On-the-fly Data Race Detection with the Enhanced OpenMP Series-Parallel Graph

Nader Boushehrinejadmoradi(✉), Adarsh Yoga, and Santosh Nagarakatte

Rutgers University, New Brunswick, NJ 08901, USA
{naderb,adarsh.yoga,santosh.nagarakatte}@cs.rutgers.edu

Abstract. This paper proposes OMP-RACER, a dynamic apparent data race detector for OpenMP programs. Apparent data races are those races that manifest in a program considering the logical series-parallel relations of the execution. By identifying apparent races, OMP-RACER can detect races that occur not only in the observed schedule but also in other schedules for a given input. Our key contribution is a data structure to capture series-parallel relations between various fragments of an OpenMP program with both structured and unstructured parallelism directives, which we call the Enhanced OpenMP Series-Parallel Graph (EOSPG). OMP-RACER maintains information about previous accesses with each memory access and uses the EOSPG to check if they can logically execute in parallel. OMP-RACER detects more races with similar overheads when compared to existing state-of-the-art race detectors for OpenMP programs.

Keywords: OpenMP · Data races · Series-parallel relations · EOSPG.

1 Introduction

Data races are common in OpenMP programs as with any multithreaded program. Data races can cause non-determinism, make the execution dependent on the memory model, and cause debugging issues. Two accesses are said to constitute a data race if they access the same memory location, one of them is a write, and they can execute in parallel. Data races can be classified into apparent races and feasible races [17]. Data races that manifest when we consider the computation, synchronization, and parallel constructs are termed feasible races. Although there is a large body of work on detecting feasible races [10,20,24], they detect races in a given schedule (*i.e.*, interleaving). Detecting feasible races also requires interleaving exploration either systematically or through prioritization [6,16]. In contrast, data races that occur in an execution of a program primarily considering the parallel constructs, but without taking the actual computation into account, are termed apparent races. An apparent race may not be a feasible race in scenarios where the computation itself may change when the parallel threads are scheduled in a different order. Every apparent race is also a feasible race for Abelian programs [8].

K. Milfeld et al. (Eds.): IWOMP 2020, LNCS 12295, pp. 149–164, 2020.
https://doi.org/10.1007/978-3-030-58144-2_10

To detect apparent races, one needs a data structure that represents the logical series-parallel relations between various fragments of the program. Further, any race detector also needs to maintain access history metadata with each memory location that records previous accesses to that location. Prior work on detecting apparent races has primarily focused on task-parallel programs [9,21,22,27,30], which have structured parallelism. This paper focuses on detecting apparent races in OpenMP programs with both work-sharing and tasking directives.

This paper proposes OMP-RACER, an on-the-fly apparent data race detector for OpenMP programs. To encode the logical series-parallel relations between various fragments of the execution in the presence of both structured and unstructured directives (*e.g.*, taskwait and dependencies), we propose a new data structure that we call the Enhanced OpenMP Series-Parallel Graph (EOSPG). It enhances the OpenMP Series-Parallel Graph (OSPG), which we previously proposed for profiling serialization bottlenecks [4], with support for unstructured directives. Specifically, the EOSPG encodes the nesting depth of the tasks that enables it to capture the logical series-parallel relations for a larger class of OpenMP programs than prior state-of-the-art.

The EOSPG accurately encodes logical series-parallel relations between any two fragments of an OpenMP execution (where a fragment is the longest sequence of serial instructions without any OpenMP directives encountered in the dynamic execution). This logical series-parallel relation encoded by the EOSPG is a property of the program for a given input. It enables OMP-RACER to detect races not just in a given schedule but also in other schedules for a given input. It can alleviate the need for exploring schedules with race detection, which is an advance compared to per-schedule detectors based on vector clocks [10,13,24]. The EOSPG supports a large subset of directives in the OpenMP specification. It still does not support undeferred tasks and their interaction with dependency clauses, which we plan to explore in future work.

Apart from constructing the EOSPG during the execution of the program, OMP-RACER also maintains access history metadata with every memory location. On a memory access, OMP-RACER consults the per-location access history metadata and uses the EOSPG to check if the current access can logically happen in parallel using least common ancestor (LCA) queries (see Sect. 3). OMP-RACER provides two modes: a precise mode and a fast mode. In the precise mode, OMP-RACER detects data races when the program uses taskwait directives with no restrictions. The metadata per-memory location is proportional to the nesting level of tasks. In the fast mode, OMP-RACER performs a quick execution to construct the EOSPG and identifies whether the program uses taskwaits in a fully nested manner, which results in structured parallelism. Subsequently, it performs an execution to detect races by maintaining a constant amount of information per-memory location. In our experiments, the fast mode detects all data races in the DataRaceBench suite [14] without any false positives. The performance overhead of OMP-RACER in its fast mode is comparable to a single execution of Archer [2]. Further, Archer generally requires multiple executions of the same program with the same input to detect a given data race while OMP-RACER does not. OMP-RACER is open source and publicly available [5].

Contributions. This paper proposes EOSPG, a novel data structure, to encode logical series-parallel relations for an OpenMP program with both structured and unstructured directives. This paper presents mechanisms to construct and use the EOSPG to detect data races that occur not only in the observed schedule but also in other possible schedules for a given input.

2 Overview of OMP-RACER

This section provides an overview of race detection with OMP-RACER. We use the program in Fig. 1(a) that uses different OpenMP directives to compute the sum of an array in parallel to illustrate OMP-RACER. It uses worksharing (*i.e.*, `single` and `for`) and tasking directives (*i.e.*, `task`) to add parallelism to the program. This example has a data race due to insufficient synchronization between child tasks and the implicit task executing the single directive (lines 12–24 in Fig. 1(a)). The thread encountering the taskwait directive only waits for its child tasks to complete but does not wait for its descendant tasks, which results in unstructured parallelism [19]. The dynamic execution trace with the memory accesses generated when the program is executed on a machine with two threads is shown in Fig. 1(b). The two accesses involved in the data race are executed by the same thread in this schedule. However, OMP-RACER can detect this race because it uses the logical series-parallel relation.

Enhanced OpenMP Series-Parallel Graph. OMP-RACER executes on-the-fly with the program and constructs the EOSPG during program execution. EOSPG is an extension of the OSPG [4]. The EOSPG is an ordered directed acyclic graph (DAG) that captures the dynamic execution of an OpenMP program as a set of program fragments. A fragment is the longest sequence of instructions in the dynamic execution of the program between two OpenMP directives. Each fragment executes serially in a thread or a task. By design, each program fragment is a leaf node in the EOSPG. The intermediate nodes of the EOSPG encode the series-parallel relation between the leaf nodes. EOSPG captures the logical series-parallel relations between any pair of fragments in the program for a given input. Given any two fragments, we can identify if they may execute in parallel by performing a least common ancestor (LCA) query between the leaf nodes corresponding to the two fragments.

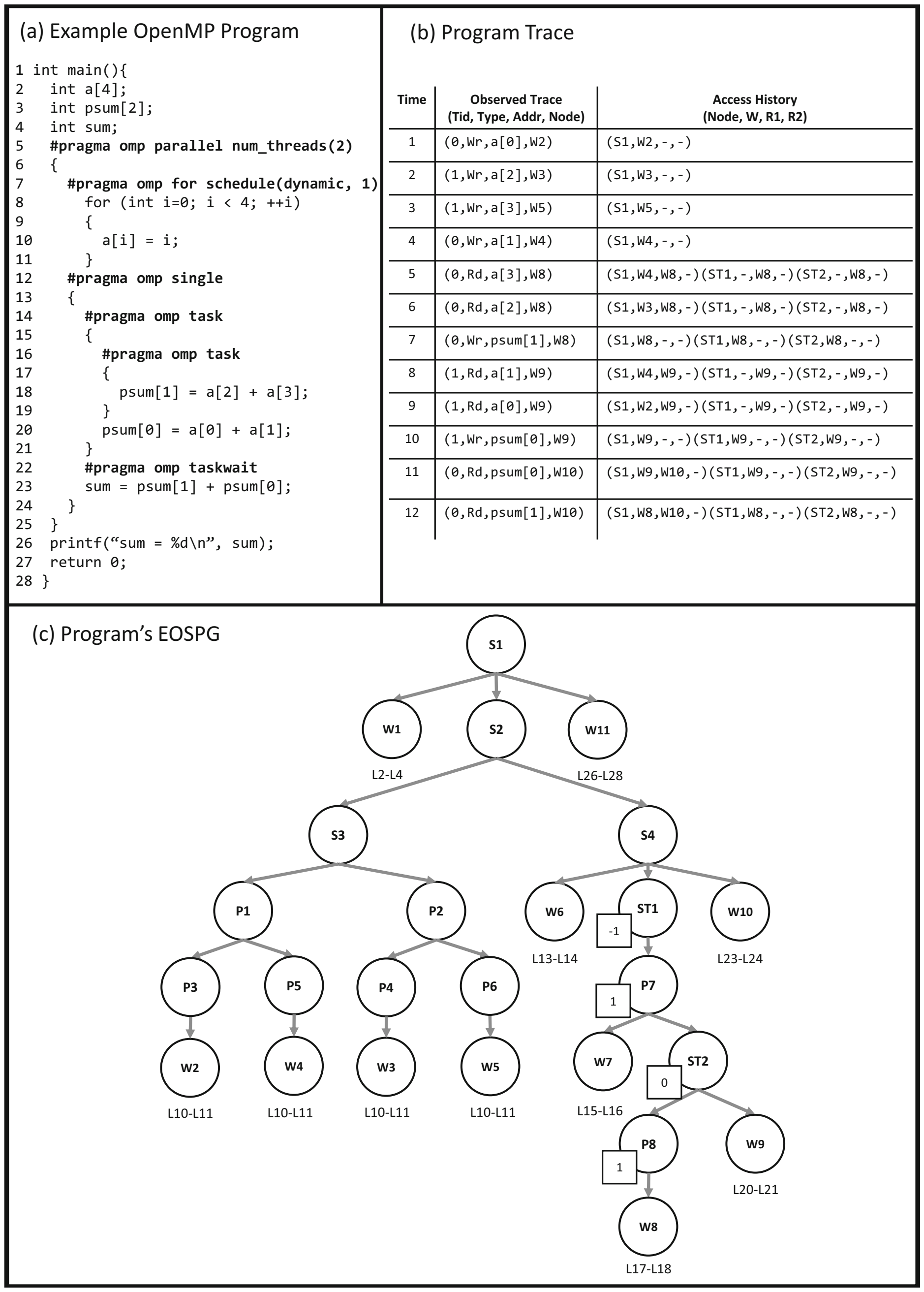
Nodes in an EOSPG. The EOSPG generated for the example program is shown in Fig. 1(c). Each W-node represents a fragment of the execution. The intermediate nodes can be one of the following types: S-node, P-node, or an ST-node (see Sect. 3). The subtree under the S-node executes in series with the siblings and their descendants to the right. Similarly, the subtree under the P-node executes in parallel with the siblings and their descendants on the right. In the EOSPG, ST-nodes captures the fact that the subtree under the ST-node has encountered a taskwait directive. As taskwait only serializes a task's immediate children, it is necessary to count the nesting depth of the tasks. Hence, each ST-node and P-node maintain a value (*e.g.*, *st_val*) to account for the nesting depth. Each P-node contributes a value of 1 to the nesting depth. Each ST-node that has seen a taskwait serializes the immediate children and nullifies the contribution of one P-node under it. Hence, the value of the ST-node starts at 0, and upon encountering a taskwait, it changes to -1.

Checking if Two Accesses can Execute in Parallel. In the absence of ST-nodes and dependencies, two W-nodes, W_i and W_j, where W_i is to the left of W_j, logically execute in parallel if the left child of the least common ancestor (LCA) of W_i and W_j on the path to W_i is a P-node. In the presence of ST-nodes, the procedure is slightly more involved (see Sect. 3). When the left child of the LCA on the path to W_i is an ST-node, we compute the sum of the values related to nesting depth maintained with each ST-node and P-node from W_i to the left child of the LCA. If this sum is greater than zero, then the two nodes execute in parallel. Otherwise they execute serially.

In Fig. 1(c), W-nodes `W2` and `W5` logically execute in parallel because the left child of the LCA node `S3` is the P-node `P1`. Intuitively, these are parallel chunks of a dynamic for loop. A pair of W-nodes, `W8` and `W10`, in Fig. 1(c) execute in parallel because the left child of the LCA node (*i.e.*, `S4`) is the ST-node `ST1` and the sum of *st_val* values from `W8` to `ST1` is 1, indicating a logical parallel relation. In contrast, W-nodes `W9` and `W10` execute in series because the left child of the LCA node (*i.e.*, `S4`) is the ST-node `ST1`. However, the sum of *st_val* values from `W9` to `ST1` is 0. Intuitively, the taskwait on line 22 serializes their execution.

Metadata for Data Race Detection. To detect races, OMP-RACER maintains access history metadata with each memory location. To store access histories, it has two modes: a fast mode and a precise mode. In the fast mode, it performs a complete execution of the program to first check if all taskwaits are fully nested (*i.e.*, they create a taskgroup, where each parent task waits for its child tasks with a taskwait), and there are no critical sections. In such cases, it treats all ST-nodes as S-nodes and maintains two parallel reads and a write with each memory location. In precise mode, OMP-RACER maintains additional information about two reads and writes with each ST-node that is present on the path from the W-node to the root of the EOSPG. The execution has an apparent race if the current operation happens in parallel with the previous conflicting operations to the same memory location in the access history metadata associated with the root node or the access history metadata associated with each of the ST-nodes on the path from the current node to the root of the EOSPG.

Illustration of Race Detection. Figure 1(b) provides the dynamic execution trace and the updates to the access history metadata for each memory location using the precise mode for the example program in Fig. 1(a). The write operation to `psum[1]` at line 18 of the example program corresponds to the W-node, `W8`. Upon this write operation, the metadata associated with `psum[1]`



(a) Example OpenMP Program

```
1 int main(){
2   int a[4];
3   int psum[2];
4   int sum;
5   #pragma omp parallel num_threads(2)
6   {
7     #pragma omp for schedule(dynamic, 1)
8       for (int i=0; i < 4; ++i)
9       {
10        a[i] = i;
11      }
12    #pragma omp single
13    {
14      #pragma omp task
15      {
16        #pragma omp task
17        {
18          psum[1] = a[2] + a[3];
19        }
20        psum[0] = a[0] + a[1];
21      }
22      #pragma omp taskwait
23      sum = psum[1] + psum[0];
24    }
25  }
26  printf("sum = %d\n", sum);
27  return 0;
28 }
```

(b) Program Trace

Time	Observed Trace (Tid, Type, Addr, Node)	Access History (Node, W, R1, R2)
1	(0,Wr,a[0],W2)	(S1,W2,-,-)
2	(1,Wr,a[2],W3)	(S1,W3,-,-)
3	(1,Wr,a[3],W5)	(S1,W5,-,-)
4	(0,Wr,a[1],W4)	(S1,W4,-,-)
5	(0,Rd,a[3],W8)	(S1,W4,W8,-)(ST1,-,W8,-)(ST2,-,W8,-)
6	(0,Rd,a[2],W8)	(S1,W3,W8,-)(ST1,-,W8,-)(ST2,-,W8,-)
7	(0,Wr,psum[1],W8)	(S1,W8,-,-)(ST1,W8,-,-)(ST2,W8,-,-)
8	(1,Rd,a[1],W9)	(S1,W4,W9,-)(ST1,-,W9,-)(ST2,-,W9,-)
9	(1,Rd,a[0],W9)	(S1,W2,W9,-)(ST1,-,W9,-)(ST2,-,W9,-)
10	(1,Wr,psum[0],W9)	(S1,W9,-,-)(ST1,W9,-,-)(ST2,W9,-,-)
11	(0,Rd,psum[0],W10)	(S1,W9,W10,-)(ST1,W9,-,-)(ST2,W9,-,-)
12	(0,Rd,psum[1],W10)	(S1,W8,W10,-)(ST1,W8,-,-)(ST2,W8,-,-)

Fig. 1. (a) Example OpenMP program with a write-read apparent data race on variable psum[1] at lines 18 and 23. (b) The execution trace of the example program when executed with two threads. The first column specifies the ordering of the observed trace. The second column specifies the memory access as a 4-tuple comprised of thread id, memory access type, memory access location, and the W-node performing the access. The third column illustrates the access history maintained by OMP-RACER for the corresponding memory access as a 4-tuple with the node identifier, W-node corresponding to the latest write, and two W-nodes corresponding to two parallel read accesses (- denotes the empty set). (c) The program's EOSPG. The code fragment each W-node represents is shown below it. The square boxes next to some EOSPG nodes represent the value indicating the nesting depth in the presence of taskwaits.

is updated to include `W8`. The path from the root of the EOSPG to `W8` has two ST-nodes: `ST1` and `ST2`. In addition, the metadata corresponding to `psum[1]` is also updated to include `W8` for each ST-node on the path to the root node, resulting in three entries for `psum[1]` in the access history (as illustrated at time 7 in Fig. 1(b)). Eventually, the program execution reaches the taskwait directive at line 22 in Fig. 1(a). OMP-RACER updates the *st_val* of node `ST1` from 0 to -1 to record the presence of a taskwait. Later, when the implicit task executing the single directive performs a read operation on `psum[1]` at line 23 in Fig. 1(a), the W-node representing the current read is `W10`. OMP-RACER retrieves the metadata for `psum[1]`, which includes access histories corresponding to the root node of the EOSPG and ST-nodes `ST1` and `ST2`. Since the path from `W10` to the root node of the EOSPG does not contain any ST-nodes, only the metadata entry corresponding to the root node of the EOSPG, `S1`, is updated to include the current read access (time 12 in Fig. 1(b)). Next, OMP-RACER looks for possible data races by checking if the current read operation happens in parallel with any previous write accesses recorded in the access history metadata. In our example, OMP-RACER checks if the current read access may happen in parallel with the previously recorded write operation corresponding to W-node `W8`. In this case, the LCA of `W8` and `W10` is `S4`. The left child of `S4` on the path to `W8` is an ST-node `ST1`. Next, OMP-RACER computes the sum of the *st_val* values from `W8` to `ST1`, which evaluates to 1. Thus, `W8` and `W10` may execute in parallel, and one of the operations is a write operation, which results in OMP-RACER reporting an apparent race on the memory access to `psum[1]`.

3 OMP-RACER Approach

The goal of OMP-RACER is to detect apparent data races that manifest not just in a given schedule but also in other schedules for a given input. OMP-RACER constructs the Enhanced OpenMP Series-Parallel Graph (EOSPG) to represent series-parallel relations, maintains access history metadata with each memory location, and checks them to catch races.

Enhanced OpenMP Series-Parallel Graph. EOSPG is a data structure that captures series-parallel relations between various fragments of an OpenMP execution in the presence of both structured and unstructured directives. It builds on our prior work, the OpenMP Series-Parallel Graph (OSPG) [4]. Specifically, the OSPG assumed that taskwaits are fully nested. According to the OpenMP specification [19], the taskwait directives need not be fully nested. EOSPG is an enhancement of OSPG to handle directives that can result in unstructured parallelism such as taskwaits and other features such as dependencies.

Definition. EOSPG is a directed acyclic graph (DAG), $G = (V, E)$, where the set V consists of four types of nodes, W-nodes, S-nodes, P-nodes, and ST-nodes. Thus, $V = V_w \cup V_p \cup V_s \cup V_{st}$. The set of edges, $E = E_{pc} \cup E_{dep}$, where E_{pc} denotes the parent-child edges between nodes and E_{dep} denotes the dependency edges. The EOSPG has a root S-node which has a unique directed path consisting of only E_{pc} edges to all other nodes. A node's depth is defined as the number of edges on the path consisting of E_{pc} edges from the root node to it. Nodes with the same parent are referred to as sibling nodes. E_{dep} edges are between two sibling nodes. Thus an E_{pc} edge between a pair nodes, (v_1, v_2), establishes a parent-child relation between the two nodes, where v_1 is the parent node of v_2. Moreover, sibling nodes in an EOSPG are ordered from left to right, which corresponds to the logical ordering of operations in the program.

In contrast to the OSPG, the EOSPG has a new type of node (ST-nodes). Further, each P-node and ST-node maintains additional information to encode the nesting depth that is required to correctly identify series-parallel relations in the presence of the taskwait directive. The state is maintained with each node is called *st_val*, which is an integer in $\{-1, 0, 1\}$.

W-node. Similar to the OSPG, a W-node represents a serial fragment of dynamic execution in the program. By construction, a W-node is always a leaf node in the EOSPG. A fragment either starts from the beginning of the program or when the execution encounters an OpenMP directive. The fragment continues until the program ends, or it reaches another OpenMP directive. In the absence of any OpenMP directives in the program's execution, the entire program is serially executed. Hence, the EOSPG of a sequential program consists of a single W-node that is a direct child of the root S-node. A W-node has an *st_val* of zero.

S-node and P-node. These nodes encode the logical series-parallel relations between W-nodes. An S-node establishes a serial relation (whereas a P-node establishes a parallel relation) between all its descendant W-nodes and all right siblings and their descendant W-nodes. A P-node and an S-node have an *st_val* of one and zero, respectively. The *st_val* of a P-node is one because a P-node creates a parallel strand of execution and contributes a level to nested parallelism.

ST-node. This node also encodes the logical series-parallel relations between W-nodes. Unlike S-nodes or P-nodes, the logical series-parallel relation between W-nodes in the subtree under an ST-node and its right siblings and their descendants depends on whether there has been a taskwait and the nesting depth of the node. Effectively, for an ST-node, its descendant W-nodes are partitioned into two subsets. First, W-nodes that execute serially with all right siblings and the descendants of the ST-node. Second, W-nodes that run in parallel relative to the right siblings and their descendants of the ST-node. The OSPG did not have any ST-nodes [4]. We added this node to the OSPG to enable capturing the logical series-parallel relations in the presence of OpenMP directives that do not fall under structured parallelism. ST-nodes are used whenever creating a P-node or an S-node is not sufficient to capture the logical series-parallel relations between program fragments.

Construction of the EOSPG. A program's EOSPG is constructed incrementally and in parallel during program execution. Each executing thread adds nodes to a subtree of the EOSPG. Different threads will operate on different subtrees of the EOSPG and updates can be done with limited use of synchronization.

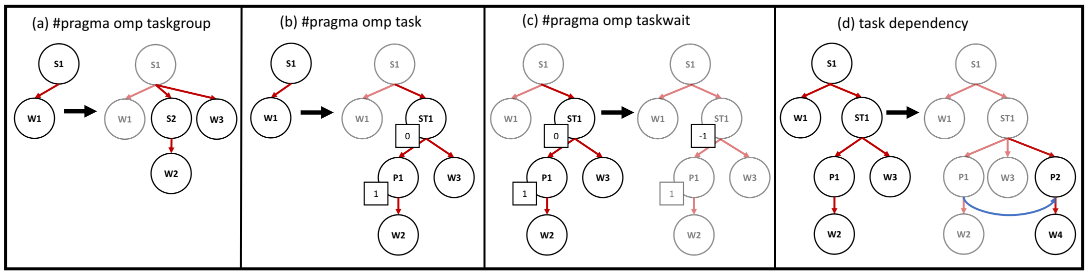


Fig. 2. EOSPG construction for different OpenMP directives. The nodes before encountering a directive are greyed out after the EOSPG is updated.

Except for the handling of the ST-nodes, the construction of the EOSPG is similar to the construction of the OSPG [4]. Here, we highlight the changes to the construction algorithm to capture the logical series-parallel relation that were not supported in the original OSPG design. Figure 1(c) illustrates the EOSPG for the program in Fig. 1(a) after the program completes execution.

Handling Task Synchronization Directives in the EOSPG. OpenMP supports task-based parallelism using the task directive. During program execution, when the currently running task, whether implicit or explicit, encounters a task directive, the current task creates a new child task, becoming its parent task. The child task may execute in parallel with the continuation of the parent task. Moreover, OpenMP provides several options to synchronize task execution. These options include the taskgroup directive, the taskwait directive, and task dependencies.

Taskgroup Directive. A taskgroup directive enforces a serial ordering between the structured block associated with the taskgroup and the code fragments that execute after the taskgroup. Namely, the code following the taskgroup waits for the completion of all created tasks and their descendants within the taskgroup's structured block. As illustrated in Fig. 2(a), the EOSPG captures this serial relation by adding an S-node, `S2`, when the program trace encounters the beginning of a taskgroup. All the fragments in the taskgroup are contained in the subtree rooted at node `S2`. Moreover, the fragment executing after the taskgroup will be the right siblings of newly created S-node, depicted as W-node `W3` in Fig. 2(a). Therefore, creating the S-node `S2` captures the serial relation between all fragments executing within the taskgroup and all the fragments executing after it.

Taskwait Directive. A taskwait directive specifies a serial ordering between the task encountering the directive and its children. However, unlike the taskgroup directive, a taskwait does not enforce a serial ordering with the parent task and its grandchildren and descendant tasks. While fully nested taskwaits produce a behavior similar to that of a taskgroup, it becomes more challenging to correctly capture series-parallel relations when taskwaits are not fully nested.

To capture the series-parallel relations induced by the taskwait directive correctly, we also need to take the nesting level into account. Further, during the dynamic execution, we do not know whether the execution will see a taskwait directive in the future. Hence, whenever a parent task encounters a task directive and creates its first child task in the program or spawns a new task after a taskwait, we add an ST-node, followed by a P-node to the program's EOSPG as illustrated in Fig. 2(b). We create an ST-node at this point in the execution because we do not know a priori the nesting level of the newly created child task and whether at each nesting level, including the current one, the execution will encounter a taskwait directive. The subsequent P-node captures the parallel relation between the newly created child task and its sibling tasks, if any.

An ST-node partitions the W-nodes under its subtree into two subsets because a taskwait does not serialize all the W-nodes in the subtree. (1) W-nodes that execute serially with all right siblings of the ST-node and their descendants. (2) W-nodes that run in parallel relative to the right siblings of the ST-node and their descendants. To determine if a W-node is a member of the first or the second subset, we use the nodes' *st_val* values on the path to the ST-node. Intuitively, *st_val* values on the path to the ST-node, capture the nesting level and the number of encountered taskwaits. Whenever an ST-node is created, its *st_val* is initially set to zero (Fig. 2(b)). If later during the execution, the task that created the ST-node encounters a taskwait, we capture this information by setting the *st_val* of the corresponding ST-node to -1 (Fig. 2(c)). To capture the nesting level under an ST-node's subtree, the newly created P-node, which is the immediate child of the ST-node, will have an *st_val* of one.

Key Invariant in the Presence of ST-nodes. For a pair of W-nodes, (W_i, W_j), where W_i is under the subtree of ST-node ST_k and W_j is either a right sibling or a descendant of a right sibling of ST_k, the pair of W-nodes execute in parallel if the sum of the *st_val* values of the EOSPG nodes on the path from ST_k to W_i is a positive integer. Otherwise, the two W-nodes execute in series.

Task Dependencies. In OpenMP, sibling tasks logically execute in parallel. However, with task dependencies, OpenMP supports user-defined ordering between sibling tasks. We capture the serial ordering produced by task dependencies in the EOSPG by adding E_{dp} edges between P-nodes that correspond to dependent tasks. For example, consider two sibling tasks $t1$ and $t2$ where $t2$ is dependent on $t1$. The EOSPG captures this task dependency as follows. By construction, each task has a corresponding P-node in the EOSPG, labeled as `P1` and `P2` in Fig. 2(d). The E_{dp} edge, $(p1, p2)$, captures the underlying task dependency. The dependency of two sibling tasks can be checked by looking for a path comprised of E_{dp} edges between the corresponding P-nodes.

To check if a pair of W-nodes, (`W2`, `W4`) in Fig. 2(d), execute in series due to a task dependency, we first check if they are sibling tasks. This is accomplished by computing the LCA. If the LCA node is not an ST-node, then they are not sibling tasks, and the dependency edges are ignored. A task dependency only serializes the sibling tasks, which does not imply the serialization of its nested descendants. If the LCA is a ST-node, then we identify the corresponding P-nodes to check the if pair of W-nodes are at the same nesting level; we use the sum of the *st_val* values of EOSPG nodes on the path from `W2` to the LCA.

The pair of W-nodes execute in series if this sum equals to 1, and there exists a directed path comprised of E_{dp} edges between the two sibling P-nodes.

Checking Series-Parallel Relations. Using the EOSPG, we can check if two W-nodes logically execute in parallel. Given a pair of W-nodes, W_l and W_r where W_l is to the left of W_r, this procedure is as follows. (1) Compute the least common ancestor (LCA) of the two nodes W_l and W_r. (2) Identify the left child of the LCA on the path to W_l. If this left child is a S-node or a W-node, then the two nodes logically execute in series. (3) If the left child of the LCA on the path to W_l is a P-node, check if the two W-nodes under consideration are serialized by dependency edges. Identify the child of the LCA on the path to W_r. If there is a directed path between these two P-nodes and are at the same nesting level, they execute in series. Otherwise, they logically execute in parallel. (4) If the left child of the LCA on the path to W_l is a ST-node, check if the two nodes are serialized by fully nested taskwaits. Determine the count of the *st_val* values on the path from W_l to the child of the LCA. If this count is greater than 0, then two nodes execute in parallel. Otherwise, they execute in series.

Metadata. OMP-RACER maintains access history metadata with each shared memory address. In the fast mode when the taskwait directives are properly nested and the program does not use locks, then OMP-RACER maintains three W-nodes corresponding to the previous write and two previous reads (R_1 and R_2) per-memory location similar to prior work [22,30]. The invariant maintained by OMP-RACER is that if any future memory access is involved in a data race with prior n reads to the same memory location $R_{1..n}$, then it will also have a data race with R_1 or R_2. This invariant is maintained by choosing (R_1, R_2) such that, $L = LCA(R_1, R_2)$, is closer to the root node than $L' = LCA(R_1, R_K)$ or $L'' = LCA(R_2, R_K)$ for any $R_K \in R_{1..n}$.

In the precise mode, OMP-RACER stores a number of W-nodes per shared memory location that increases proportionally to the size of the lockset and the number of active ST-nodes in the program. When the EOSPG has ST-nodes, maintaining only two read accesses for the entire program to detect the first data race is no longer sufficient. Consider two parallel reads, (R_1, R_2) that occur in tasks that are at an outer nesting level of the program and a parallel read that occurs in a task at an inner nesting level, R_3. Maintaining the earlier invariant results in keeping (R_1, R_2) in the access history. Leading to potentially missing data races that involve R_3. For example, this could happen if the outer nesting level is synchronized with a taskwait that does not synchronize the inner nesting level, as depicted in Fig. 1(a) (lines 14–23). Hence, OMP-RACER maintains two additional reads and one write for each active ST-node in the program. To detect data races in the presence of locks, OMP-RACER tracks the set of locks held before an access (*i.e.*, lockset [8]) and maintains up to two W-nodes for prior parallel reads (R_1, R_2) and up to two W-nodes for prior parallel writes (W_1, W_2) for each lockset per memory location [22,30].

Metadata Updates and Checks on Each Access. On every memory access, the metadata for that memory location is retrieved, checked for races, and is updated. In the precise mode, the metadata is a list of access history entries. Each entry is uniquely identified by the lockset and a node of the EOSPG (*i.e.*, either root node or a ST-node). Each entry consists of a lockset, a node of the OSPG (*i.e.*, a root or an ST-node), two reads, and two write operations. On a memory access with a lockset (l_c) in a W-node W_c, the metadata is checked as follows. OMP-RACER iterates over the list of access histories to retrieve a 6-tuple (lockset, node, R_1, R_2, W_1, W_2). For every entry, if the intersection of the lockset of the access history and l_c is non-empty, OMP-RACER checks if the current node W_c and previous reads/writes in the access history are conflicting and can logically execute in parallel. If so, it reports an apparent race. Subsequently, the metadata is updated as follows. Starting from W_c, traverse the EOSPG to the root node and identify all ST-nodes on the path to the root node. For every ST-node encountered and the root node, OMP-RACER creates or retrieves a new entry from the list of access history entries that corresponds to the current lockset. This entry contains four W-nodes. If the LCA of W_c and one of the existing nodes is closer to the root than the existing LCA of the nodes, then W_c is added to the access history. Otherwise, information about W_c is already subsumed by the existing information in the access history.

In the fast mode, OMP-RACER runs the program once to construct the EOSPG and to identify whether the program uses locks and uses taskwaits in a properly nested manner. During the construction of the EOSPG, when an ST-node completes execution, if the *st_val* of the all ST-nodes is -1, then the program contains properly nested taskwait directives. In the subsequent race detection execution, the access history per-memory location contains two reads and a write operation. A current access is an apparent data race if it is conflicting with the prior access in the access history and can happen in parallel.

Scaling to Long Running Applications. Our approach can scale to long running applications because it is not necessary to maintain the entire EOSPG in memory. The EOSPG and the access history metadata can be cleared at the end of the parallel directive. Further, any EOSPG node can be deallocated even before the end of the parallel directive when it does not have any reference in the access history metadata space.

4 Experimental Evaluation

Prototype. OMP-RACER prototype supports C/C++ OpenMP programs. It uses LLVM-10's OpenMP runtime and the OMPT interface to construct the EOSPG. It also includes an LLVM pass to instrument memory accesses. OMP-RACER constructs a program's EOSPG and performs data race detection on-the-fly during execution. OMP-RACER has two modes: a precise and a fast mode. The precise mode detects data races even when the program uses locks and imposes no restriction on how taskwaits are used in the program, which can have significant overheads. In the fast mode, OMP-RACER first checks if the program has fully nested taskwaits. If so, it uses a constant amount of metadata per memory location in the subsequent execution for race detection.

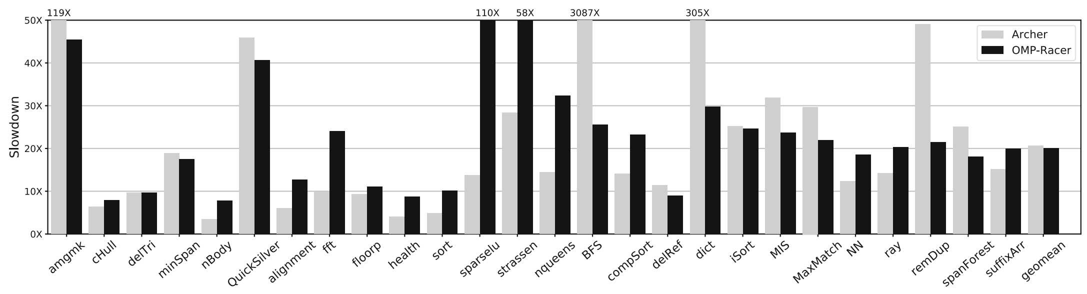


Fig. 3. small Performance slowdown of OMP-RACER and Archer with various PBBS, BOTS, and Coral application suites.

It is significantly faster. We report all evaluation results with the fast mode. OMP-RACER prototype is publicly available [5].

Benchmarks. We evaluate the detection abilities of OMP-RACER with DataRaceBench1.2.0 [14]. OMP-RACER does not support the target directive and SIMD parallelization, yet. Out of 116 programs, 106 do not contain these directives. To measure performance overheads, we use a suite of 26 OpenMP applications from Coral, BOTS, and PBBS benchmarks suites. We performed all experiments on a Ubuntu 16.04 machine with a 16-core Xeon 6130 processor running at 2.1 GHz and with 32 GB of memory. We use the latest version of Archer [2] with LLVM-10, which is the state-of-the-art for OpenMP programs, to compare the detection abilities and overheads with OMP-RACER.

Detection Ability. We compare OMP-RACER and Archer's effectiveness in detecting races using DataRaceBench. OMP-RACER detects data races in all the 106 programs from a single execution and does not produce any false positives (*i.e.*, 100% detection rate). As we detect apparent races, OMP-RACER detects races that do not manifest in a particular schedule. Archer did not detect many of the races in a single execution. When we ran Archer with multiple threads and multiple times, it detected 95% of the races. We observed that Archer misses races in some programs (*e.g.*, DRB013) when executed with a low number of threads. Archer does not precisely capture the semantics of task synchronization and task dependency, which results in false negatives. In summary, OMP-RACER is more effective in detecting races compared to Archer.

Performance Overheads. Figure 3 reports the performance overhead of OMP-RACER and Archer with our performance applications. The runtime overhead of OMP-RACER in its fast mode, on average, is 20×. The overhead of Archer is 21×. When the program performs significant recursive decomposition (*e.g.*, with `Strassen` and `SparseLU`), OMP-RACER has higher runtime overhead compared to Archer. The height of the EOSPG is proportional to the nesting level of the program. An increase in height can increase the cost of performing LCA queries, which results in higher overheads.

We also measured the impact of increasing the number of threads and the costs of EOSPG creation. The overhead of OMP-RACER decreases with the increase in the number of threads with scalable applications as the instrumentation code is executed in parallel. The average cost of constructing the EOSPG is 1.12× on average compared to the baseline program without any instrumentation. Hence, performing an initial execution to check if the taskwaits are properly nested in the fast mode is inexpensive compared to the cost of overall race detection.

5 Related Work

Race detection has been widely studied for parallel programs. These include both approaches that rely on static analysis [3,7,26] and dynamic analysis [9–12,15,18,21–25,30]. Static analysis tools can detect races for all inputs. However, they report false positives due to conservative analyses. Among dynamic analysis tools, Eraser [23] uses locksets to identify data races. Subsequent approaches have used happens-before relation with vector clocks [13] to detect races [10,24]. ThreadSanitizer [24] makes numerous trade-offs to scale vector-clocks to large applications.

Our work is inspired by prior approaches that use logical series-parallel relations for fork-join programs, which include labeling [11,15,18] and construction of series-parallel graphs [1,9,21,22,25,28–30]. OMP-RACER proposes a novel series-parallel graph (*i.e.*, EOSPG) to accurately capture series-parallel relations induced by the directives according to the OpenMP specification.

Among OpenMP tools for dynamic race detection, ROMP [11] and Archer [2] are closely related. ROMP [11] expands upon offset-span labeling to support OpenMP directives, including tasking and task synchronization. Asymptotically, the operations in the EOSPG are comparable to ROMP's offset-span labeling since the length of labels and the depth of the EOSPG grow proportional to the nesting level of the program. However, the public prototype of ROMP is not mature to run with large applications. Archer [2] builds upon ThreadSanitizer by extending it to support OpenMP semantics. As Archer is a per-schedule detector, it is necessary to run an application with Archer multiple times and with multiple thread counts to detect races. Compared to Archer, OMP-RACER is able to detect more races that not only occur in the observed schedule but also in other possible schedules for a given input from a single execution.

6 Conclusion

This paper makes a case for detecting apparent races in OpenMP programs using logical series-parallel relations. The Enhanced OpenMP Series-Parallel Graph precisely models logical series-parallel relations for a significant portion of the OpenMP specification, which makes it useful for building numerous performance analysis and debugging tools. It supports both work-sharing and tasking directives. The ability to detect races not only in the observed schedule but also in other possible schedules for a given input with OMP-RACER can alleviate the need for repeated executions and interleaving exploration. Our preliminary results with OMP-RACER are promising and we plan to support more features from the OpenMP specification in the future.

Acknowledgments. We thank the reviewers for their feedback. This paper is based on work supported in part by NSF CAREER Award CCF–1453086, NSF Award CCF-1908798, and NSF Award CCF-1917897.

References

1. Agrawal, K., Devietti, J., Fineman, J.T., Lee, I.T.A., Utterback, R., Xu, C.: Race detection and reachability in nearly series-parallel dags. In: Proceedings of the Twenty-Ninth Annual ACM-SIAM Symposium on Discrete Algorithms, p. 156–171. SODA 2018 (2018)
2. Atzeni, S., et al.: Archer: effectively spotting data races in large OpenMP applications. In: 2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS), pp. 53–62. IPDPS 2016 (2016)
3. Basupalli, V., et al.: ompVerify: polyhedral analysis for the OpenMP programmer. In: Chapman, B.M., Gropp, W.D., Kumaran, K., Müller, M.S. (eds.) IWOMP 2011. LNCS, vol. 6665, pp. 37–53. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-21487-5_4
4. Boushehrinejadmoradi, N., Yoga, A., Nagarakatte, S.: A parallelism profiler with what-if analyses for OpenMP programs. In: Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis, pp. 16:1–16:14. SC 2018 (2018)
5. Boushehrinejadmoradi, N., Yoga, A., Nagarakatte, S.: Omp-racer data race detector (2020). https://github.com/rutgers-apl/omprace
6. Burckhardt, S., Kothari, P., Musuvathi, M., Nagarakatte, S.: A randomized scheduler with probabilistic guarantees of finding bugs. In: Proceedings of the 15th International Conference on Architectural Support for Programming Languages and Operating Systems, pp. 167–178. ASPLOS (2010)
7. Chatarasi, P., Shirako, J., Kong, M., Sarkar, V.: An extended polyhedral model for SPMD programs and its use in static data race detection. In: Ding, C., Criswell, J., Wu, P. (eds.) LCPC 2016. LNCS, vol. 10136, pp. 106–120. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-52709-3_10
8. Cheng, G.I., Feng, M., Leiserson, C.E., Randall, K.H., Stark, A.F.: Detecting data races in cilk programs that use locks. In: Proceedings of the 10th ACM Symposium on Parallel Algorithms and Architectures, pp. 298–309. SPAA (1998)
9. Feng, M., Leiserson, C.E.: Efficient detection of determinacy races in cilk programs. In: Proceedings of the 9th ACM Symposium on Parallel Algorithms and Architectures, pp. 1–11. SPAA (1997)
10. 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 (2009)
11. Gu, Y., Mellor-Crummey, J.: Dynamic data race detection for OpenMP programs. In: Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis, pp. 61:1–61:12. SC 2018 (2018)
12. Jannesari, A., Bao, K., Pankratius, V., Tichy, W.: Helgrind+: an efficient dynamic race detector. In: 2009 IEEE International Symposium on Parallel Distributed Processing, pp. 1–13 (2009)

13. Lamport, L.: Time, clocks, and the ordering of events in a distributed system. Commun. ACM, pp. 558–565 (1978)
14. 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 International Conference for High Performance Computing, Networking, Storage and Analysis, pp. 11:1–11:14. SC 2017 (2017)
15. Mellor-Crummey, J.: On-the-fly detection of data races for programs with nested fork-join parallelism. In: Proceedings of the 1991 ACM/IEEE Conference on Supercomputing, pp. 24–33. Supercomputing (1991)
16. Nagarakatte, S., Burckhardt, S., Martin, M.M., Musuvathi, M.: Multicore acceleration of priority-based schedulers for concurrency bug detection. In: Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation, pp. 543–554. PLDI (2012)
17. Netzer, R.H.B., Miller, B.P.: What are race conditions?: Some issues and formalizations. ACM Lett. Program. Lang. Syst. pp. 74–88 (1992)
18. Nudler, I., Rudolph, L.: Tools for the efficient development of efficient parallel programs. In: Proceedings of the 1st Israeli conference on computer system engineering (1988)
19. OpenMP Architecture Review Board: Openmp 5.0 complete specification, November 2017. https://www.openmp.org/wp-content/uploads/OpenMP-API-Specification-5.0.pdf
20. Pozniansky, E., Schuster, A.: Efficient on-the-fly data race detection in multithreaded c++ programs. In: Proceedings of the Ninth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, pp. 179–190. PPoPP (2003)
21. Raman, R., Zhao, J., Sarkar, V., Vechev, M., Yahav, E.: Efficient data race detection for Async-finish parallelism. In: Proceedings of the 1st International Conference on Runtime Verification, pp. 368–383. RV (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 (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 (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 (2009)
25. Utterback, R., Agrawal, K., Fineman, J.T., Lee, I.T.A.: Provably good and practically efficient parallel race detection for fork-join programs. In: Proceedings of the 28th ACM Symposium on Parallelism in Algorithms and Architectures. SPAA 2016 (2016)
26. Ye, F., Schordan, M., Liao, C., Lin, P.H., Karlin, I., Sarkar, V.: Using polyhedral analysis to verify OpenMP applications are data race free. In: 2018 IEEE/ACM 2nd International Workshop on Software Correctness for HPC Applications (Correctness), pp. 42–50 (2018)
27. Yoga, A., Nagarakatte, S.: Atomicity violation checker for task parallel programs. In: Proceedings of the 2016 International Symposium on Code Generation and Optimization, pp. 239–249. CGO (2016)
28. Yoga, A., Nagarakatte, S.: A fast causal profiler for task parallel programs. In: Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering, pp. 15–26. ESEC/FSE 2017 (2017)

29. Yoga, A., Nagarakatte, S.: Parallelism-centric what-if and differential analyses. In: Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation, p. 485–501. PLDI 2019 (2019)
30. Yoga, A., Nagarakatte, S., Gupta, A.: Parallel data race detection for task parallel programs with locks. In: Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, pp. 833–845. FSE (2016)