order to reduce the runtime and memory overheads.
- **Archer v2:** We will embed the aforementioned new runtime technique in a second version of Archer as a replacement of the TSan runtime.

The first three goals have been achieved in our previous workshop [20] and conference [10] papers.

2 Background

Data race detection is probably one of the most widely studied of problems in concurrent programs design and debugging, and has been shown to be NP-hard [16]. Data races are in general a symptom for a large number of root-causes: lack of atomicity [8], unintended sharing [7], or a misunderstanding of how generated code behaves (e.g. miscompilation [3]). Many techniques have been proposed to detect data races, either static or dynamic.

Static race detection methods are known to provide wide coverage of the program, since they can reason about all the inputs and the thread interleavings, however are also known to generate many false positives [19] and miss races [27]. Dynamic race detection techniques analyze a specific trace of a program that is actually executed. Dynamic race detectors process the programs events in parallel during the execution [11, 21, 9, 26]. A large number of dynamic race detection tools are based on one of the following algorithms: happens-before, lockset or both (hybrid techniques). These algorithms are described in detail in [17]. Most of dynamic data race detectors use these algorithms to detect races in Pthreads programs. In fact, existing tools, such as TSan [26], implement a very optimized happens-before based techniques that guarantee high precision and accuracy, a runtime overhead of $5x - 20x$, and a memory overhead of $2x - 10x$. However, none of them is actually able to identify correctly data races in OpenMP programs.

Even though, an OpenMP program is typically translated in a Pthreads program by the compiler, these tools often miss races or report false positives because cannot recognize synchronization mechanism used by the OpenMP programming paradigm such as barriers, critical sections, etc. Only existing commercial tools, such as Intel®Inspector XE or Sun Studio Data-Race Detection Tool, provide data race detection

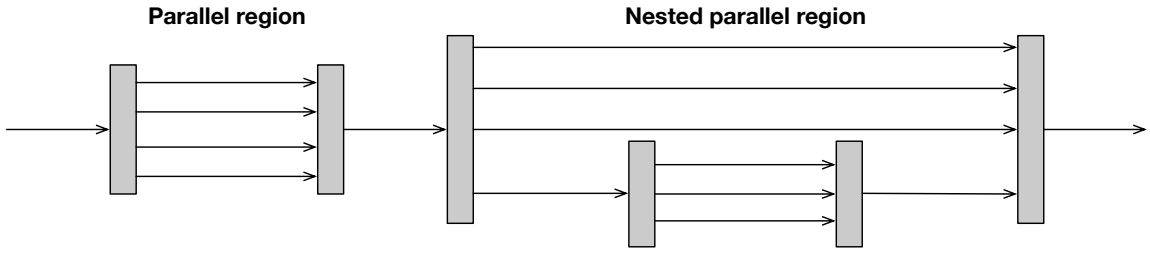

Figure 1: OpenMP nested parallelism

for OpenMP programs through a binary instrumentation of the executable, and generally applying an implementation of the happens-before relation. Experiments show that such tools [20] are not very precise and accurate (indeed they often miss races and report many false alarms). Furthermore they generate a very high runtime and memory overhead that make their usage often infeasible especially with large scientific HPC applications.

3 Overall Research Plan

To the best of our knowledge there are not, out there, OpenMP data race detectors that guarantee low runtime and memory overhead while maintaining high precision and accuracy. This motivates the need for better data race detection techniques and tools for structured parallel programming paradigms. As claimed in § 1, we propose several different contributions to obtain a better data race detection tool for OpenMP programs. In the remainder of this section I will discuss and detail our proposed contributions and how the combination of them helps to obtain a better OpenMP data race detector.

Sequential Blacklisting: We exploit OpenMP's structured parallelism to identify guaranteed sequential regions within OpenMP code. Such analysis would be difficult to conduct in the context of unstructured parallelism (e.g., PThreads). Figure 1 shows an example of OpenMP parallel and nested parallel regions. As we can see, the parallel regions are alternated by the solely main thread execution. The code executed by the main thread is sequential and cannot race with any other threads existing in a parallel region. Therefore, the instructions executed by the main thread during its sequential execution can be excluded from the runtime analysis reducing the overhead.

Data Dependency Analysis: We identify and suppress race free parallel loops from race checking. Listing 1 shows an example of two parallel for-loops. The first one is easily and automatically parallelizable, the array will be divided in different chunks and each of them will be assigned to different threads. However, the second parallel for-loops has a data dependency in the array accesses, which will introduce a data race when the array's chunks are assigned to different threads. Indeed, it may happen that two threads will access the same locations simultaneously without any synchronization mechanism. The OpenMP compiler do not apply any check and will parallelize both loop in the same way. We introduce a data dependency analysis at compiling time that identify loops with and without dependencies. The firsts will be blacklisted and excluded by the runtime analysis since race free, the seconds instead will be checked at runtime.

Archer v1: The two previous static analysis techniques and an instrumented version of the OpenMP runtime make the first version of the OpenMP data race detector Archer. The static analyses will be implemented as a LLVM/Clang passes and integrated in the compilation process. We will annotate the OpenMP runtime to communicate the TSan about the happens-before relations between threads in presence of unknown synchronization mechanisms. For example, OpenMP barriers and OpenMP critical section are unknown to the TSan runtime, which would make it reports false positives. This first version of Archer is the initial step towards low overhead data race detector for large OpenMP applications, and it would also allow us to evaluate the benefits of the static analyses in terms of both overheads reduction, and precision

and accuracy of the data race detection process.

Listing 1: OpenMP loops with and without loop-carried data dependency.

```
#pragma omp parallel for
for(int i = 0; i < N; i++) {
  a[i] = a[i] + 1;
}

#pragma omp parallel for
for(int i = 0; i < N; i++) {
  a[i] = a[i + 1];
}
```

Clock-less runtime algorithm: We propose a new data race detection algorithm that, differently from past and modern Pthreads data race detection techniques usually based on vector and lamport clocks, exploits the structured parallelism of parallel programming models, such as OpenMP, to guarantee high an equivalent or better data race detection precision and accuracy using less runtime and memory resources. In OpenMP the concurrency structure generated by its fork-join model is much simpler than other programming model such as Pthreads.

3.1 A Formal Analysis of the OpenMP Concurrency Structure

To highlight the well-structured concurrency model of an OpenMP program consider the example show in figure 2. The example shows a main thread [1] that spawns a parallel region of two threads. After that, each thread will be creates its own nested parallel regions, which after their execution-life will join to a single thread, and so on for each region, until they join again to the main thread. Differently from the Pthreads programming models, in OpenMP the main thread (or in general the forking thread) will be always part of the parallel region generated, and it will do the same work as the others newly created threads. This emphasize the concurrency model of the threads within a parallel region and threads across parallel regions. In fact, in OpenMP two threads can race within a same parallel regions (e.g. thread 3 and 4) or across parallel regions (e.g. thread 3 and 5), but two threads can not race if they belong to two consequent parallel regions (e.g. thread 3 and 9). Threads that belongs to two consequent parallel regions are not concurrent since there is always a join point that separate them, furthermore in OpenMP, at the end of each parallel region there is an implicit barrier which synchronizes all threads before starting the next parallel task. We base our idea of data race detection to these facts, and exploit an existing thread labeling approach for fork-join [15] models to quickly identify if two threads are concurrent. In this way, we can limit the data race detection only to the concurrent threads [2].

We formally define the concurrency structure of OpenMP from the point of view of race checking. To the best of our knowledge the OpenMP concurrency model has never been formally documented. We define a state machine and a set of transition rules that model the behavior of an OpenMP program. The appendix A shows an excerpt of the formal definition of the state machine. A full definition of the operational semantics is defined in [2]. We abstract away from data state as much as possible, using uninterpreted functions to occasionally bridge the gap. The transition system can fire, for any thread, any rule at any point changing the state of the system. Moreover, it can also fire the "RaceCheck" rule at any point of the state machine execution, flagging the existence of a race.

The transition system is an ideal state machine that guarantees to detect every race present in an OpenMP program, without reporting any false positive. From the formal point of view, the transition system is a machine with infinite memory and in fact it keeps track of every memory access performed by any thread in the system. This approach would not be feasible in the real-world since the runtime and memory resources are limited, but most importantly because our main goal is to obtain a low runtime and memory overhead data race detection algorithm. Even though the state machine is an ideal system, it allows to model all the corner cases in the OpenMP race-checking problem and design a real better algorithm that guarantee correctness and soundness.

We believe that the transition system can help to design a precise and fast data race detection algorithm. We intend to implement an algorithm that at runtime models the concurrency structure of the parallel

[1] A thread is represented as a circle.

[2] A common happens-before algorithm based on vector clocks would perform a race check also on non-concurrent threads, because of its lack of knowledge on the concurrency structure of the OpenMP program.

regions of an OpenMP program. The concurrent model of the program will be stored in a lock-free and fast data structure [5, 14] that can be concurrently read and written by multiple threads. Furthermore, we will rely in another lock-free and fast data structure to maintain a smaller representative set of memory accesses performed by the threads. We plan to perform the data race detection check at every barrier checkpoint. This will limit the number of race checks performed during the program execution, differently from vector-clock techniques that generally perform the check at every memory accesses. A similar idea is used by GPU race checkers, such as GKLEE [12], where the race check is performed only within a barrier interval. The real race-check is illustrated by the "RaceCheck" rule in appendix A.5 on the smaller set of memory accesses. The OpenMP Tools Application Programming Interfaces (OMPT) [1] implemented by the OpenMP runtime provides all the information to build the concurrency structured of an OpenMP program at runtime. We will indeed based our implementation on the OMPT API guaranteeing a standard and portable solution of the data race detection algorithm.

ARCHER v2: The second version of ARCHER will embed both static analyses and the runtime analysis technique for data race detection of structured parallelism. We plan to integrate ARCHER in the LLVM/Clang compiler infrastructure and guarantee the same ease to use that characterize most LLVM tools. In fact, we will provide a new compilation flag (i.e. "-archer") that will instrument the OpenMP program at compile time and will link it against the ARCHER runtime library. The program can then be executed as usual, while the ARCHER runtime library will perform the data race detection providing at the end of the execution a detailed report about the detected races.

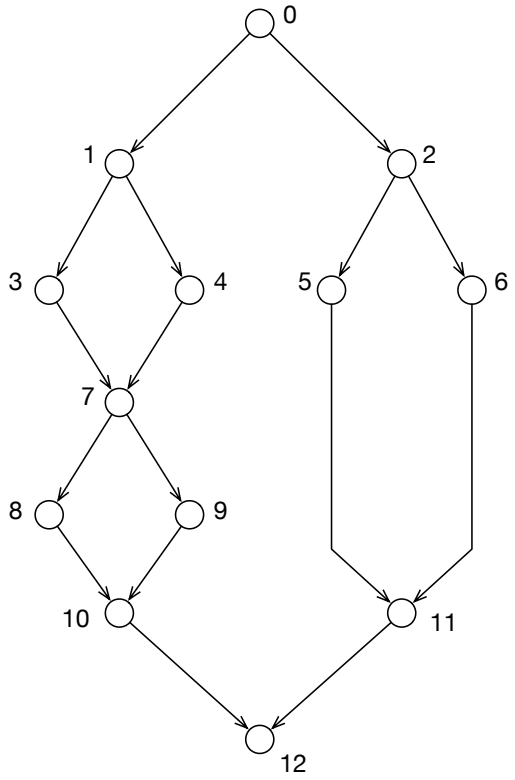

Figure 2: Structure of an OpenMP program with nested parallelism.

4 Accomplished Work

The first part of the work has been accomplished and published in [20, 10].

In [20], where I am a co-author, we present a feasibility study of applying the static analysis and the runtime annotations approach introduced in § 3. We demonstrate that the static analysis through sequential blacklisting and data dependency analysis can respectively help to exclude sequential and race free OpenMP regions of code from the runtime analysis in order to reduce the overhead. While the runtime annotation approach can be used to make TSan aware of OpenMP synchronization primitives, so that it can report correctly data races in an OpenMP program, without reporting any of all the false positives previously and erroneously detected in the OpenMP runtime.

My main contribution is presented in the paper [10], which has been accepted at IPDPS'16 and I will present in May 2016. In this work we introduce ARCHER a data race detection for OpenMP applications. ARCHER embeds seamlessly the static analyses for sequential blacklisting and data dependency on TSan compiler instrumentation mechanism as LLVM passes and provides an annotate version of the OpenMP runtime to enable TSan on OpenMP programs. The static analyses in synergy with the instrumentation mechanism identify the sequential and the race free regions of

code instrumenting only the rest of the code potentially racy. As a result, the compiler produces an instrumented executable linked against the annotated OpenMP runtime provided by ARCHER. The annotations communicate with the TSan runtime in order to establish an implicit happens-before relations, between the involved threads, in those cases where the OpenMP runtime adopts synchronization primitive unknown by TSan (e.g. omp barrier, etc.). This solution builds the knowledge of the happens-before relation into the runtime libraries to obtain precision and accuracy in the data race detection process. As shown in [10], this first part of the work shows good results in terms of reduction of runtime overhead. In fact, ARCHER, on the OmpSCR [6] and AMG [4] benchmarks, achieve a speedup of about 25% while maintaining the same or better precision and accuracy respect existing data race detectors such as Intel®Inspector XE and TSan.

I have conducted and contributed to this work during our current collaboration between our research group and a group of computer scientists at the LLNL, where I have been a student intern in the past two summers (Summer'14 and Summer'15). I will spend Summer'16 at IBM T.J. Watson Research Center as a student intern, where I will be involved in the development of the OpenMP Tools Application Programming Interfaces for Performance Analysis and Debugging [1] (OMPT and OMPD API). This experience will give me the opportunity to further my investigation on the OpenMP runtime and improve my knowledge of the OMPT and OMPD API that are central in the implementation of our proposed race detection technique.

5 Timeline

1. **November 2014:** Presented paper at LLVM-HPC Workshop 2014 at SC 2014.
2. **May 2016:** Present paper on ARCHER v1 at IPDPS 2016.
3. **July 2016:** Submit paper to POPL 2017, or PPoPP 2017. We aim for a first version of the proposed runtime algorithm.
4. **Septmber – February 2017:** Release final version of ARCHER (combination of static analysis and new runtime) with extensive evaluation of the tool on large HPC benchmarks. Sumbit it to PLDI, IPDPS, CGO, or PACT.
5. **May 2017:** Submit PhD dissertation.

References

[1] A. Eichenberger, J. Mellor-Crummey, M. Schulz. OMPT and OMPD: OpenMP Tools Application Programming Interfaces for Performance Analysis and Debugging. `http://openmp.org/mp-documents/ompt-tr.pdf`.

[2] S. Atzeni and G. Gopalakrishnan. Openmp concurrency operational semantics. `https://goo.gl/B9cV6z`.

[3] H.-J. Boehm. How to miscompile programs with "benign" data races. In *USENIX Conference on Hot Topic in Parallelism*, pages 3–3, 2011.

[4] Center for Applied Scientific Computing (CASC) at LLNL. AMG2013. `https://codesign.llnl.gov/amg2013.php`.

[5] M. Desnoyers, P. E. McKenney, A. S. Stern, M. R. Dagenais, and J. Walpole. User-level implementations of read-copy update. pages 375–382, 2012.

[6] A. J. Dorta, C. Rodríguez, F. de Sande, and A. González-Escribano. The OpenMP source code repository. In *PDP*, pages 244–250, 2005.

[7] J. Erickson, S. Freund, and M. Musuvathi. Dynamic analyses for data-race detection. `http://rv2012.ku.edu.tr/invited-tutorials/`, 2012.

[8] J. Erickson, M. Musuvathi, S. Burckhardt, and K. Olynyk. Effective data-race detection for the kernel. In *OSDI*, 2010.

[9] C. Flanagan and S. N. Freund. FastTrack: Efficient and precise dynamic race detection. In *PLDI*, pages 121–133, 2009.

[10] IPDPS'16. *ARCHER: Effectively Spotting Data Races in Large OpenMP Applications*, May 2016.

[11] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, pages 558–565, 1978.

[12] P. Li, G. Li, and G. Gopalakrishnan. Practical symbolic race checking of GPU programs. In *Supercomputing*, pages 179–190, 2014.

[13] Compute codes. `https://wci.llnl.gov/simulation/computer-codes`.

[14] A. Matveev, N. Shavit, P. Felber, and P. Marlier. Read-log-update: A lightweight synchronization mechanism for concurrent programming. In *Proceedings of the 25th Symposium on Operating Systems Principles*, pages 168–183, 2015.

[15] 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*, pages 24–33, 1991.

[16] R. H. B. Netzer and B. P. Miller. What are race conditions?: Some issues and formalizations. *LOPLAS*, pages 74–88, 1992.

[17] R. O'Callahan and J.-D. Choi. Hybrid dynamic data race detection. *SIGPLAN Not.*, pages 167–178, 2003.

[18] OpenMP Architecture Review Board. OpenMP application program interface version 4.0. `http://www.openmp.org/mp-documents/OpenMP4.0.0.pdf`.

[19] P. Pratikakis, J. S. Foster, and M. Hicks. LOCKSMITH: Practical static race detection for C. *TOPLAS*, 33(1):3:1–3:55, 2011.

[20] J. Protze, S. Atzeni, D. H. Ahn, M. Schulz, G. Gopalakrishnan, M. S. Müller, I. Laguna, Z. Rakamarić, and G. L. Lee. Towards providing low-overhead data race detection for large openmp applications. In *Proceedings of the 2014 LLVM Compiler Infrastructure in HPC*, LLVM-HPC '14, pages 40–47, 2014.

[21] 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.*, pages 27–37, 1997.

[22] CORAL/Sierra. `https://asc.llnl.gov/coral-info`.

[23] SUMMIT: Scale new heights. Discover new solutions. `https://www.olcf.ornl.gov/wp-content/uploads/2014/11/Summit_FactSheet.pdf`.

[24] M. Süßand C. Leopold. Common mistakes in OpenMP and how to avoid them: A collection of best practices. In *Proceedings of the 2005 and 2006 International Conference on OpenMP Shared Memory Parallel Programming*, pages 312–323, 2008.

[25] Trinity. `http://www.lanl.gov/projects/trinity/`.

[26] ThreadSanitizer, a data race detector for C/C++ and Go. `https://code.google.com/p/thread-sanitizer/`.

[27] J. W. Voung, R. Jhala, and S. Lerner. RELAY: Static race detection on millions of lines of code. In *ESEC-FSE*, pages 205–214, 2007.

Appendix A OpenMP Concurrency Operational Semantics

This is an excerpt of the operational semantics, the full version can be found in [2].

A.1 Convention

- $\mathbb{N}$ is the set of natural numbers, $\{0, 1, 2, \ldots\}$.
- $t \in TID$ is a thread ID (a natural number) for some $TID \in \mathbb{N}$ (TID thread IDs are allowed).
- Let $ADDR \in \mathbb{N}$ be the range of memory addresses accessed by the various threads.

A.2 Offset-Span Labels

We follow the concept of offset-span labels introduced in the paper of Mellor-Crummey [15]. An offset-span label (*osl* for short) labels each thread's execution point with a sequence of pairs, marking its lineage in the concurrency structure defined by prior forks and joins. The domain of $OSL = ((\mathbb{N} \times \mathbb{N}))^{\mathbb{N}}$, i.e. each member $osl \in OSL$ is a sequence of pairs $\langle a_1, b_1\rangle, \langle a_2, b_2\rangle, \ldots, \langle a_n, b_n\rangle$ suppose $osl_1, osl_2 \in OSL$. These labels are sequential exactly when one of the OSLs (say osl_1) is a prefix of the other. Otherwise, they are concurrent. For further details, please see [15].

A.3 System State

The state of the system consists of a global states GS and a set of thread local states TP (Thread Pool). The total state ts of any system is a pair "Global State, Thread Pool;" i.e., a specific total state ts is:

$$ts = \langle gs, tp\rangle$$

Each total state ts comes from the domain TS, where $TS = GS \times TP$.
Each global state gs is a 5-tuple:

$$\langle bm, m, n, rw, \sigma\rangle$$

Each global state gs comes from the domain GS, where

$$TS = BM \times M \times N \times RW \times \Sigma$$

where:

- The domain $BM = ParRegID \mapsto (\mathbb{N}\times\mathbb{N})$. Thus, for each $bm \in BM$, we have $bm : ParRegID \mapsto (\mathbb{N}\times\mathbb{N})$. Given a $p \in ParRegID$, bm returns a pair of natural numbers (a, b), where a is the "current Barrier Count" and b is the "target Barrier Count." When a thread t with offset span label osl executes a $ParBegin(N)$ instruction, N threads are created, and an entry $\langle osl, (0, N)\rangle$ is added to function bm[3]. The first field 0 will be incremented each time we close a barrier or a parallel region, in a manner to be described momentarily. Threads that have to meet a barrier or have to hit the $ParEnd$ constructs.
- Mutex m comes from domain M, where $M = \{-1\} \cup \mathbb{N}$. Each m is initialized to -1 when the mutex is free. When thread $t \in TID$ acquires the mutex, we record the value t in m, indicating that the mutex is taken, and also taken by which thread.
- "Nutex" n comes from domain N where $N = Names \mapsto (\{-1\} \cup TID)$. That is, given a named mutex name $n \in Names$, $N[n] = -1$ means that this named mutex ("nutex") is free. Otherwise, $N[n] = t$, recording the fact that this nutex is held by thread t.
- Let memory access-type $MAT = \{R, W\}$.
- $rw \in RW$ is a tuple that maintains all the memory accesses of each thread in the system. We have $RW = TID\times OSL\times\mathbb{N}\times ADDR\times MAT\times M\times N$. Each memory access performed by thread t is recorded as the tuple

  $$\langle tid, osl, bl, addr, mat, mutex, nutex\rangle$$

  where:

[3]Recall that functions are single-valued relations, or sets of pairs with unique second component for each given first component. Thus, $\{\langle osl, (0, N)\rangle\}$ is a function. We will allow functions to evolve, i.e. undefined for items explicitly added.

  - $tid \in TID$ is the thread ID;
  - $osl \in OSL$ is the offset-span label;
  - $bl \in \mathbb{N}$ is the barrier label of the last barrier seen by the thread t;
  - $addr \in ADDR$ is the memory address;
  - $type \in \{R, W\}$ records reads or writes;
  - *mutex* is the mutex state (value of M in GS) at the time of the access;
  - *nutex* is the nutex state (value of N in GS) at the time of the access.
- $\sigma \in \Sigma$ is the data state of the system, as described earlier.

The local state TP is the thread pool that contains a list of 3-tuples, each of which is the local state of a thread:

$$\langle tid, osl, bl \rangle$$

The domain $TP = 2^{TID \times OSL \times \mathbb{N}}$ where:
- $t \in TID$ is thread ID of the thread;
- $osl \in OSL$ is an offset-span label;
- $bl \in \mathbb{N}$ is the label of the barrier the thread has witnessed last. We assume that each barrier instruction is of the form $bar(L)$ where $L \in \mathbb{N}$ carries the barrier number. A thread crossing a barrier sets its bl to the value L.

A.4 Helper Functions and Predicates

- *as*: is used as in Ocaml (it allows a name for a whole structure, as well as helps us refer to the inner details of the structure).
- *most*(*lst*): we define *most* as a function that return the same list given in input except the last element (i.e. in Python lst[:-1]).
- ||: This operator is used to describe that two different threads are concurrent. In particular, given two offset-span labels $l1$ and $l2$, $l1 \parallel l2$ (read $l1$ and $l2$ are concurrent) means that the two threads associated to the two offset-span labels are concurrent. In case the labels represent barrier labels, it means that the two barrier are concurrent, in other words they happen in two different (nested) parallel regions.
- *SpawnChildren*($\langle ptid, posl, pbl \rangle, \sigma, N$): Given the parent's TID (*ptid*), offset-span label (*posl*) and barrier label (*pbl*), this function creates a pool of N threads – specifically, the local states of these threads $\langle tid, osl, bl \rangle$. It initializes the offset-span label *osl* for each thread created using the rules in § A.2, by extending *posl* with pairs $[0, N]$ through $[N-1, N]$. The *bl* is set to *pbl*. The TIDs are somehow uniquely generated (*osl* could be used as an index into an evolving TID bijection).
- *GetChildJoin*(*tp*): returns the single thread-state triple that results from fusing all the threads in the thread pool *tp*. The offset-span labels are all chopped, and the single data state that is carried forward has the right PC and data state, going forward.
- *Concurrent*($OSL, t1, t2$) is as described in §A.2.
- Function *AddRW*($\langle tid, osl, bl, addr, mat, m, n \rangle$) adds the access into the RW Structure. The record says "an access by *tid* with offset-span label *osl* and barrier label *bl* is performed at address *addr* with memory access type *mat*, when the mutex state is *m* and the nutex state is *n*."
- *Full*(*bm*, *osl*): This predicate keeps the counts the number of threads that have reached a *ParEnd*(N) (or a *Barrier*(*bid*)) construct. In order to count the threads, it uses the structure *bm* which is indexed by the *ParRegID* represented by the offset-span label *osl*. In other word, the predicate *Full* means that other threads have reached the construct and have incremented the counter in the *bm* structure. From a functional language point of view *Full* would look like:

```
let Full(bm, osl) =
   let (count, N) = bm[osl]
   in (count == N - 1)
```

A.5 Structured Operational Semantics Rules

$$\text{ParBegin(N)}\ \frac{\begin{array}{c} gs \text{ as } (bm, m, n, rw, \sigma) \in GS \\ te \text{ as } (tid, osl, bl) \in TP \\ at(tid, \sigma, ParBegin(N)) \\ tp' = (tp - \{te\} \cup SpawnChildren(\langle tid, osl, bl\rangle, \sigma, N)) \\ bm' = bm \cup \{\langle osl, (0, N)\rangle\} \\ \sigma' = nxt(\sigma, tid) \end{array}}{\langle gs, tp\rangle \longrightarrow \langle gs' \text{ as } \langle bm', m, n, rw, \sigma'\rangle, tp'\rangle}$$

$$\text{ParEnd(N)}\ \frac{\begin{array}{c} gs \text{ as } (bm, m, n, rw, \sigma) \in GS \\ te \text{ as } (tid, osl, bl) \in TP \\ tp' \subseteq tp \\ at(tid, \sigma, ParEnd(N)) \\ Full(bm, most(osl)) \\ bm' = bm - \{\langle most(osl), *\rangle\} \\ \sigma' = nxt(\sigma, tid) \\ tp'' = tp - tp' \cup GetChildJoin(tp') \end{array}}{\langle gs, tp\rangle \longrightarrow \langle gs' \text{ as } \langle bm', m, n, rw, \sigma'\rangle, tp''\rangle}$$

$$\text{ParEnd(N)}\ \frac{\begin{array}{c} gs \text{ as } (bm, m, n, rw, \sigma) \in GS \\ te \text{ as } (tid, osl, bl) \in TP \\ tp' \subseteq tp \\ at(tid, \sigma, ParEnd(N)) \\ \neg Full(bm, most(osl)) \\ bm[most(osl)] \text{ as } (count, N) \\ bm' = bm \cup \{\langle osl, (count + 1, N)\rangle\} \\ \sigma' = nxt(\sigma, tid) \end{array}}{\langle gs, tp\rangle \longrightarrow \langle gs' \text{ as } \langle bm', m, n, rw, \sigma'\rangle, tp\rangle}$$

$$\text{LoadStore}\ \frac{\begin{array}{c} gs \text{ as } (bm, m, n, rw, \sigma) \in GS \\ te \text{ as } (tid, osl, bl) \in TP \\ at(tid, \sigma, LoadStore(addr, mat)) \\ rw' = AddRW(tid, osl, bl, addr, mat, mutex, nutex) \\ \sigma' = nxt(\sigma, tid) \end{array}}{\langle gs, tp\rangle \longrightarrow \langle gs' \text{ as } \langle bm, m, n, rw', \sigma'\rangle, tp\rangle}$$

$$\text{AcquireMutex}\ \frac{\begin{array}{c} gs \text{ as } (bm, m, n, rw, \sigma) \in GS \\ te \text{ as } (tid, osl, bl) \in TP \\ at(tid, \sigma, AcquireMutex()) \\ m = -1 \\ m' = tid \\ \sigma' = nxt(\sigma, tid) \end{array}}{\langle gs, tp\rangle \longrightarrow \langle gs' \text{ as } \langle bm, m', n, rw, \sigma'\rangle, tp\rangle}$$

$$\text{ReleaseMutex}\ \frac{\begin{array}{c} gs \text{ as } (bm, m, n, rw, \sigma) \in GS \\ te \text{ as } (tid, osl, bl) \in TP \\ at(tid, \sigma, ReleaseMutex()) \\ m = t \\ m' = -1 \\ \sigma' = nxt(\sigma, tid) \end{array}}{\langle gs, tp\rangle \longrightarrow \langle gs' \text{ as } \langle bm, m', n, rw, \sigma'\rangle, tp\rangle}$$

$$\text{AcquireNutex(name)}\ \frac{\begin{array}{c} gs \text{ as } (bm, m, n, rw, \sigma) \in GS \\ te \text{ as } (tid, osl, bl) \in TP \\ at(tid, \sigma, AcquireNutex(name)) \\ n[name] = -1 \\ n' = n[name \rightarrow tid] \\ \sigma' = nxt(\sigma, tid) \end{array}}{\langle gs, tp\rangle \longrightarrow \langle gs' \text{ as } \langle bm, m, n', rw, \sigma'\rangle, tp\rangle}$$

$$\text{ReleaseNutex(name)}\ \frac{\begin{array}{c} gs \text{ as } (bm, m, n, rw, \sigma) \in GS \\ te \text{ as } (tid, osl, bl) \in TP \\ at(tid, \sigma, ReleaseNutex(name)) \\ n[name] = tid \\ n' = n[name \rightarrow -1] \\ \sigma' = nxt(\sigma, tid) \end{array}}{\langle gs, tp\rangle \longrightarrow \langle gs' \text{ as } \langle bm, m, n', rw, \sigma'\rangle, tp\rangle}$$

$$\text{Barrier(bid)}\ \frac{\begin{array}{c} gs \text{ as } (bm, m, n, rw, \sigma) \in GS \\ te \text{ as } (tid, osl, bl) \in TP \\ at(tid, \sigma, Barrier(bid)) \\ Full(bm, most(osl)) \\ bm' = bm - \{\langle osl, *\rangle\} \\ \sigma' = nxt(\sigma, tid) \end{array}}{\langle gs, tp\rangle \longrightarrow \langle gs' \text{ as } \langle bm', m, n, rw, \sigma'\rangle, tp\rangle}$$

$$\text{Barrier(bid)}\ \frac{\begin{array}{c} gs \text{ as } (bm, m, n, rw, \sigma) \in GS \\ te \text{ as } (tid, osl, bl) \in TP \\ at(tid, \sigma, Barrier(bid)) \\ \neg Full(bm, most(osl)) \\ bm[most(osl)] \text{ as } (count, N) \\ te' as (tid, osl, bid) \\ tp' = tp - te \cup \{te'\} \\ bm' = bm \cup \{\langle osl, (count + 1, N)\rangle\} \\ \sigma' = nxt(\sigma, tid) \end{array}}{\langle gs, tp\rangle \longrightarrow \langle gs' \text{ as } \langle bm', m, n, rw, \sigma'\rangle, tp'\rangle}$$

$$\text{RaceCheck}\ \frac{\begin{array}{c} gs \text{ as } (bm, m, n, rw, \sigma) \in GS \\ te_1 \text{ as } (tid_1, osl1, bl1) \in tp \\ te_2 \text{ as } (tid_1, osl1, bl1) \in tp \\ (tid_1 \neq tid_2) \\ Concurrent(osl, tid_1, tid_2) \\ i \in rw[tid_1] \\ j \in rw[tid_2] \\ (rw[tid_1][i].addr == rw[tid_2][j].addr) \\ (rw[tid_1][i].mat == W) \vee (rw[tid_2][j].mat == W) \\ (rw[tid_1][i].mutex == -1) \vee (rw[tid_2][j].mutex == -1) \\ (rw[tid_1][i].nutex \cap rw[tid_2][j].nutex = \emptyset) \\ (rw[tid_1][i].bl == rw[tid_2][j].bl) \vee (rw[tid_1][i].bl \parallel rw[tid_2][j].bl) \end{array}}{\langle gs, tp\rangle \rightarrow RaceFail(\sigma, addr, tid_1, tid_2)}$$