Runtime Determinacy Race Detection for OpenMP Tasks

Hassan Salehe Matar and Didem Unat

Koç University, Istanbul, Turkey
{hmatar, dunat}@ku.edu.tr

Abstract. One potential problem when writing parallel programs with OpenMP is to introduce determinacy races where for a given input, the program may unexpectedly produce different final outputs at different runs. Such startling behavior can result from incorrect ordering of OpenMP tasks. We present a method to detect determinacy races in OpenMP tasks at runtime. Based on OpenMP program semantics, our proposed solution models an OpenMP program as a collection of tasks with inferred dependencies among them where a task is implicitly created with a *parallel* region construct or explicitly created with a *task* construct. We define happens-before relation among tasks based on such dependencies for determining an execution order when detecting determinacy races. Based on this formalization, we developed a tool, TaskSanitizer, which detects and reports concurrent memory accesses whose tasks do not have common dependencies. Finally, TaskSanitizer works at runtime, has been able to find bugs in micro-benchmarks and it is reasonably efficient to be utilized in a working environment.

1 Introduction

OpenMP 3.0 introduced shared memory task execution model [1] in which programmers specify computations in units called *tasks*, which can be executed by concurrent threads. In OpenMP 4.0 [2], a programmer can specify execution order between tasks through *in* and *out* data dependencies, where a succeeding task waits for the completion of the preceding task's execution. Even though programmers have more flexibility to express various types of parallelism with the new tasking attributes, these new features can introduce subtle bugs if the operational semantics and scheduling policy of the OpenMP runtime are not reasoned about. One of such concurrency bugs is *a determinacy race* which occurs when concurrently executing entities access the same memory location without specified ordering between them and at least one access is a write to that memory location [8, 16, 21, 22]. As a result, a program with determinacy races may produce different final output results at different runs on the same input [18]. Determinacy races are possible if the programmer does not specify necessary dependency between concurrent tasks which access the same memory locations. Since there is no specific order defined by the programmer, the scheduler is free to execute the tasks in any order or concurrently.

The existing state-of-the-art runtime race detection tools for OpenMP such as Archer [3] – and general race detectors [11]– check for proper locking in programs which protects shared memory objects but can fail to detect determinacy races which stem from improper ordering of executions. Protecting memory accesses with critical sections or other explicit locking is not sufficient to avoid determinacy races. Rather, proper ordering of the executing entities is essential to avoid undesirable nondeterminism in OpenMP programs for correctness.

We present an algorithm to detect determinacy races in OpenMP programs by utilizing the concept of OpenMP tasks and their dependencies. Unlike the state-of-the-art race detection tools [3] that rely on *happens-before* model at thread level, we apply *happens-before* model at task level, which provides the advantage of reducing randomness due to scheduling. We implement our algorithm as an open source tool based on compile-time instrumentation through LLVM [15] compiler pass to instrument shared memory accesses in the program. The tool uses the OpenMP Performance Tools API (OMPT) [7] to monitor OpenMP-related events such as task creation, scheduling, and execution. In summary, the main contributions of this paper are:

- A formal definition of the determinacy races and a technique for detecting such races in OpenMP tasks. To our knowledge, no prior work has been done for detecting determinacy races in OpenMP tasks with mixed structures of critical and non-critical sections.
- Determinacy race detection tool for OpenMP called *TaskSanitizer*.
- Evaluation of our method using micro-benchmark applications and comparison of results against a race detection tool for OpenMP programs.

2 Background in OpenMP Tasks

Explicit tasks in OpenMP can be created with the construct *omp task*, which is readily available since OpenMP 3.0 [1]. For each task, OpenMP creates a work block which includes a sequence of program statements and the data environment. This block is set aside to be executed by a thread until the runtime schedules it. Starting with OpenMP 4.0 [2], it is possible to specify execution order among explicit tasks using the *depend* clause, where a programmer specifies input and output data dependencies between tasks. A collection of tasks through dependencies forms an implicit task dependency graph in which a task is not runnable until all its dependencies are satisfied. The runnable tasks can then be scheduled by the OpenMP runtime. If two or more tasks are simultaneously runnable at a given point in time, they can execute in any order or concurrently.

Every part of an OpenMP program executes in a task assigned to one or more threads. For example, implicit tasks can be generated at parallel regions with the OpenMP *parallel* construct and each implicit task is executed to completion by one thread in the thread group of the parallel region [1]. Figure 1 shows a simple OpenMP program, where a default implicit task is created as part of the main program. This task then creates two implicit tasks through the parallel region at line 3. One of these tasks executes the *single* region at line 4, which creates

two explicit tasks t and u at lines 6 and 10, respectively. Both of these tasks have critical sections, in which they set different values to a shared variable i. This example has a determinacy race which is explained in detail in Section 3.

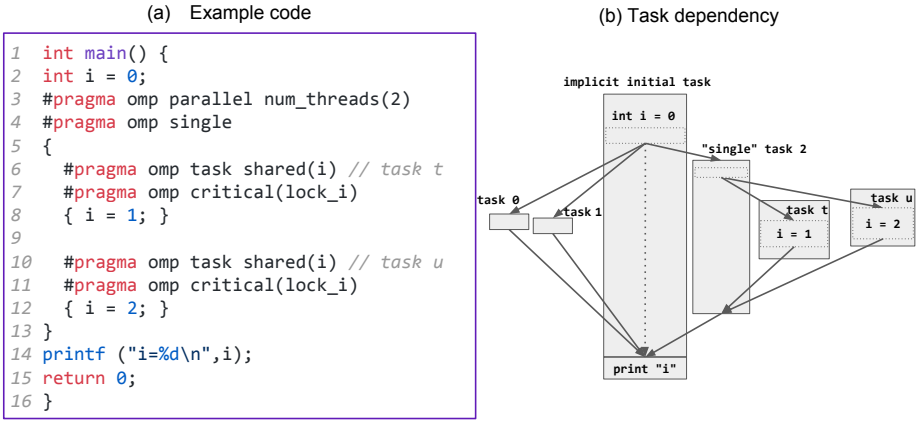

Fig. 1: OpenMP example illustrates explicit and implicit tasks and their logical flow dependency between tasks. The code example has a determinacy race.

3 Determinacy Race Detection

In this section, we first define determinacy races and present motivation on detecting them with the help of an OpenMP example. Then, we formally define a task with its operations and we devise happens-before (HB) relations between these operations for capturing partial ordering among them. Finally we use the defined HB relations to present our algorithm for detecting determinacy races.

3.1 Definition and Motivating Example

Determinacy race occurs between two tasks if the following two conditions are satisfied: *(i)* there is no ordering between these tasks enforced by task dependency, and *(ii)* both tasks access a common shared memory location and at least one access is a write. If simultaneously runnable tasks modify the same memory locations, different scheduling (i.e; order of execution) of these tasks may result in nondeterministic final values on these memory locations.

Many runtime race detection algorithms [9, 23, 24] do not take the notion of dependency into account. They monitor proper synchronization of threads on memory accesses to detect races. In this work, we monitor the proper ordering of tasks and critical sections to ensure that different possible ordering of critical sections in these tasks always generate a single, deterministic final program state. This helps the programmer to notice if nondeterminisim was not intentional.

We have provided a simple OpenMP program in Figure 1, where there is no specified dependency between tasks *t* and *u*. As a result, their critical sections can execute in any order and thus the final result for i can either be 1 or 2 despite the fact that accesses to the shared variable are protected by a common lock. Unless the developer intends the program to behave as such, only one deterministic result is expected. The same issue arises if one of the tasks reads the value of i in a critical region and the other task writes to i. It is worth noting that in a typical program these two tasks might have been created in separate function calls, thus the critical sections may be well far apart from each other and can be easily overlooked.

3.2 Formalizing Task Operations

In order to establish HB relations and set up rules between tasks for detecting determinacy races, we first define relevant task operations:

- `create(t,u)`: task *t* creates task *u*.
- `wait(t,u)`: task *t* awaits termination of task *u* at *taskwait* or at a barrier.
- `read(t,mem)`: task *t* reads value from shared memory location *mem*.
- `write(t,mem,v)`: task *t* writes value `v` to shared memory location *mem*.
- `out(t,u,x)`: signifies dependency from task *t* to task *u* through storage location *x*. Task *t* is the predecessor and *u* is the dependent task.
- `in(u,t,x)`: signifies dependency from task *t* to task *u* through storage location *x*. Task *u* becomes runnable once *t* completes its execution.

Having defined task operations, we elaborate on shared memory accesses and associate them to segments of a task, rather than the task itself. We define a task as an enclosed sequence of unique *tasksegment*s and *synchronization* operations executed together, as shown in Figure 2. A tasksegment is a sequence of consecutive shared memory accesses between two synchronization operations in a task. Therefore, a synchronization operation in a task ends the current tasksegment and a new tasksegment starts at the next shared memory access operation in the task after the synchronization operation. We define synchronization operations as operations which trigger execution among tasks and are `create`, `wait`, `out`, and `in`. For example, Figure 3 shows three tasks (a parent and two child tasks) but contains four tasksegments. In other words, in our formal task operations we differentiate the code bodies (e.g. tasksegment *s1* and tasksegment *s4*) that result from imperfectly nested tasks. Since this is necessary to establish HB relations, we revise the shared memory access operations as follows:

- `read(t,s,mem)` shared memory access that appears in tasksegment *s* where task *t* reads a value from shared memory location *mem*.
- `write(t,s,mem,v)` shared memory access that appears in tasksegment *s* where task *t* writes value `v` to shared memory location *mem*.

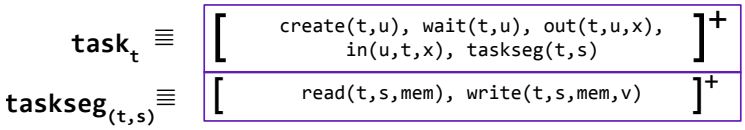

Fig. 2: Defining a task as a sequence of tasksegments (taskseg) and synchronizations

3.3 Happens-before Relations Between Task Operations

For partial ordering of operations in an OpenMP program, we use happens-before (HB) ordering of events [14] by employing dependency among synchronization operations. Happens-before relation is a transitive-closure relation. For given three operations a, b, and c if there is an HB relation from a to b and from b to c, then there is an HB relation from a to c. We will infer this relation while categorizing HB relations between tasks operations. We use symbol $\prec$ to refer to an HB relation in general and use $<_\pi$ to refer to an inferred HB relation due to transitive-closure property.

$$\mathrm{a} \prec \mathrm{b} \wedge \mathrm{b} \prec \mathrm{c} \rightarrow \mathrm{a} <_\pi \mathrm{c}$$

We identify four types of HB edges among operations between tasks. These are (i) an HB relation among memory operations performed within a tasksegment; (ii) between a task and its child task through *create*; (iii) relation between *out* and *in* dependency operations; and (iv) relation at *wait* operation. We then use these HB relations to infer HB relations among tasksegments in tasks.

1. HB by program order: This is the basic type of HB relation where program operations within a tasksegment are ordered according to their execution sequence. Similarly, tasksegments within a task are ordered by program order.

2. HB relation by task dependency: If tasks t and u have a commonly specified data dependency such that u has an input dependency from t, then all tasksegments – as well as their enclosing memory operations – in t happen-before all tasksegments in u.

$$\frac{out(t,u,x) \prec in(u,t,x)}{\forall_{taskseg_{(t,a)}} \forall_{taskseg_{(u,b)}} taskseg_{(t,a)} <_\pi taskseg_{(u,b)}}$$

3. HB relation between a task and its child task: tasksegments of a task which execute before creating a child task happens-before the tasksegments executed in the created child task. For two tasks t and u:

$$\frac{create(t,u)}{\forall_{taskseg_{(t,a)}} taskseg_{(t,a)} <_\pi create(t,u) \rightarrow \forall_{taskseg_{(u,b)}} taskseg_{(t,a)} <_\pi taskseg_{(u,b)}}$$

4. HB relation at taskwait and barrier synchronizations: The last operation of a child task happens before the *taskwait* or implicit barrier synchronization operation of the parent task. Therefore, all tasksegments of such task have HB relation with subsequent tasksegments of the parent task after the wait operation is completed.

$$\frac{wait(t,u)}{\forall_{taskseg_{(t,a)}} wait(t,u) <_\pi taskseg_{(t,a)} \rightarrow \forall_{taskseg_{(u,b)}} taskseg_{(u,b)} <_\pi taskseg_{(t,a)}}$$

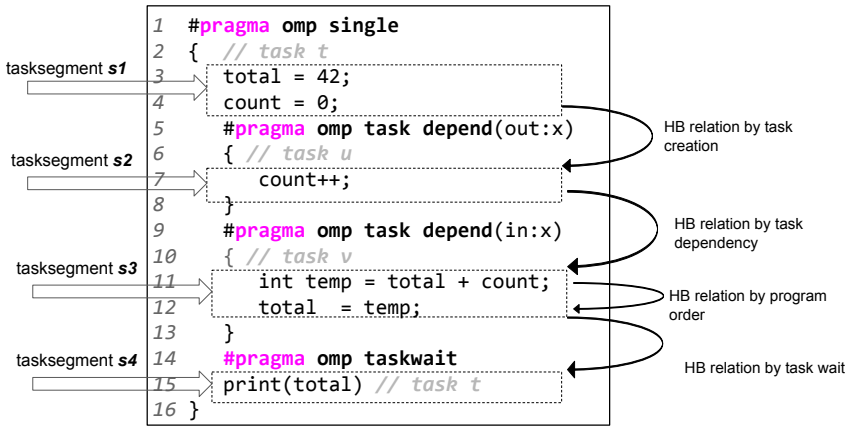

Fig. 3: Example with four categories of HB relation among operations of tasks.

We use example Figure 3 to illustrate the four categories of HB relations. The memory operations at lines 11 and 12 belong to the same tasksegment *s3* and thus are ordered by program order. Moreover, there is an HB relation between memory operations at lines 4 and 7 because their corresponding tasksegments have an HB relation through task creation synchronization operation as task t executing the single region creates an explicit task u at line 5. Moreover, all operations in tasksegment *s2* happen-before all operations in tasksegment *s3* because of specified dependency between tasks u and v. Finally, memory operations in tasksegments *s3* and *s4* happen before the print statement in tasksegment *s4* because of the *wait* synchronization operation at line 14. Without *taskwait*, we would not be able to establish an HB relation between *s4* with *s2* or *s3*.

3.4 Determinacy Race Detection Algorithm

Algorithm 1 provides pseudo-code for determinacy race detection between any two memory operations (α and β) in an OpenMP program. Between lines 4 and 9, it retrieves information of the operations: their task identifiers (IDs), tasksegment IDs as well as the memory addresses they accessed. Then at line 10, the algorithm checks if the operations access the same memory location

and belong to two different tasks and tasksegments. At line 11, it checks if the corresponding tasksegments do not have an HB relation as inferred using the four HB types from Section 3.3. If there is no HB, then it reports a determinacy race bug if one operation is a *write* and the other a *read* at lines 12 and 13. In the case that they both are *write* actions, it reports a determinacy race if they are not commutative (lines 14 - 16).

Algorithm 1 Detecting determinacy race between two shared memory ops

```
1: procedure checkDeterminacyRace(α, β)
2:     Input: α                                   ▷ a shared memory operation
3:     Input: β                                   ▷ another shared memory operation
4:     t ← getTaskID(α)
5:     u ← getTaskID(β)
6:     seg1 ← getTasksegmentID(α)
7:     seg2 ← getTasksegmentID(β)
8:     mem1 ← getMemoryAddress(α)
9:     mem2 ← getMemoryAddress(β)
10:    if mem1 = mem2 and t ≠ u and seg1 ≠ seg2 then      ▷ on different tasks
11:        if not HappensBefore(seg1, seg2) then          ▷ check if no HB
12:            if isWrite(α) ≠ isWrite(β) then            ▷ one write, one read
13:                reportBug(α, β)
14:            else if isWrite(α) and isWrite(β) then     ▷ both write
15:                if not isCommutative(α, β) then        ▷ check commutativity
16:                    reportBug(α, β)
17:                end if
18:            end if
19:        end if
20:    end if
21: end procedure
```

Detecting Commutative Operations: Shared memory accesses can result in falsely detected determinacy races if these accesses involve in commutative arithmetic operations between same-lock critical sections. Two concurrent arithmetic operations on a shared memory location are commutative if their order of execution does not alter the final value produced. For example, if *var += temp1* and *var -= temp2* are in two different same-lock critical sections, then re-ordering them does not affect the final value of *var*. Thus in line 16 of Algorithm 1, we use the formalization of commutativity operation detection proposed in [18] to identify such memory actions and do not report determinacy races on them.

4 Implementation

As shown in Figure 4, we implement our method as a tool that has three main parts (i) instrumentation; (ii) inferring happens-before relation between program operations; and (iii) determinacy race detection at runtime.

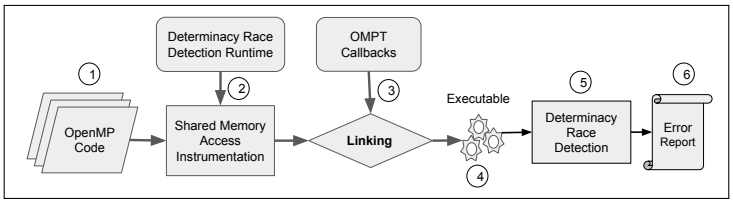

Fig. 4: Showing implementations of TaskSanitizer: architecture and tool flow.

1. **Instrumentation:** We instrument an OpenMP program source code at compile-time through LLVM / Clang infrastructure [15]. The instrumentation injects our determinacy race detection runtime callbacks, which implement Algorithm 1, in step ②. We customize the shared memory instrumentation module of ThreadSanitizer [24] to identify shared memory operations and associated source code line numbers and functions for traceability in case of determinacy races. Moreover, we identify and store program statements which are in critical sections. These are later used by our algorithm to detect commutative operations on potential determinacy races where our tool does not report them if the ordering of those critical sections does not alter the final output.
2. **Constructing HB relations:** To capture HB ordering between tasks and operations, we implemented a module that uses the OMPT interface [7] in step ③ of Figure 4 to register callbacks which capture synchronization operations. First, we locate the implicit tasks as well as explicit tasks defined using the tasking clause for specifying the ordering of program events. Second, task dependencies through *depend* clause as well as custom synchronization idioms such as locks and barriers are located to reason about the happens-before ordering. Finally, we use these operations to infer HB relations between task operations. Moreover, we assign a unique identification to each task and tasksegment at creation, during program execution. This has three advantages (a) Unique ID differentiates different instances of the same task code block or tasksegment executed at different times. (b) A task may run to completion by a single thread or its parts may be scheduled to different threads. Similarly two concurrent tasks may be executed by the same thread. Our approach is transparent from threads, hence regardless which thread(s) execute a task, a unique ID preserves its dependencies with other tasks and avoids false determinacy race alarms. (c) Each tasksegment has the same set of HB meta-data, as opposed to each memory operation, thus unique ID of the tasksegment is used to retrieve HB metadata for each of its memory operations.
3. **Runtime determinacy race detection:** As shown in Figure 4, we link the library we implemented at step ③ to produce the instrumented executable binary, which executes at step ④. At step ⑤ relevant program events are captured at runtime and detection is performed and a bug report is generated in step ⑥. The tool reports a pair of line numbers where a common shared memory location was accessed by concurrent tasks. This pair is helpful for the

developer to revisit the source code and eliminate determinacy races. This module also implements the technique proposed in [18] to check if operations with determinacy races are commutative as they execute in critical sections of the same lock given that their execution order does not affect final output of the program to reduce false positives.

5 Results

We evaluate our tool on nine micro-benchmarks on three categories: (a) the number and nature of determinacy races reported as well as no determinacy races reported in correct programs, (b) detection comparison with Archer [3], (c) the runtime overhead with respect to input size. We first provide a brief summary of the applications before discussing evaluation results. The first five applications are custom implementations with races, accessible through TaskSanitizer[1].

- ***RacyBackgroundExample:*** implements the example in Figure 1. There are two tasks each containing a critical section associated with the same lock. One task sets 1 to shared variable i while the other sets 2 without enforced dependency thus exhibiting a determinacy race as these operations do not commute even though they are in critical sections.
- ***RacyBanking:*** We mimic the motivating banking example in [18]. An initial task sets the account balance to 1000. Then three concurrent tasks access the account balance without specified dependency among them, thus causing three determinacy races and the updates on the account do not commute.
- ***RacyFibonacci:*** This program computes Fibonacci of a given number n using *memoization* technique of caching intermediate results in a shared integer array. A task for n creates two concurrent child tasks to compute Fibonacci of *n-1* and *n-2*, respectively, and each stores its result in the *memoization* array. The task then sums the results from the array after a synchronization barrier with the child tasks. There are determinacy races in this example on five program locations between two concurrent sibling tasks as they access the memoization array without inferred dependency between them.
- ***RacyMapReduce:*** constructs histogram of words from a text file. It splits the input text into four chunks. Then each chunk is processed by *map* tasks. The partial results are merged into a final histogram by *reduce* tasks which are concurrent to each other, exhibiting four determinacy races while inserting new words into the final histogram and updating word counts.
- ***RacyPointerChasing:*** traverses a singly-linked list and creates an explicit task for each node to insert a number to the node for the purpose of forming an arithmetic sequence in the linked-list. In this program, two random nodes in the list mistakenly contain common memory address for storing their terms which breaks the arithmetic sequence. As a result, their corresponding tasks concurrently write values to the memory, causing a determinacy race.

[1] https://github.com/hassansalehe/TaskSanitizer/tree/master/src/benchmarks

– ***sectionslock1-orig-no:*** As part of the DataRaceBench micro-benchmark suite [17], this program creates two parallel sections, which have critical sections in which one section increases a shared variable by 1 and other section increases it by 2. There are no determinacy races because these operations in critical sections commute and our tool does not report a bug.
– ***taskdep1-orig-no:*** As part of DataRaceBench, the program creates two explicit tasks with the first task setting 1 to a shared variable and the succeeding sibling task setting 2. These tasks have specified dependency between them and thus no determinacy races.
– ***taskdep3-orig-no:*** As part of DataRaceBench, this program creates two explicit tasks. The first task has dependency with each of the other sibling tasks which are concurrent to each other. Since the concurrent tasks only read from a shared variable, there is no determinacy race.
– ***taskdependmissing-orig-yes:*** As part of DataRaceBench, this program creates two concurrent explicit tasks which have no dependency in between. They modify a shared variable and thus constitute a determinacy race.

Table 1: Comparing detection results of TaskSanitizer against Archer

				TaskSanitizer	**Archer**
Application	Input size	Number of tasks	Known races	Races found	Races found
RacyBackgroundExample	-	6	1	1	0
RacyBanking	-	11	3	3	2
RacyFibonacci	5	137	8	8	11
RacyMapReduce	-	17	4	4	1
RacyPointerChasing	14	34	1	1	0
sectionslock1-orig-no	-	2	0	0	0
taskdep1-orig-no	-	6	0	0	0
taskdep3-orig-no	-	8	0	0	0
taskdependmissing-orig-yes	-	6	1	1	0 or 1

5.1 Precision Evaluation of TaskSanitizer

Table 1 lists the reported bugs by our tool, TaskSanitizer and number of determinacy races known in advance for micro-benchmarks. In ***RacyBackgroundExample*** two concurrent tasks execute two critical sections which each sets different value to a shared memory location. This exhibits a determinacy race since the tasks do not have HB relation and their memory operations do not *commute* in critical sections. Our tool does not check for commutativity in remaining buggy programs as their operations happen outside critical sections. Even though tasks with critical sections in ***sectionslock1-orig-no*** do have dependency, there is no determinacy race reported because increment operation in these sections *commute*. Finally, our tool does not report false positives in the remaining programs.

5.2 Comparing Detection with Archer

We compare our determinacy race detection results with data race detection results of Archer [3], which is an efficient tool based on ThreadSanitizer for detecting data races. Data race detection in Archer differs from determinacy race

detection in our approach on two essences: (i) It relies on thread-level concurrency and thus it fails to detect races in concurrent tasks scheduled to execute by the same thread. (ii) It aims at detecting violations of locking critical sections which have shared memory accesses whereas our method focuses on different ordering of events leading to determinacy races.

As shown on Table 1, Archer failed to detect races in ***RacyBackgroundExample*** and ***RacyPointerChasing*** despite multiple runs. Archer fails to detect the race in ***RacyBackgroundExample*** because memory operations are protected by a common lock. However, our tool detects determinacy races because the locks do not enforce deterministic ordering and thus the program can produce different results at different runs.

Archer does not detect a race in ***taskdependmissing-orig-yes*** and other buggy programs when concurrent tasks in the program are scheduled to execute with one thread. Therefore, Archer detects the race only if two tasks are executed by different threads whereas our tool detects the determinacy race in the program at all runs. This is because Archer depends on program threads to infer concurrency whereas our approach abstracts away threads and detects determinacy races at task level. Moreover, the number of races it reported on the remaining buggy programs varied from zero to the expected depending on scheduling of concurrent tasks to different threads. However it detected two races in ***RacyBanking*** and did not produce false alarms in correct programs.

5.3 Overhead Evaluation

Even though the focus of this work is the method for detecting determinacy races, we also measured the slowdown of determinacy race detection in the micro-benchmark applications which accept varying input sizes, namely ***RacyFibonacci***, ***RacyMapReduce*** and ***RacyPointerChasing*** as shown in Figure 5. By increasing input size, we calculated execution times of the application without determinacy race detection as well as with detection. We calculated slowdown by dividing detection time by execution time without detection. The determinacy race detection slowdown from this experimental setting ranges from 1.0 to 1.71X, but we plan to evaluate with larger applications in our future work.

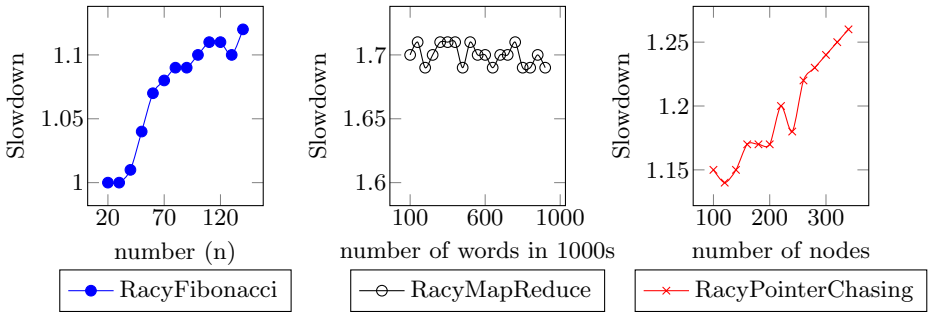

Fig. 5: Slowdown of determinacy race detection in programs as input size increases

6 Related Work

Archer is an efficient tool for detecting data races in OpenMP programs between concurrent threads [3]. Through LLVM, it uses static analysis polyhedral techniques to ignore sequential code and instrument concurrent portion of the program. Then it uses runtime analysis to detect races in those parts by employing *ThreadSanitizer* [24] race detector in the background. In contrast, we detect determinacy races where ordering between concurrent components is missing. Archer may fail to detect such cases and it also misses concurrent tasks executed by the same thread. By building the happen-before relations on tasks rather than threads, we can catch these situations.

Determinacy race detection in [25] targets task-based programming models with *async, finish* and *future* constructs. There are works on detecting determinacy races in a very strict two-dimensional pipeline parallel program structures which restrict task dependency to at most two [6,27]. Other works target determinacy races [8, 16, 21, 22] for structured parallelism programming models like X10 and Habanero. Most work targets data race detection [9,12,19,23,24] which manifest as a result of improper synchronization in programs.

DFinspec [18] proposes a technique for detecting output nondeterminism for Atomic Dataflow (ADF) [10] programs due to missing or improper ordering among tasks. It assumes that all concurrent portions of the program execute in atomic tasks. Unlike ADF, in OpenMP tasks are not atomic, thus the proposed solution in DFinspec would not work on OpenMP programs. The Starsscheck tool [5] identifies inconsistencies in *pragma* annotations for programs written in Starss programs [13]. The tool verifies that the programmer correctly annotates the application by checking the input and output dependencies of tasks. By assuming that a task accesses shared memory through only input dependencies, it fails to detect concurrent tasks accessing shared memory locations that are not specified through input dependencies.

A closely related work [8] proposes an algorithm for detecting *determinacy* races for Cilk programs [4] in which a spawned thread may execute concurrently with parent or sibling threads. These threads may need proper synchronization for shared memory accesses. We target OpenMP tasks where a task becomes runnable when all its dependencies are satisfied. Vechev et. al [26] uses a static sequential analysis to verify determinism for task-based parallel programs by leveraging numerical abstractions. They locate code sections that can execute concurrently and check for dependent memory accesses between those sections.

7 Conclusion

We propose a method to detect determinacy races in OpenMP tasks where unintended missing dependency between tasks can result in nondeterministic execution. We define happens-before relation among tasks based on their dependencies for determining an execution order when detecting determinacy races and implement our algorithm as a tool on top of ThreadSanitizer. We evaluated our

solution with a set of small applications in terms of bug detection and overhead. The tool successfully finds bugs in benchmarks and its efficiency is reasonable.

Acknowledgments:

This work has been funded under the Affordable Safe & Secure Mobility Evolution (ASSUME) project for smart mobility.

Data Availability Statement and Acknowledgments: The datasets and code generated during and/or analysed during the current study are available in the Figshare repository [20]: https://doi.org/10.6084/m9.figshare.6392252

References

1. Openmp 3.0 api, `www.openmp.org/wp-content/uploads/spec30.pdf`
2. Openmp 4.0, `http://www.openmp.org/wp-content/uploads/OpenMP4.0.0.pdf`
3. Atzeni, S., Gopalakrishnan, G., Rakamarić, Z., Ahn, D.H., Laguna, I., Schulz, M., Lee, G.L., Protze, J., Müller, M.S.: Archer: Effectively spotting data races in large openmp applications. In: 2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS). pp. 53–62 (May 2016)
4. Blumofe, R.D., Joerg, C.F., Kuszmaul, B.C., Leiserson, C.E., Randall, K.H., Zhou, Y.: Cilk: An efficient multithreaded runtime system. SIGPLAN Not. 30(8), 207–216 (aug 1995)
5. Carpenter, P.M., Ramirez, A., Ayguade, E.: Starsscheck: A tool to find errors in task-based parallel programs. In: D'Ambra, P., Guarracino, M., Talia, D. (eds.) Euro-Par 2010 - Parallel Processing. pp. 2–13. Springer Berlin Heidelberg (2010)
6. Dimitrov, D., Vechev, M., Sarkar, V.: Race detection in two dimensions. In: Proceedings of the 27th ACM Symposium on Parallelism in Algorithms and Architectures. pp. 101–110. SPAA '15, ACM, New York, NY, USA (2015)
7. Eichenberger, A.E., Mellor-Crummey, J., Schulz, M., Wong, M., Copty, N., Dietrich, R., Liu, X., Loh, E., Lorenz, D.: OMPT: An OpenMP Tools Application Programming Interface for Performance Analysis, pp. 171–185. Springer Berlin Heidelberg, Berlin, Heidelberg (2013)
8. Feng, M., Leiserson, C.E.: Efficient detection of determinacy races in cilk programs. Theory of Computing Systems 32(3), 301–326 (1999)
9. Flanagan, C., Freund, S.N.: Fasttrack: Efficient and precise dynamic race detection. In: Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation. pp. 121–133. PLDI '09, ACM, New York, NY, USA (2009)
10. Gajinov, V., Stipic, S., Unsal, O., Harris, T., Ayguade, E., Cristal, A.: Integrating dataflow abstractions into the shared memory model. In: Computer Architecture and High Performance Computing (SBAC-PAD), 2012 IEEE 24th International Symposium on. pp. 243–251 (Oct 2012)
11. Hong, S., Kim, M.: A survey of race bug detection techniques for multithreaded programmes. Software Testing, Verification and Reliability 25(3), 191–217 (2015)
12. Kuru, I., Matar, H.S., Cristal, A., Kestor, G., Unsal, O.: Parv: Parallelizing runtime detection and prevention of concurrency errors. In: Qadeer, S., Tasiran, S. (eds.) Runtime Verification. pp. 42–47. Springer Berlin Heidelberg (2013)

13. Labarta, J.: Starss: A programming model for the multicore era. In: PRACE Workshop "New Languages & Future Technology Prototypes" at the Leibniz Supercomputing Centre in Garching (Germany) (2010)
14. Lamport, L.: Time, clocks, and the ordering of events in a distributed system. Commun. ACM 21(7), 558–565 (Jul 1978)
15. Lattner, C., Adve, V.: LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In: Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04). Palo Alto, California (Mar 2004)
16. Lee, I.T.A., Schardl, T.B.: Efficiently detecting races in cilk programs that use reducer hyperobjects. In: Proceedings of the 27th ACM Symposium on Parallelism in Algorithms and Architectures. pp. 111–122. SPAA '15, ACM, New York, NY, USA (2015)
17. Liao, C., Lin, P.H., Asplund, J., Schordan, M., Karlin, I.: Dataracebench: A benchmark suite for systematic evaluation of data race detection tools. In: Proceedings of the Intr. Conf. for HPC, Networking, Storage and Analysis. pp. 11:1–11:14. SC '17, ACM, New York, NY, USA (2017)
18. Matar, H.S., Mutlu, E., Tasiran, S., Unat, D.: Output nondeterminism detection for programming models combining dataflow with shared memory. Parallel Computing 71, 42 – 57 (2018)
19. Matar, H.S., Tasiran, S., Unat, D.: EmbedSanitizer: Runtime Race Detection Tool for 32-bit Embedded ARM, pp. 380–389. Springer International Publishing, Cham (2017)
20. Matar, H.S., Unat, D.: Source code and user guide for euro-par 2018 paper: Runtime determinacy race detection for openmp tasks. Figshare (2018). Code., https://doi.org/10.6084/m9.figshare.6392252
21. Raman, R., Zhao, J., Sarkar, V., Vechev, M., Yahav, E.: Efficient data race detection for async-finish parallelism. In: Proceedings of the First International Conference on Runtime Verification. pp. 368–383. RV'10, Springer-Verlag (2010)
22. Raman, R., Zhao, J., Sarkar, V., Vechev, M., Yahav, E.: Scalable and precise dynamic datarace detection for structured parallelism. In: Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation. pp. 531–542. PLDI '12, ACM, New York, NY, USA (2012)
23. Savage, S., Burrows, M., Nelson, G., Sobalvarro, P., Anderson, T.: Eraser: A dynamic data race detector for multi-threaded programs. In: Proceedings of the 16th ACM Symposium on Operating Systems Principles. pp. 27–37. SOSP '97, ACM, New York, NY, USA (1997)
24. Serebryany, K., Iskhodzhanov, T.: Threadsanitizer: Data race detection in practice. In: Proceedings of the Workshop on Binary Instrumentation and Applications. pp. 62–71. WBIA '09, ACM, New York, NY, USA (2009)
25. Surendran, R., Sarkar, V.: Dynamic determinacy race detection for task parallelism with futures. In: Falcone, Y., Sánchez, C. (eds.) Runtime Verification. pp. 368–385. Springer International Publishing, Cham (2016)
26. Vechev, M., Yahav, E., Raman, R., Sarkar, V.: Automatic verification of determinism for structured parallel programs. In: Proceedings of the 17th International Conference on Static Analysis. pp. 455–471. SAS'10, Springer-Verlag, Berlin, Heidelberg (2010)
27. Xu, Y., Lee, I.T.A., Agrawal, K.: Efficient parallel determinacy race detection for two-dimensional dags. In: Proceedings of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming. pp. 368–380. PPoPP '18, ACM, New York, NY, USA (2018)