GT-Race: Graph Traversal Based Data Race Detection for Asynchronous Many-Task Parallelism

Lechen Yu and Vivek Sarkar(✉)

College of Computing, Georgia Institute of Technology, Atlanta, GA, USA
{lechen.yu,vsarkar}@gatech.edu

Abstract. *Asynchronous Many-Task* (AMT) parallelism is growing in popularity because of its promise to support future platforms with new heterogeneity and resiliency requirements. It supports the construction of parallel programs with fine-grained tasks, thereby enabling portability across a wide range of platforms. However, applications written for AMT parallelism still remain vulnerable to data races, and existing data race detection tools are unsuitable for AMT programs because they either incur intractably large overheads or are limited to restricted task structures such as fork-join parallelism.

In this paper, we propose GT-Race, a new graph-traversal based data race detector for AMT parallelism. It leverages the computation graph data structure, which encodes the general happens-before structures in AMT programs. After introducing a baseline algorithm for data race detection, we propose key optimizations to reduce its time and space complexity, including the *epoch adjacency list* to compress the computation graph representation, the *reachability cache* combined with *depth filtering* to reduce the number of unnecessary traversals, and *bounded race detection* to limit the range of data that is monitored. The impact of these optimizations is demonstrated for nine benchmark programs written for the Open Community Runtime (OCR), an open source AMT runtime that supports point-to-point synchronization and disjoint data blocks.

Keywords: Debugging and correctness tools · Data race detection
Asynchronous many-task parallelism

1 Introduction

With the ever-increasing complexity of modern computing architectures (e.g., large numbers of heterogeneous processing units, multi-level hierarchical memories, and high-bandwidth interconnect networks), applications on these machines must leverage the architectural complexity to perform efficiently. Although widely used high-performance parallel runtimes (e.g., PThreads, MPI, and OpenMP) provide comprehensive low-level interfaces to help programmers

M. Aldinucci et al. (Eds.): Euro-Par 2018, LNCS 11014, pp. 59–73, 2018.
https://doi.org/10.1007/978-3-319-96983-1_5

leverage the underlying architecture, the programmers have to tune the application to select the best granularity manually. In addition, manually tuned applications are not performance-portable. In order to mitigate these two problems, Asynchronous Many-Task (AMT) runtimes [1] (e.g., Cilk [2], Habanero-C (HC) [3], Realm [4] and Open Community Runtime (OCR) [5]) become a new trend in HPC area. AMT runtimes hide low-level details of architecture from programmers. When writing a parallel program executing on top of AMT runtimes (we refer to it as AMT program in this paper), programmers only need to split the program logic into *tasks*, a hardware agnostic abstraction of code snippets that can execute independently on any process unit, and specify the dependences among tasks. AMT programs can achieve higher performance with less programming and tuning efforts, compared to MPI implementations [4].

Although AMT parallelism alleviates the difficulty of writing efficient and portable parallel programs, AMT programs are still prone to data races, a notorious error in parallel programs. A data race occurs when a parallel program issues two unordered memory accesses to the same location, such that at least one of the accesses is a write.

There has been a lot of past work on detecting data race automatically at runtime [6–11]. But all of them suffer from at least one of the following four limitations:

- Incurring a space overhead that is proportional to the square of the number of dynamic tasks.
- Leveraging a locking scheme to detect data races, which introduces false positives for parallel programs that use synchronization primitives other than locks.
- Forcing the parallel program to execute in sequential order, which wastes the available hardware parallelism.
- Detecting data races based on restricted parallel structures.

Currently, there does not exist any data race detection algorithm with tractable overhead that can support the general AMT parallelism.

In this paper, we propose GT-Race, a new graph-traversal based data race detector for AMT parallelism. It leverages the computation graph data structure [12], which encodes the general happens-before structures in AMT programs. After introducing a baseline algorithm for data race detection, we propose key optimizations to reduce its time and space complexity, including the *epoch adjacency list* to compress the computation graph representation, the *reachability cache* combined with *depth filtering* to reduce the number of unnecessary traversals, and *bounded race detection* to limit the range of data that is monitored. The impact of these optimizations is demonstrated for nine benchmark programs written for the Open Community Runtime (OCR), an open source AMT runtime that supports point-to-point synchronization and disjoint data blocks.

The rest of this paper is organized as follows: In Sect. 2 we discuss a case study to show how an AMT program can encounter data races. Based on the clarified notion of data race, we illustrate the graph traversal based race detection

algorithm and several optimizations. We discuss the implementation of GT-Race in Sect. 3. We evaluate the performance of GT-Race in Sect. 4. Section 5 summarizes some related works about data race detection, and finally in Sect. 6 we briefly conclude with some possible directions for future works.

2 GT-Race

In this section, we introduce GT-Race, an on-the-fly dynamic data race detector for AMT programs. First, we illustrate the architecture of GT-Race, and then we present several optimizations applied by GT-Race, which reduce the space overhead and improve the efficiency of data race detection.

2.1 Computation Graph and Data Races

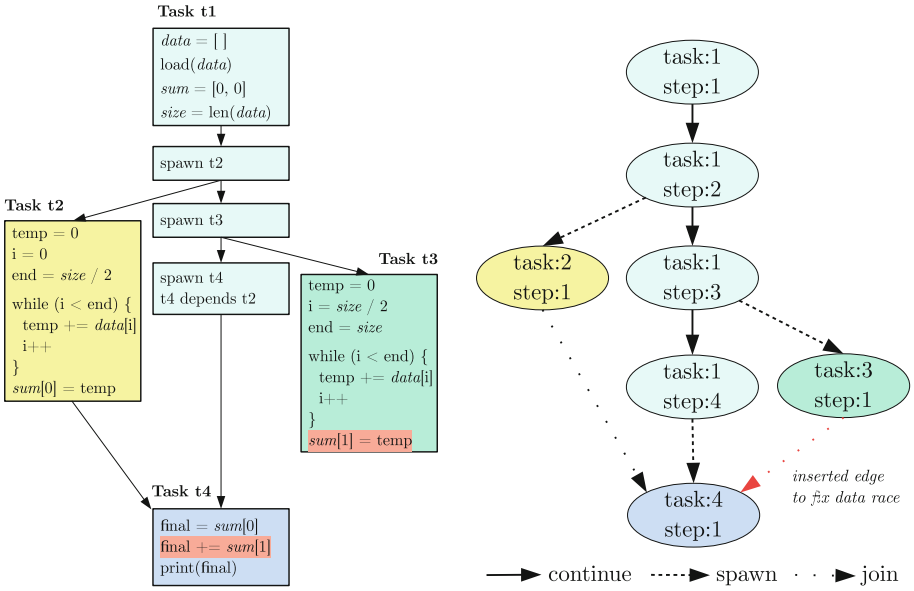
The constructs in an AMT program can be divided into three classes according to their semantics:

- *spawn constructs*: They submit a new task to the underlying AMT runtime. The new task may execute in parallel with the caller.
- *synchronization constructs*: They specify dependences among tasks that can impact task scheduling in the AMT runtime. A task will be ready for execution after all its specified dependences are satisfied [13].
- *computation constructs*: All other constructs not related to task management belong to this class.

In a computation graph for a dynamic AMT program execution, each node denotes a *step* [12], an arbitrary sequential computation belonging to a task, which ends with a spawn construct or a synchronization construct, and each edge denotes an ordering constraint among the involved tasks. For any two steps in the computation graph, the happens-before relation holds if and only if there exists a directed path between the two steps. When two unrelated steps issue memory accesses to the same shared variable, and at least one step writes to that variable, the two memory accesses create a data race. Figure 1 shows how data races can occur in an AMT program. Figure 1a is a buggy implementation of parallel summation, and Fig. 1b is the corresponding computation graph. To fix the program, we need to link $t3$ to $t4$ by a join edge to guarantee that $t4$ will observe $t3$'s result.

2.2 Overview

Figure 2 shows GT-Race's architecture. It comprises three components: computation graph, shadow memory, and data race checker. The computation graph and shadow memory update dynamically according to the AMT program's runtime behaviors. These two components record happens-before relations and historical memory accesses respectively. Our implementation of shadow memory is similar



(a) A problematic implementation of parallel summation containing a data race

(b) Corresponding computation graph for the AMT program in Figure 1a

Fig. 1. Case study

to [14]. In order not to miss potential data races, the shadow memory records the latest write and all reads after the write for each shared memory location. The key module of GT-Race is the data race checker, which leverages the data in the computation graph and shadow memory to analyze the order of memory accesses. For each read (write) to a shared variable, the data race checker carries out a graph traversal on the computation graph to verify the read is causally ordered after the concurrent write(concurrent write and all concurrent reads). If the graph traversal fails to find any paths between the concurrent memory accesses, GT-Race will output the associated debug information of the conflicting memory accesses and the computation graph to help programmers figure out the cause of the detected race.

2.3 Epoch Adjacency List: A Compressed Representation for Computation Graph

Since the computation graph is a sparse graph, a straightforward way to store it is by using an adjacency list. Due to the large number of steps a task may contain, it is not memory efficient to allocate a list for each step. Further, explicitly storing all steps and edges may also slow down the graph traversal because of the redundant continue edges.

Since all steps in the same task execute sequentially, we can determine their execution order in constant time by numbering steps belonging to the same task (we refer to these numbers as *epochs*). Inspired by this idea, we propose the *epoch adjacency list*, a compressed storage for computation graphs. In an epoch adjacency list, each task occupies an edge list that records incoming spawn and join edges. For each edge in the edge list, the associated cell marks its source step using the source task ID and epoch.

2.4 Optimization: Reachability Cache

The original graph traversal algorithm is a breadth-first search that loops through the computation graph to find out directed paths between the two memory accesses. It is inefficient due to failing to utilize the locality in the AMT program. For two tasks that both access a shared variable, it is highly possible they have other common variables so that the race checker will check their causal ordering multiple times during the program execution.

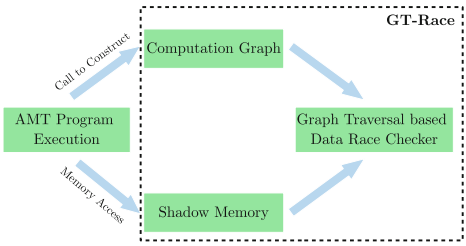


Fig. 2. GT-Race architecture

In order to reuse the results of previous graph traversal, we store them in a reachability cache and look up the cache during graph traversal to avoid redundant explorations. This can be implemented by adding cache lookup and cache update operations to the graph traversal algorithm. If there exists a record in the cache, the graph traversal terminates immediately. Otherwise, the graph traversal proceeds to check the next enqueued step. This optimized graph traversal algorithm is shown in Algorithm 1.

2.5 Optimization: Depth Filtering

Since the time overhead of graph traversal is dominated by the number of nodes and edges it accessed, traversing a large computation graph in a brute-force manner is always time-consuming. Every time after accessing a task, the algorithm will loop through all incoming edges and enqueue connected dependent tasks to avoid omitting any potential path to the expected destination, which leads to the inefficiency of graph traversal. In order to mitigate the time overhead, we introduce a guidance *depth* to help prune irrelevant tasks when looping through incoming edges. For any task t, its *depth* is defined by these two formulas:

- $depth(t_0) = 0$, where t_0 is the entry point of the whole program.
- $depth(t) = Max(depth(t_i)) + 1$, where t_i is a dependent task of t.

```
Data: Computation Graph CG, Reachability Cache cache, Operation
      op1, op2
Result: If op1 happens before op2, return true, otherwise return false
// Bounded Race Detection
if !isBoundedMemory(op2.getMemoryAddress()) then
    return true
end
dst ← CG.getStep(op1), src ← CG.getStep(op2), queue ← ∅
queue.push_back(src)
while !queue.empty() do
    curr ← queue.next(), queue.findNext()
    if curr.task = dst.task ∧ curr.epoch ≥ dst.epoch then
        cache.update(src, dst)
        return true
    end
    // Reachability Cache
    if cache.reachable(curr, dst) then
        cache.update(src, dst)
        return true
    end
    for prev in curr.incomingEdges do
        if !queue.contain(prev) then
            // Depth Filtering
            if prev.depth >= dst.depth then
                queue.push_back(prev)
            end
        end
    end
end
return false
```

Algorithm 1. Revised Graph Traversal

The calculation of depth executes along with the computation graph construction and it does not increase the time complexity. According to the definition, we can deduce Theorem 1 (Depth-Reachability Theorem) and apply it to filter tasks.

First we introduce Lemma 1 and prove its correctness. Then we derive Theorem 1 on the basis of Lemma 1. For simplicity, we assume that all tasks in an AMT program are indivisible, so that each node in the corresponding computation graph represents a single task. It is straightforward to extend the theorem to the step level.

Lemma 1. *For two tasks a, b, if these exists a directed edge in the computation graph from a to b (we denote the edge as* $a \rightarrow b$*), then* $depth(a) < depth(b)$.

Proof. We need to consider two cases:

- Suppose b is the entry point of the program, then b has no preceding tasks, which is contrary to the assumption of Lemma 1.
- Suppose b is not the entry point of the program. Then $depth(b) = Max(depth(t_i)) + 1$, for all predecessors, t_i of b (with edges $t_i \rightarrow b$). So $depth(b) \geq depth(a) + 1$. Hence, the lemma statement is true.

Theorem 1 (Depth-Reachability Theorem). *For two tasks t_i, t_j, if $depth(t_i) \geq depth(t_j)$, then there exists no directed path from t_i to t_j in the computation graph.*

Proof. We prove Theorem 1 by contradiction. Suppose there exists two tasks a, b such that $depth(a) \geq depth(b)$ and there is a path $a \rightarrow t_1 \rightarrow t_2 \ldots t_n \rightarrow b$ in the computation graph. By Lemma 1, we know $depth(a) < depth(t_1)$, $depth(t_1) < depth(t_2) \ldots depth(t_n) < depth(b)$. So $depth(a) < depth(b)$, which contradicts the assumption. Theorem 1 is thus proved by contradiction.

2.6 Optimization: Bounded Race Detection

Apart from the graph traversal, the majority of time and space overhead comes from the shadow memory. Since GT-Race allows the AMT program to execute in parallel, all threads have to access shadow memory with proper synchronizations when they try to record a memory access or get previous memory accesses. These synchronizations are indispensable for the correctness, but they slow down GT-Race's execution.

For better performance and higher accuracy, in GT-Race, we bound the range of data race detection by programmers' knowledge. Since programmers have a full understanding of the AMT program, they are eligible to point out error-prone shared variables. We add an additional option for GT-Race which allows programmers to mark these variables before launching GT-Race. GT-Race will only record memory accesses and carry out data race detection for marked shared variables and ignore the accesses to other variables.

Line 2 in Algorithm 1 shows how bounded race detection works after capturing a memory access. Before starting a graph traversal, the algorithm first checks the desired address of the memory access to see whether it falls in the range of marked variables. If the memory access is to an outside memory location, the algorithm returns true immediately since programmers assume the accessed memory location will not be involved in any data race. The memory address check in line 2 avoids needless graph traversal during the program execution, which is beneficial for efficiency.

3 Implementation

We have developed a prototype implementation of GT-Race (see Fig. 3 for the architecture), based on the algorithm in Sect. 2. GT-Race works as a back-end

tool of Intel Pin[1], a dynamic instrumentation framework that monitors the program execution and inserts callbacks for certain operations such as *construct calls* and *memory accesses*. These callbacks record parameters of operations at runtime and pass them to GT-Race. GT-Race will call corresponding modules to analyze the collected data.

The prototype is designed for Open Community Runtime (OCR) [5], an open-source AMT runtime that supports point-to-point synchronizations. An OCR program consists of three basic objects: (a) Event Driven Task (EDT) (b) data block (c) event.

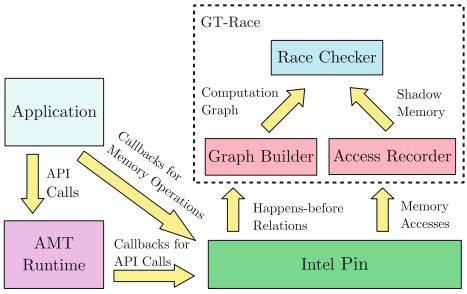


Fig. 3. Prototype architecture

An EDT is the basic execution unit that performs its computation asynchronously. It may have dependences on other EDTs and events. Once all its dependences are satisfied, an EDT can run non-preemptively without being interrupted by other EDTs. A data block is a chunk of consecutive memory managed by the OCR runtime automatically. It is the only way to share data among EDTs and may have various access modes (e.g. read-only, read-write, exclusive write, constant). Although data blocks in read-only and constant modes are supposed to be data race free, it is still possible to introduce data races for these blocks since the OCR runtime will not prevent EDTs from issuing write operations to these data blocks. In order not to miss any data race, we take all data blocks into consideration when detecting data races. However, it is also possible to constrain GT-Race so that it only performs data race detection for a specified subset of data blocks. Event is a synchronization object used to coordinate the activity of EDTs. The semantics is similar to that of a semaphore or latch. An EDT linking to an event, e, through its outgoing edges must wait for the termination of all EDTs linking through e's incoming edges.

As shown in Fig. 3, the inserted callbacks hide the internal details of OCR objects from GT-Race. They instead record operations on OCR objects as general happens-before relations and memory accesses that GT-Race can handle. Callbacks tackling API calls are injected into the OCR runtime. They treat both EDTs and events as tasks (an event can be viewed as a no-op task created solely for the purpose of synchronization) and dependences as directed edges among corresponding tasks. Callbacks for memory operations are weaved into the OCR application. When the application executes memory operations on data blocks, associated callbacks will convert them into equivalent memory read/write operations on shared memory locations. The separation of data collection and data race detection avoids unnecessary modifications to GT-Race when applying it to a new AMT runtime.

[1] https://software.intel.com/en-us/articles/pin-a-dynamic-binary-instrumentation-tool.

4 Performance Evaluation

4.1 Environment and Benchmarks

To evaluate the performance of GT-Race, we carried out several experiments using the OCR benchmarks. All experiments were conducted on an Intel workstation with a 24-core Intel Xeon E5-2667 processor and 125 GB of memory, running 64-bit Ubuntu 15.04. We performed experiments on nine OCR benchmarks from the OCR app repository[2]. These benchmarks are either scientific computing programs or mini-apps derived from real-world applications. All nine OCR benchmarks were compiled using GCC 4.9.2 with -O3 optimization level, and executed on top of a customized OCR v1.1 runtime with 24 threads. No data races were detected in these benchmarks used for performance evaluation, but we separately tested our tool for correctness with synthetically introduced data races. Note that the performance of our algorithm is not impacted by the absence or presence of data races.

Though we compare the technical aspects of our approach with related work in Sect. 5, we did not find any implementation of related work that could be used to obtain useful performance comparisons with GT-Race. For example, direct use of the vector clock approach is not practical for AMT parallelism because it would require that each task have its own entry in every vector clock.

4.2 Space Overhead of GT-Race

Table 1 contains dynamic statistics for each benchmark, when executed with a standard input from the OCR repository. Furthermore, the "Memory Usage of CG" columns show the space overhead of the computation graph with different storage strategies. We see that the memory space used by the optimized epoch adjacency list is only 29.20%–37.85% of that used by the unoptimized adjacency list representation. The improvement in memory usage is due to the implicit storage of steps and continue edges in the epoch adjacency list. *UTS* generates the largest computation graph, which spawns more than 400,000 tasks with millions of dependences at runtime. The corresponding computation graph is around 32 MB, which demonstrates the memory efficiency of epoch adjacency list.

4.3 Performance of GT-Race

Summary of Results. Table 2 lists the uninstrumented execution time and overhead of data race detection for each benchmark. All timing measurements are the geometric mean of 10 runs. We use separate columns in Table 2 to analyze the performance and the effectiveness of different optimizations in GT-Race. All listed slowdowns are relative to the uninstrumented running times in the "Base Time" column.

[2] https://xstack.exascale-tech.com/git/public/apps.git.

Table 1. Dynamic benchmark statistics. The first four columns contain the benchmark name, along with the numbers of tasks, events, and dependences created when executing the benchmark on a standard input in the OCR repository. The next two columns give the computation graph size in bytes for the unoptimized case, and the optimized case using epoch adjacency lists. The last column shows the ratio of the optimized size to the unoptimized size.

Benchmark	Tasks	Events	Dependences	Memory usage of CG (bytes)		
				Original	Epoch adjacency list	Ratio
Cholesky	222	605	1,101	81,504	30,848	37.85%
FFT	9	9	38	2,560	896	35.00%
Fibonacci	364,179	242,785	1,213,925	82,546,936	29,134,224	35.29%
QuickSort	3,937	7,871	19,678	1,385,360	503,776	36.36%
SmithWaterman	6,401	19,683	51,521	3,511,192	1,241,680	35.36%
UTS	302,014	111,116	1,692,399	104,689,464	33,688,464	32.18%
RSBench	30,033	50	766,540	43,648,232	12,745,968	29.20%
XSBench	36,835	52	898,874	51,222,232	14,972,176	29.23%
LCS	9,817	24,578	74,505	4,997,760	1,742,400	34.86%

Performance of Computation Graph Construction. The "CG" column reports the overhead of GT-Race when only constructing the computation graph. The geometric mean overhead is 1.11× which is not significant. Since we utilize a lock-free data structure to store the computation graph in GT-Race, it reduces overhead due to unnecessary synchronizations and can also handle a large computation graph efficiently.

Performance of Shadow Memory. The "CG + SM" column shows the overhead of GT-Race when tracking shadow memory but performing no graph traversals for data race detection. Although the geometric mean overhead is 4.95×, *RSBench* has an overhead greater than 10×. Since each instance of shadow memory has to synchronize concurrent accesses from multiple threads to maintain a consistent historical record, GT-Race sacrifices some performance for correctness. But the slowdown is acceptable compared to existing work.

Performance of Data Race Detection. The last two "Slowdown" columns compare the effectiveness of optimizations for graph traversal. With the help of the reachability cache, GT-Race completed every benchmark's test in 4 min and incurred 7.77× slowdown on cache usage. We also list the statistics on the cache usage in Table 3. The cache miss ratio is less than 2% for all benchmarks except *Fibonacci* and *UTS*. Besides, reachability cache also helps reduce the number of accessed steps during graph traversal. For all benchmarks except *RSBench* and *XSBench*, the average (arithmetic mean)[3] number of accessed steps is much smaller compared to the size of computation graph.

[3] We use geometric mean for ratios, and arithmetic mean for absolute counts such as accessed steps.

Depth filtering further reduces the slowdown of data race detection. For *UTS* and *RSBench*, it reduces the slowdown from 10.22× to 5.50× and 83.19× to 80.86×. For other benchmarks, the improvement is not substantial. Although *FFT* sees an increased overhead, the slowdown is close to the version without depth filtering. The reason for the performance for depth filtering in FFT is that its computation graph is not large and reachability cache already improves the performance of graph traversal, which causes the overhead of calculating depths to overshadow the performance gain.

Table 2. Benchmark results. Columns 4–6 contain slowdowns relative to the base time.

Benchmark	Base time (sec)	Slowdown			
		CG	CG + SM	Race detection (cache only)	Race detection (cache + depth filtering)
Cholesky	1.66	1.01	1.86	1.88	1.83
FFT	1.58	1.00	2.94	3.08	3.01
Fibonacci	5.54	1.05	1.22	1.31	1.29
QuickSort	1.46	1.02	7.58	7.62	7.73
SmithWaterman	1.59	1.05	8.39	8.89	8.43
UTS	6.29	1.45	3.88	10.22	5.50
RSBench	2.07	1.25	28.10	34.68	34.22
XSBench	2.68	1.20	7.03	83.19	80.86
LCS	1.62	1.03	5.50	6.81	6.71
Geometric mean		1.11	4.95	7.77	7.12

Performance of Bounded Race Detection. We performed another experiment on *RSBench* and *XSBench* (the two benchmarks with the largest overheads) to evaluate the impact of bounded race detection. During the experiment, GT-Race only monitored memory accesses and executed data race detection for a subset of data blocks whose size is smaller than a predefined threshold. For

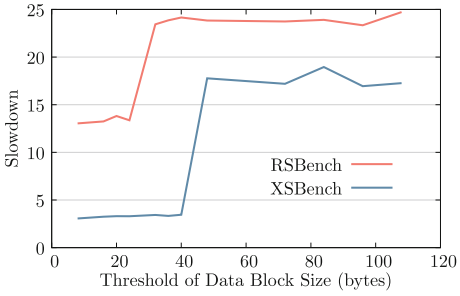


Fig. 4. Bounded race detection result

Table 3. Cache usage

Benchmark	Cache miss	Cache hit	Arith. mean steps
Cholesky	550	224,510	4
FFT	6	262,144	6
Fibonacci	653,329	849,746	4
Quicksort	19,431	283,366	4
SmithWaterman	25,281	1,452,800	5
UTS	549,166	2,629,321	48
RSBench	30,029	14,876,544	1,743
XSBench	73,632	3,128,693	3,204
LCS	20,480	2,329,707	144

each threshold, GT-Race tested 10 runs for each benchmark. We utilize this experiment to roughly evaluate the impact of the number of monitored data blocks on the slowdown of data race detection.

We list the data in Fig. 4. For both *RSBench* and *XSBench*, the slowdown is small when GT-Race carried out data race detection with a low threshold. At a certain point, the slowdown increases significantly then stays constant for a long period. This scenario is because the workload of data race detection is irregular on different data blocks and the slowdown is dominated by a few shared variables that are frequently accessed. These results show that input from the programmer, or perhaps a smart debugger, on which data blocks to monitor can have a significant impact on the overhead of data race detection.

5 Related Work

Since GT-Race is a graph traversal based dynamic data race detector, we relate our work to the state-of-art studies in the following areas.

Dynamic Data Race Detection for Multithreaded Programs: Most dynamic data race detectors are based on vector clock or lockset. FastTrack [6] is the state-of-art vector clock based race detector. It applies a concise representation of vector clock to compress the timestamps of concurrent operations. Although FastTrack reduces the time overhead of vector clock comparison and the space overhead of shadow memory, the size of each vector clock is still proportional to the number of threads. Furthermore, FastTrack can only report data races in the executed thread interleaving.

Eraser [7] is a lockset based lightweight race detector which finds out data races by the locking discipline. It incurs less runtime overhead to the program and can predict data races in other possible interleavings, but it may generate a large amount of false positive. Some work [8] combine lockset with vector clock to achieve both high accuracy and low overhead. They use lockset to replace the expensive vector clock when the program issues lock operations, and report data races when both vector clock and lockset do not prove the correctness of a memory access. These hybrid race detectors can achieve a good trade-off between accuracy and performance.

Because the above-mentioned race detectors are designed for general multithreaded programs, They either cannot handle synchronization constructs in AMT parallelism, or incur unacceptable time and space overhead due to the neglect of structural properties in AMT programs.

Dynamic Data Race Detection for AMT Programs: Some data race detectors are only targeting specific AMT runtimes and utilize the structural properties of the computation graph to verify AMT programs efficiently. SP-bags [9] and ALL-SETS [15] utilize the serial-parallel (SP) structure of Cilk programs to detect data races in amortized bound time and constant space. ESP-bags [16] is an extension to SP-bags that supports *finish* construct in async-finish AMT runtimes. The determinacy race detector in [10] leverages dynamic

task reachability graph to handle async-finish AMT runtimes with futures. However, all these approaches require the program to execute in depth-first order, which wastes the available hardware parallelism in the underlying platform. PTRacer [11] is a parallel on-the-fly data race detector for async-finish AMT runtimes that support locks. It combines SPD3 and ALL-SETS to detect data races with constant memory space. PTRacer also adds a symbolic diagnosis phase after the dynamic analysis to predict hidden races at schedule sensitive branches of the not-taken paths. But PTRacer does not provide any support to point-to-point synchronization constructs.

Reachability Query for DAGs: GT-Race can be abstracted as conducting reachability queries on the computation graph to verify the causal ordering between concurrent memory accesses. Although reachability query has been comprehensively studied over decades, existing work is not suitable for GT-Race. According to the survey presented by [17], state-of-art reachability query algorithms [18–20] compute a label for every node when preprocessing the graph, and return the reachability between any two nodes by comparing assigned labels. These algorithms can answer reachability queries efficiently, but they require an expensive labeling process in advance, which is too time-consuming for large graphs. In addition, the space overhead of each label is proportional, or even square to the number of nodes, which will deplete available memory space quickly. The unacceptable time and space overhead of the labeling process restrict the usage of reachability query algorithms in GT-Race.

6 Conclusion and Future Work

In this paper, we propose GT-Race, a new graph-traversal based data race detector for AMT parallelism. It leverages the computation graph data structure, which encodes the general happens-before structures in AMT programs. GT-Race executes a graph-traversal based data race detection algorithm for each pair of concurrent memory accesses. After one execution, GT-Race can report data races in all possible thread interleavings for the same input. In order to reduce the time and space complexity for race detection, we also apply a few optimizations in GT-Race, such as *epoch adjacency list* to compress the representation of computation graph, *reachability cache* and *depth filtering* to avoid unnecessary explorations, and *bounded race detection* to reduce the range of monitored memory space. Based on our race detection techniques, we have implemented a prototype of GT-Race for OCR. The evaluation on a set of open source OCR benchmarks shows that our tool handles all OCR constructs and incurs acceptable time and space overhead to the program execution.

GT-Race addresses the challenges of data race detection for AMT programs mentioned in Sect. 1 as follows (a) The space complexity of the computation graph is linearly proportional to the number of tasks and dependences, which makes GT-Race scalable to AMT programs (b) GT-Race detects data races by using the happens-before relations among tasks, which incurs no false positives (c) When detecting data races, GT-Race doesn't require the AMT program to

execute in sequential order. GT-Race works in parallel, thereby fully utilizing hardware parallelism for debugging executions as well (d) Since the computation graph is a general representation of happens-before relations, GT-Race can be applied to other AMT runtimes beyond OCR.

For future research, we plan to combine some static analysis techniques with GT-Race to filter out race-free shared variables during dynamic data race detection. We also plan to further improve the efficiency of graph traversal by learning the structural properties in the computation graph more comprehensively.

References

1. Pebay, P., Bennett, J.C., et al.: Towards asynchronous many-task in situ data analysis using legion. In: 2016 IEEE International Parallel and Distributed Processing Symposium Workshops. IEEE, pp. 1033–1037 (2016)
2. Blumofe, R.D., Joerg, C.F., Kuszmaul, B.C., Leiserson, C.E., Randall, K.H., Zhou, Y.: Cilk: an efficient multithreaded runtime system. J. Parallel Distrib. Comput. **37**(1), 55–69 (1996)
3. Chatterjee, S., Tasirlar, S., et al.: Integrating asynchronous task parallelism with MPI. In: 2013 IEEE 27th International Symposium on Parallel & Distributed Processing (IPDPS), pp. 712–725. IEEE (2013)
4. Treichler, S., Bauer, M., Aiken, A.: Realm: an event-based low-level runtime for distributed memory architectures. In: Proceedings of the 23rd International Conference on Parallel Architectures and Compilation, pp. 263–276. ACM (2014)
5. Mattson, T.G., Cledat, R., et al.: The open community runtime: a runtime system for extreme scale computing. In: 2016 IEEE High Performance Extreme Computing Conference (HPEC), pp. 1–7. IEEE (2016)
6. Stenzel, O.: The Physics of Thin Film Optical Spectra. SSSS, vol. 44, pp. 163–180. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-21602-7_8
7. Savage, S., Burrows, M., et al.: Eraser: a dynamic data race detector for multithreaded programs. ACM Trans. Comput. Syst. (TOCS) **15**(4), 391–411 (1997)
8. O'Callahan, R., Choi, J.D.: Hybrid dynamic data race detection. In: ACM Sigplan Notices, vol. 38, pp. 167–178. ACM (2003)
9. Feng, M., Leiserson, C.E.: Efficient detection of determinacy races in cilk programs. Theory Comput. Syst. **32**(3), 301–326 (1999)
10. Surendran, R., Sarkar, V.: Dynamic determinacy race detection for task parallelism with futures. In: Falcone, Y., Sánchez, C. (eds.) RV 2016. LNCS, vol. 10012, pp. 368–385. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-46982-9_23
11. 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. ACM (2016)
12. Sarkar, V.: Comp 322: fundamentals of parallel programming module 1: parallelism (2017). https://wiki.rice.edu/confluence/download/attachments/4435861/module1.pdf?version=5&modificationDate=1519055242728&api=v2
13. Tasirlar, S., Sarkar, V.: Data-driven tasks and their implementation. In: Proceedings of the 2011 International Conference on Parallel Processing, ICPP 2011, pp. 652–661, Washington, DC, USA. IEEE Computer Society (2011)

14. Nethercote, N., Seward, J.: How to shadow every byte of memory used by a program. In: Proceedings of the 3rd International Conference on Virtual Execution Environments, pp. 65–74. ACM (2007)
15. 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 Tenth Annual ACM Symposium on Parallel Algorithms and Architectures, pp. 298–309. ACM (1998)
16. Raman, R., Zhao, J., et al.: Efficient data race detection for async-finish parallelism. Form. Methods Syst. Des. **41**(3), 321–347 (2012)
17. Wei, H., Yu, J.X., Lu, C., Jin, R.: Reachability querying: an independent permutation labeling approach. Proceed. VLDB Endow. **7**(12), 1192–1202 (2014)
18. Wang, H., He, H., Yang, J., Yu, P.S., Yu, J.X.: Dual labeling: answering graph reachability queries in constant time. In: 2006 Proceedings of the 22nd International Conference on Data Engineering, p. 75, ICDE 2006. IEEE (2006)
19. Cheng, J., Huang, S., et al.: TF-label: a topological-folding labeling scheme for reachability querying in a large graph. In: Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data, pp. 193–204. ACM (2013)
20. Trißl, S., Leser, U.: Fast and practical indexing and querying of very large graphs. In: Proceedings of the 2007 ACM SIGMOD International Conference on Management of Data, pp. 845–856. ACM (2007)