CS610 - Programming for Performance
2020-2021-I: Paper Reading 1

Name: Sarthak Singhal Roll No.: 170635

1. ARCHER

Paper Summary

OpenMP is a widely used programming model, but there are a very few efficient development tools for it. The pre-existing data race checkers have high overheads, generate many false positives, can't handle large code sizes and also don't provide effective debugging support for concurrency bugs. The paper introduces a new OpenMP data race detector ARCHER, that achieves high accuracy and also low overheads on large applications. Huge number of memory accesses and their happens-before ordering need to be tracked to check for races for which ARCHER employs scalable methods for happens-before tracking. It also exploits structured parallelism of OpenMP for blacklisting accesses which has never been done in any OpenMP race checker. ARCHER has also made a significant impact in some active projects, e.g. it helped LLNL in finding concurrency bug in HYDRA which is a large multiphysics application.

ARCHER is implemented using LLVM/Clang tool infrastructure and it employs modular static and dynamic analysis stages for its implementation. The static analysis phase blacklists the loads/stores which are guaranteed race free. Polly is used for data dependency analysis to generate a Parallel Blacklist and also a call graph is built and traversed to find regions which don't lie inside OpenMP constructs to create Sequential Blacklist. These blacklists are then fed to a customized version of TSan(ThreadSanitizer) which is a pre-existing dynamic race checker so that only potentially racy sections are checked. TSan can't be used directly as it doesn't know about synchronization points of OpenMP due to which it will generate a large number of false positives as due to synchronization points threads will enter critical section in serialized manner but TSan will be unaware of that and will report a data race. So, TSan's annotation API is used to mark synchronization points within OpenMP runtime to prevent false positives. Finally, this customized TSan instrumentation generates a selectively instrumented binary which will be used to get data race reports. ARCHER significantly outperforms both TSan and Intel®Inspector XE in terms of precision, accuracy, runtime and memory overheads when evaluated over OmpSCR benchmark suite and AMG2013 from HPC CORAL benchmark suite and even found some data races which were not known prior.

Key insights/Strengths

- The static analysis phase before the dynamic analysis phase is a very good idea to decrease the runtime overhead. The static analysis phase blacklists the memory accesses that can never be the cause of a data race due to which the work that has to be done during runtime is reduced. Due to static analysis the instrumentation of independent loop iterations and sequential code regions is avoided. The OmpSCR benchmarks are small, so there are only few blacklisting opportunities but then also on average ARCHER with static analysis is 15% faster compared to ARCHER without static analysis while in both cases catching all the data races. In AMG2013, which is real-world HPC application with a larger codebase(approx. 75,000 lines of code), ARCHER with static analysis is better in performance in terms of speed by a factor between 1.2 and 1.5 depending on number of threads as compared to ARCHER without static analysis.

- The idea of modifying TSan to reduce false positives is also very subtle. As unmodified TSan doesn't know about the synchronization points of OpenMP like barrier, critical, atomic, single, task, reduction, etc. it can report a data race even though threads will enter the critical section in serialized manner only. Thus, TSan's annotation API is used to mark

these synchronization points within OpenMP runtime to prevent false positives.

- The benchmarks and the applications used for the evaluation of ARCHER were picked nicely. They contained smaller benchmarks like OmpSCR benchmark suite with basic programs like fft and quicksort. They even evaluated performance over larger codebases like AMG2013 containing approximately 75,000 lines of code. ARCHER even helped LLNL to resolve crashes in an application named HYDRA by detecting benign data races.

Weaknesses

- ARCHER uses TSan in the dynamic analysis and TSan uses a thread-centric scheme for reasoning about orderings. Hence, ARCHER will incur false negatives when conflicting, logically concurrent accesses are both performed by same physical thread. Different iterations may be scheduled to same physical thread by the OpenMP runtime and the thread-centric data race detection algorithms will not report a data race if there was any as those iterations are executed by the same thread.

- Although ARCHER has less memory overhead as compared to Intel®Inspector XE, but then also it appears to be unnecessarily large. This happens due to TSan's runtime shadow memory allocation policy. As ARCHER uses TSan with some modifications but which are disjoint with the shadow memory allocation policy, ARCHER has a large memory overhead. What happens is that when any array is initialized then all of the array elements are accessed and hence shadow memory is allocated for the whole array by TSan. But when the array locations are not live after a certain point, then also TSan does not selectively deallocate the shadow memory due to which memory is never deallocated and the overhead is high.

Unsolved problems/potential future work

- To make ARCHER more efficient TSan can be modified further to reason about logical concurrency instead of concurrency of implementation threads to decrease false negatives. Also, selective deallocation can be employed to deallocate the shadow memory for the array elements which are not live to reduce memory overheads.

- The benefit of static analysis depends on the proportion of the sequential regions and data independent loops. If the sequential regions and data independent loops will have large proportion w.r.t. the code then there would be more chance for Sequential and Parallel Blacklisting respectively which will in turn reduce the runtime overhead. These experiments could be done over many case studies to know more about the impact of sequential regions and data independent loops on the runtime overhead.

- Currently the static phase partitions the code into just 2 sections race free or potentially racy. The potentially racy sections can be more fine tuned to find some race free subsections from them and just use the potentially racy sub-section of this potentially racy section during the runtime to increase more performance.

- With later OpenMP versions device constructs will be introduced which will allow the code to run on GPUs also and there would be risk of data races on them also. So, more comprehensive data race detection techniques should be researched and developed as tooling in that area is not much comprehensive.

- More efficient and scalable data race checkers can also be designed for Intel TBBs and Pthread as they are also widely used for parallelism. Although, it may be difficult to create checkers for Pthreads due to unstructured parallelism.

2. ROMP

Paper Summary

The complexity of OpenMP presents a challenge for data race detection. If only static analysis is used then it must be conservative to avoid races but which can lead to more false positives. Hence, dynamic detectors are necessary to use. To precisely detect a data race, the detectors must know about the way how the programming model imposes ordering and mutual exclusion. It has been shown prior that reasoning about ordering and mutual exclusion separately leads to false positives. Hence, hybrid race detectors maintain happens-before orderings, lock sets, and access history for each memory location. The paper introduces ROMP, which is hybrid, on-the-fly, dynamic race detector that tracks logically concurrent tasks instead of thread-level concurrency.
ROMP uses a variant of pre-existing hybrid data race detection algorithm, All-Sets. The difference is in the way of pruning access histories which is essential for efficiency in terms of memory overhead. The pruning is done in 2 ways. If the earlier access was read or earlier and current accesses are write and mutual exclusion entities on earlier access is super set of current access then the earlier access history is removed as if there was data race of some access with earlier one then data race will also occur with current one also. Similarly, on appropriate conditions the current access is not added to access histories. To represent logical concurrency among tasks, instead of thread-based model, ROMP uses OpenMP task graph model. The race detection algorithm needs mechanism for reasoning about concurrency of accesses. ROMP uses a modified version of Offset-span labelling in which each node in the graph has a sequence of labels as on creating a new node or a task it inherits label from parent node and appends label for newly created task increasing the size of vector of labels by 1. ROMP also tracks the data sharing attributes to accurately detect the races. ROMP is implemented as a shared library which contains all the data race detecting algorithms and also maintains a shadow memory for the access histories. Each slot in shadow memory contains access history of a memory address and each slot has a queueing lock to synchronize the updates to access history using which ROMP directly reports a race condition if a thread waiting to update access history for a write access already finds a lock there. ROMP outperforms the most capable OpenMP race detector ARCHER, with better accuracy, precision and recall and less memory overhead when evaluated over DataRaceBench and OmpSCR benchmarks.

Key insights/Strengths

- ROMP uses logical concurrency instead of thread-level concurrency which is used by ARCHER. This helps ROMP to avoid false negatives which are incurred by ARCHER as in the thread-centric approach if conflicting and logically concurrent accesses are performed by same physical thread it will not report those data races although they are present.

- ROMP has about 2.5X less memory overhead than ARCHER which is the best OpenMP data race checker, when evaluated on the OmpSCR benchmarks. This is maybe due to the fact that in ARCHER the shadow memory was not deallocated when not in use but ROMP removes the history accesses based on some heuristics as described briefly in the Paper Summary section.

- The task labelling algorithm employed by ROMP is fairly subtle. Each node in the graph has a sequence of labels. When a new task is created then label of its parent is taken and a new label for this task is appended to it. So, label of a node stores all the information of the path from root to that node. To reason about concurrency just two labels are compared from left to right and the first label where they differ is used to reason about concurrency of associated memory accesses. This modified version of Offset-span labelling avoids the use of complex DAG reachabiliy based algorithms.

- The paper is very detailed in every aspect. It describes all the basic terminologies like happens-before relation, mathematical formulation for conditions needed for a data race, etc. It even covers the algorithms used in a fairly detailed manner with even pseudocode. It even covers the data sharing attributes management algorithm with pseudocode which is just a smaller detail in the complete algorithm. They even added extra benchmarks from their side when evaluating over DataRaceBench benchmarks as they didn't cover all the OpenMP constructs in their benchmarks.

Weaknesses

- ROMP uses an MCS queueing lock for each slot so that if different accesses of same memory address are encountered at a time then only one of them can update the access history at a time. If a thread is waiting to update an access history for a write access because the slot is already locked then ROMP directly reports a data race. But the issue here is that if there is a memory address that is being used frequently by many threads then it will lead to a performance degradation as each time it will get accessed a lock will be taken and others might have to wait for the lock to be free.

- In some programs of DataRaceBench benchmarks, ROMP reported false negatives due to incomplete logical task label when static scheduling was used. Hence, they had to enforce dynamic scheduling using schedule(dynamic) to correctly identify all the data races.

- If an application uses dynamically-loaded shared libraries then ROMP will not do proper checking for all those shared libraries. Using dynamic libraries in a large application is very ordinary and if there would be a scope of data race in any of those libraries then ROMP might fail in detecting data race.

- To track the data attributes of OpenMP like shared, private, etc., ROMP manages states for memory locations that are accessed by using OMPT callbacks. It does so by manipulating bits in the access histories. But when a memory location is deallocated then it uses lazy destruction policy in a way that it doesn't clear the history records instead only marks the locations as deallocated. Although, this a trade-off between time and memory overhead, ROMP doesn't try the non-lazy version i.e. clearing the history records instead of just marking them as deallocated. It does not evaluate the lazy version over different benchmarks; maybe it is possible that there is only a little impact on runtime but memory overhead significantly decreases which can only be confirmed by evaluations.

Unsolved problems/potential future work

- When multiple threads are used then the program execution slows down as ROMP uses mutual exclusion to protect the shadow memory slots. Reader-writer lock can be used so that multiple reads can happen at the same time which will definitely decrease runtime overhead.

- Currently ROMP maintains access histories for those memory locations also which are just read only. These type of memory locations can never be a cause of data race. So, storing access histories for them can be avoided to decrease memory overhead.

- ROMP has to be modified to check all the dynamically-loaded shared libraries also, which it is not currently able to handle.