Visualizing Correctness Issues in OpenMP Programs

Feiyang Jin, Alan Tao, Lechen Yu, and Vivek Sarkar

Georgia Institute of Technology, Atlanta GA 30332, USA
{fjin35,atao31,lechen.yu,vsarkar}@gatech.edu

Abstract. Past work on OpenMP program visualization has mainly centered on performance analysis. This paper explores how the visualization of *computation graphs* assists programmers in debugging issues related to program correctness. The motivation is twofold. First, researchers widely use computation graphs to analyze dynamic program behavior. Second, most past work focused on visualizing performance bottlenecks rather than correctness issues. This paper's contributions are as follows. First, we introduce techniques for building computation graphs using the OpenMP Tools Interface (OMPT). Second, we present a computation graph visualizer for OpenMP programs built upon these techniques. The visualizer specifically highlights data races as the correctness issue under study. Finally, the paper includes an empirical study of the performance and effectiveness of our prototype. The evaluation demonstrates that our graph builder introduces minimal overhead when integrated with state-of-the-art race detectors. We also conducted a user study involving a control group and an experimental group. The results show that our visualization greatly aids programmers in understanding and debugging data races. Beyond our main contribution, which focuses on data races, we also integrate host-device data movement in the visualization. This serves as a first step toward visualizing data mapping issues. To the best of our knowledge, our work is the first to explore understanding of correctness issues in OpenMP programs through interactive visualization of computation graphs.

Keywords: OpenMP Programs · Computation Graphs · Data Races · Host-Device Data Movement · Dynamic Program Analysis · Visualization

1 Introduction

A *computation graph* is a dynamically constructed directed graph that captures the execution of parallel programs in a partial order. In the research community, computation graphs are extensively used to model the dynamic behavior of parallel programs. Debugging and analysis of parallel programs leverage computation graphs in various ways:

1. Data race detection: a data race happens if two nodes in the computation graph access the same memory locations, at least one performs a write, and

there is no directed path in the computation graph connecting the two. This property is utilized by many dynamic race detectors [9, 14, 17, 18, 22, 23].
2. Deadlock detection: a deadlock arises when a cycle is formed in the computation graph [19]. If the program is deadlock-free, the generated computation graph will be a *directed acyclic graph* (DAG).
3. Determinacy validation: researchers have proven that programs with certain kinds of parallel constructs (e.g., spawn-sync task parallelism and async-finish task parallelism) have the property that data race freedom also guarantees determinacy [6, 9]. One key feature of determinate programs is that given the same input, different executions of the same program will always generate the same computation graph [8, 9].

While computation graphs have gained widespread conceptual usage, there remains a distinct lack of tools designed for OpenMP programmers to visualize them, together with any correctness issues detected. Most existing debugging tools use computation graphs internally. When detecting a correctness issue (e.g., data race, buffer overflow), these tools simply dump the collected trace with an error message. Given the complexity of OpenMP programs, manually analyzing the trace and figuring out the root cause can be challenging, even for experienced programmers (as illustrated by our user study results in Section 6.2). On the contrary, there have been a large amount of prior work dedicated to visualizing the results of performance analysis [1, 3, 4, 12, 15, 21], many of which are for OpenMP programs. These tools usually apply sampling/instrumentation techniques to obtain performance data and embed these data into a customized graph; later a visualization tool will display the performance data in a group of interactive view (e.g., thread view, trace view) to highlight detected performance issues.

Despite the limited efforts in visualizing correctness issues, the need for such a feature remains evident. First, as mentioned above, computation graphs are widely used to identify correctness issues in parallel programs. Introducing visualization offers an effective means to comprehend program behavior and identify the root causes of bugs. Secondly, perhaps less obvious but equally essential, visualization aids developers in recognizing certain tools' limitations. For instance, dynamic race detectors often encounter false positives (reporting non-existing races) due to design choices or implementation errors. Integrating the debug information of detected races with visualization can give a clearer view of a tool's limitations (user study results in Section 6.2 proved this).

In this paper, we take the lead in exploring how visualization of data races can help OpenMP programmers better understand the problems. We have designed and implemented a computation graph visualizer for OpenMP programs. The tool consists of two parts: an on-the-fly graph builder written in C++ and an interactive visualization interface written in Javascript. The graph builder uses *OpenMP Tools Interface* (OMPT) to build computation graphs based on runtime events. The tool provides interfaces for existing race detectors to report race information. We have integrated one race detector [23] into our tool, as an example of how the interface can be used. A second major contribution is a research user study that evaluates the efficacy of our tool. As with any software, the ultimate goal of our tool is to help programmers. Since no similar user study has been conducted to study how visualization helps programmers understand data races, we performed such one. The user study demonstrates that developers significantly benefit from using our tool to analyze data races.

In the remaining section of this paper, we first describe the high-level design of our visualization tool. Subsequently, we propose three *language-centric* approaches to build the computation graph. These approaches ensure that the graph is tightly connected to the OpenMP programming and execution model. A previous work on OpenMP performance analysis shares the same approach [3]. After introducing the graph builder, we present our visualization interface. The visualization is interactive and connected to the source code so users can follow the program's execution trace through graph navigation. In short, the key contributions of this work are:

1. A set of techniques for utilizing OMPT callbacks to build computation graphs and an implementation of the graph builder using our techniques (Section 4). The graph builder integrates an existing race detector to get race information.
2. A web-hosted visualization interface for the computation graph. The visualizer combines the graph, source code, trace, and race information. When interacting with developers, it uses carefully designed animation and interaction to highlight data races (Section 5).
3. An evaluation of the graph builder's performance is included and shows that the overhead compared to stand-alone race detection is bound by a 1.32x slowdown (Section 6.1). A research user study indicates that participants found visualization significantly helped them comprehend data races (Section 6.2).
4. An initial step towards visualizing data mapping issues. We explored how to model and present host-device data movements in the computation graph. This includes a new OMPT callback to handle data movement precisely (Section 7), an unprecedented feature among existing callbacks.

2 Background

Computation Graph: A *node* in the computation graph represents a sequence of code without any parallel constructs; the only exception is that the last statement of the node may generate parallelism by task creation or synchronization. There are three kinds of edges [17]: *continuation*, *fork* and *join*. Continuation edges mark the execution order for nodes within the same task. Fork edges represent new task creation. Join edges represent synchronization.

We say node u *happens before* node v if and only if there is a directed path from u to v in the computation graph. We denote it as $u \rightsquigarrow v$. If $u \not\rightsquigarrow v$ and $v \not\rightsquigarrow u$, we say u, v *may happen in parallel* and denote it as $u \parallel v$.
Data Races: A *data race* occurs if and only if nodes u and v access the same memory location, at least one of them conducts a write, and $u \parallel v$.

A dynamic race detector usually consists of two parts [6, 9, 11, 14, 17, 18, 23]: *shadow memory* and *reachability structure*. Shadow memory records previous reads and writes for each memory location; the reachability structure answers happens-before queries for pairs of nodes. In practice, reachability structures are often designed to save only the necessary parts of a computation graph.
OpenMP: In this work, we focus on the *task parallelism* constructs in OpenMP. In task parallelism, work is divided into tasks, and each available worker thread will be assigned a task to execute. An OpenMP program starts as an *initial task*, and the program executes sequentially until it encounters a *parallel* construct, which will initiate a *parallel region* consisting of parallel tasks. By default, a *barrier* occurs at the end of a parallel region to synchronize all tasks. All tasks created before the barrier must be completed before any task created after the barrier can start execution. The pattern is shown in Fig. 2.

3 Design

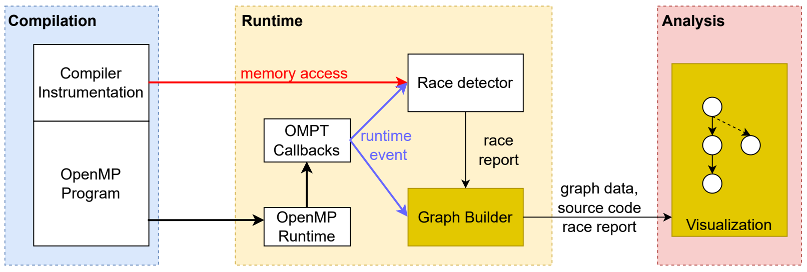


Fig. 1. Application workflow

As shown in Fig. 1, our tool consists of two fundamental components: 1) on-the-fly computation graph construction written in C++, and 2) graph visualization written in JavaScript. Upon receiving an OpenMP program to analyze, the tool executes the program with race detection enabled. During runtime, the computation graph is dynamically built using OMPT. OMPT provides callbacks to capture runtime events with marginal time overhead, and we update the graph correspondingly in each callback. The race detector we integrated is TSAN-SPD3 [23]. It relies on computation graphs to detect data races, though it does not explicitly keep the graph. At runtime, if TSAN-SPD3 detects a data race, it will report the race information to our graph builder; the graph builder will save the information in the graph. After the program is completed, the tool outputs all the information into a JSON file. The visualization component reads this JSON file to present the computation graph integrated with the source code and race report. Visualization is implemented in *D3*, a JavaScript library renowned for creating interactive visualization.

Regarding conceptual exploration, our tool is the first to spotlight correctness issues by visualizing computation graphs. Consequently, our approach lacks precedence for reference. The best experience we can learn from is visualization work for OpenMP program performance analysis [1, 3, 12, 21]. On one hand, we share similarities. We both build a graph during execution and visualize it offline. As they integrate performance information into the graph, we integrate correctness issues detected. On the other hand, our contributions are quite different from theirs. The graphs built have different granularity, and we carefully designed the graph structure to enhance comprehension. Our graph builder also has an interface for existing tools to report information. Instead of asking programmers to find issues themselves, we allow them to utilize existing tools.

Regarding implementation, we have tested several tools mentioned in Section 8 related work and explored existing graphing software such as yEd[1], Cytoscape [16] and Intel Advisor's FGA [1]. Nevertheless, we opted to construct our own graph builder and visualization for several reasons. First, the installation process of other tools posed challenges. For example, many tools in Section 8 require administrative permissions to trace programs, potentially limiting accessibility. In contrast, our graph builder is an OMPT tool compatible with any OpenMP version that supports OMPT. The visualization part is browser-based, ensuring ease of use without installation. Additionally, the dynamic nature of our visualization, facilitated by JavaScript, enables powerful animations that play a pivotal role in highlighting data races - a feature that could be more robustly supported by the alternatives we explored.

4 Graph Construction Techniques

4.1 Language-Centric Approaches

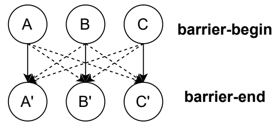


Fig. 2. OpenMP barrier pattern

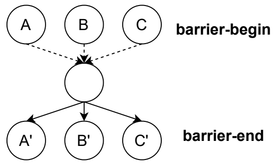


Fig. 3. Barrier visualization in our tool

These approaches aim to enhance programmers' comprehension by connecting computation graphs to OpenMP's programming model.

Approach 1: Designing for Programmers (rather than the specification) While adhering closely to the OpenMP specification is always essential, we have made adjustments in several instances to optimize the graph's presentation for improved comprehension. Here, we will use barriers as an example. In OpenMP, when a barrier occurs, all tasks must hit the barrier before any can proceed – a synchronization pattern depicted in Fig. 2. Previous work has used this presentation to illustrate barrier [13, 23].

However, such visualization would contaminate the computation graphs with overlapping, crowded edges. The situation worsens when the number of threads

[1] www.yworks.com/products/yed

becomes large (we usually test benchmarks with eight threads). To avoid the problems, we opted to redesign the barrier presentation, as shown in Fig. 3. Theoretically, this representation means "all tasks will join a single task's synchronization node before continuation." While a slight deviation from the formal specification, this redesign significantly enhances the graph's clarity and simplifies navigation.

Approach 2: Leveraging Implicit Coherence in the Context. By using the implicit coherent context, we do not need to handle all OpenMP constructs explicitly. Our existing code already handles many "quietly". We will use the `single` construct as an example. In OpenMP, a single construct specifies that only one task will execute the associated code block. An implicit barrier occurs at the end of a single construct unless a `nowait` clause is specified. When a single construct begins and ends, the OMPT `callback_work` event is invoked; the callback tells programmers which task will execute the code block.

We could utilize `callback_work` and correspondingly update the computation graph: add a continuation node for the selected task while leaving others untouched. However, such handling is unnecessary: as only one task will execute the code region, other OMPT events in the region will automatically extend the graph. Similarly, the `nowait` clause is implicitly handled by another OMPT callback, which will only be invoked if a barrier occurs. If a `nowait` clause is present, the callback will not be invoked because no barrier exists. Consequently, we do not need to save additional information when a single construct occurs. Existing OMPT callbacks will correctly record the execution.

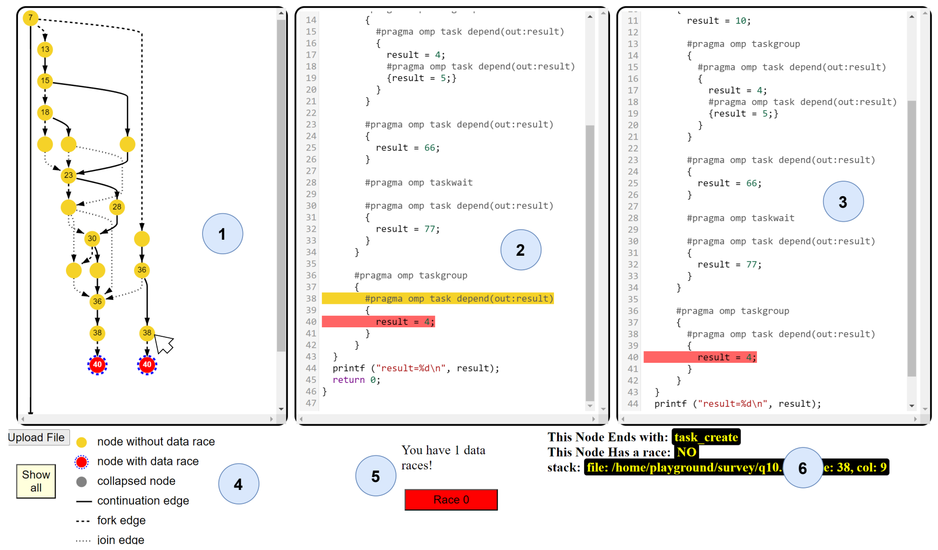
Approach 3: Applying Sufficient and Minimized Serialization The dynamic nature of OpenMP programs requires the analysis tool itself to be aware of the parallelism. Concurrent tasks can invoke the same OMPT callbacks simultaneously during runtime, potentially creating data races within the analysis tool. On the other hand, excessive serialization can impede the tool's performance.

Here, we will explain how we handle parallel regions as an example. Recall that a parallel region creates a group of new tasks. Ideally, the solution to updating the graph is to add a start node for each new task and connect the current node to each new node by a fork edge. In this case, the callback provided by OMPT is `implicit_task`; unfortunately, it will only be invoked after each new task is created. As a result, if we add a fork edge for each new task inside this callback, we introduce data races because many tasks can invoke the callback simultaneously. The resulting graph could miss several fork edges.

To safely update the graph when a group of tasks is created, we use a concurrent vector to save all the newly generated fork edges. Each call to `implicit_task` will add a fork edge to the vector. The edges will be added to the graph just before we output it. On the other hand, additional serialization is not required at the end of a parallel region. When a parallel region starts, we already know the number of new tasks being created (assume the number is n). Thus, we can create a vector v of size n and assign an index i for each new task. When each task finishes, it only needs to put its last node into $v[i]$. When the parallel region ends, one task will iterate through the vector and add join edges to the graph. Notice that this iteration is sequential, so we do not need extra synchronization.

4.2 Implementation

Our graph builder's source code can be accessed here[2]. The graph builder is implemented as a stand-alone OMPT tool. During execution, the computation graph is stored in memory and exported to a JSON file if the program completes without crashing. Many OMPT callbacks provide the program counter as a parameter. When available, we leverage this to extract trace information, such as line numbers and file names. For existing race detectors to report races, we provide functions they can call to pass the information. The functions are implemented in OMPT, so existing tools need to define them as weak functions in their code.



```
        {
          #pragma omp task depend(out:result)
          {
            result = 4;
            #pragma omp task depend(out:result)
            {result = 5;}
          }
        }

        #pragma omp task depend(out:result)
        {
          result = 66;
        }

        #pragma omp taskwait

        #pragma omp task depend(out:result)
        {
          result = 77;
        }
      }

      #pragma omp taskgroup
      {
        #pragma omp task depend(out:result)
        {
          result = 4;
        }
      }
  }
  printf ("result=%d\n", result);
  return 0;
}
```

```
        result = 10;

        #pragma omp taskgroup
        {
          #pragma omp task depend(out:result)
          {
            result = 4;
            #pragma omp task depend(out:result)
            {result = 5;}
          }
        }

        #pragma omp task depend(out:result)
        {
          result = 66;
        }

        #pragma omp taskwait

        #pragma omp task depend(out:result)
        {
          result = 77;
        }
      }

      #pragma omp taskgroup
      {
        #pragma omp task depend(out:result)
        {
          result = 4;
        }
      }
  }
  printf ("result=%d\n", result);
```

You have 1 data races!

This Node Ends with: task_create
This Node Has a race: NO
stack: file: /home/playground/survey/q10. e: 38, col: 9

Fig. 4. User interface for the visualization tool. We have hosted the tool on Amazon's Amplify web service (`https://www.drvis.ninja`). Users can access the website, view preset examples, or upload their graphs to visualize.

5 Visualization

Fig. 4 presents the UI for the visualization. It consists of 6 divisions (Div). Div 1 is the computation graph. Each node in the graph contains:

1. Stack trace information (if available). If the stack is captured, hovering the mouse cursor over the node will highlight the line in the source code. As

[2] `https://github.com/lechenyu/llvm-project/tree/ompt52_dpst`

shown in Fig. 4, line 38 is highlighted in this manner. At the same time, Div 6 will display the trace information, including the OMPT event this node ends with, if it has a race or not, and the text information of the stack. Users can move the mouse along the graph to follow the program's execution trace while examining the highlighted source code.

2. The feature to be collapsed. If an open node is clicked, it will be collapsed by hiding its outgoing edges. A node will be hidden if it is unreachable from the root node, and the process will propagate to its children recursively. This feature allows programmers to center the scope on racy nodes or regions of interest.

Div 2 and 3 contain the source code. The footer section consists of three parts: Div 4 contains the legend and file upload, Div 5 has buttons to select races, and Div 6 displays node information. Clicking on a race button in Div 5 highlights the two racy lines in Div 2 and 3. The computation graph in Div 1 will also be updated to display only up to those two racy nodes; any subsequent nodes are hidden. In Fig. 4, the racy accesses lie on the same line because two tasks concurrently execute the line. In real life, a race can be caused by two different lines or even two lines in different files.

6 Empirical Study

In this section, we study the implementation of our graph builder and visualization interface to answer the following research questions:

1. Performance (Section 6.1): How much slowdown does the graph builder introduce to the original program execution?
2. Efficacy (Section 6.2): To what extent does the visualization help programmers better understand data races compared with traditional log reports?

6.1 Performance Evaluation

The evaluation aims to examine the amount of slowdown the graph builder introduced. To measure this slowdown, we record the execution time from the program's start to completion. The graph dumping and visualization parts are not considered because they are not integral to assessing the graph builder's performance. We also plan to support a more advanced graph aggregation strategy in the future and treat it as an independent research topic.

For a better performance comparison, the benchmarks and inputs we select are the same used in the original TSAN-SPD3 (the integrated race detector in our tool) paper [23]. A total of nine programs from BOTS [5] are included. The experiment was conducted on a single-node AMD server machine, which consists of a 12-core Ryzen9 3900X operating at 3.8GHZ with 128GB memory (RAM). All benchmarks were compiled by Clang/LLVM 15.0.1 running on Ubuntu 18.04.6. The reported execution times are presented for four configurations: Base, Race, Graph, and Full. For each configuration, we report each benchmark's mean execution time of five runs. The coefficient of variation for each configuration, benchmark, and five runs is within 4.5%.

The "Base" configuration measures the execution time of the original programs. The "Graph" configuration enables only the graph builder but not the race detection. The "Race" configuration enables race detection using TSAN-SPD3 but not the graph builder. Finally, the "Full" configuration encompasses race detection and the graph builder.

Table 1. Graph builder performance and graph metrics

Benchmark	**Base**	**Graph**	**Race**	**Full**	**Full/ Race**	**#node**	**#edges**	**#fork**	**#join**
align	1.45	1.59	10.01	10.08	1.01x	9927	14891	4958	4965
fft	1.74	17.21	30.13	39.74	1.32x	77221537	134838680	28808574	57617144
fib	1.56	23.05	58.02	71.11	1.23x	74651782	134373200	29860710	59721419
health	1.67	9.40	18.89	23.16	1.23x	52547254	87578509	17515629	35031256
nqueens	3.80	49.11	77.92	98.12	1.26x	124231833	243862475	59815322	119630643
sort	2.62	3.06	21.13	26.77	1.27x	6138550	11119286	2490373	4980737
sparselu	2.39	2.62	119.28	120.17	1.01x	588669	884345	292533	295677
strassen	1.22	1.24	25.17	27.30	1.08x	294149	568676	137265	274528
uts	0.29	2.48	19.19	20.47	1.07x	12338710	20564510	4112905	8225801

The performance and graph information are shown in Table 1. The first four columns report the average execution time in seconds for the four configurations. The column "Full/Race" is the slowdown of full configuration over race configuration. The last four columns show the number of nodes, edges, fork edges, and join edges for the computation graph generated from each benchmark. Fig. 5 shows the time overheads.

Concerning only the graph builder, the data in Table 1 indicates that its slowdown correlates to the generated graph's size. The three benchmarks exhibiting the highest overhead (fft, fib, and nqueens) correspond precisely to the top three largest computation graphs generated (nodes+edges). Moreover, they also have the largest number of join edges. The increasing size of the graph and join edges bring more serialized operations within our tool, as discussed in Section 4.1. Similarly, the three benchmarks with the lowest overhead (align, strassen, and sparselu) generate the smallest computation graphs and the fewest join edges.

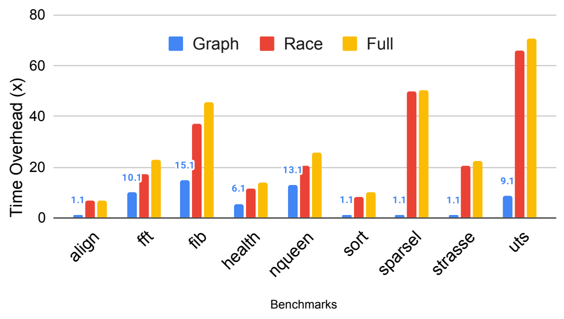


Fig. 5. Time overhead over base configuration

A second crucial finding is that integrating our graph builder with the race detector introduces no more than 1.32x overhead (Full over Race configuration). The logic is coherent because, unlike a race detector, the graph builder does not need to record the access history for each memory location or check for races when memory access occurs. As a result, the overhead imposed by the graph builder is mostly subsumed by the overhead introduced by the race detector.

6.2 Efficacy Study

To evaluate the efficacy of our visualization, we conducted an anonymous user study among graduate-level students, faculty members, and software engineers from different institutions. Participants were randomly divided into a control group and an experimental group. The control group received traditional log race reports from TSAN-SPD3, while the experimental group used race reports presented by our visualization tool. We received a total of 25 responses. After removing poor-quality and irregular data (responses with extreme response times), the experimental group had 8 valid responses, while the control group had 10.

The study consisted of three parts. **Part 1**: Both groups received an introduction to data races along with examples of race reports. In addition, participants of both groups were asked to rate their familiarity, on a scale from 1 to 10, with the following concepts: Parallel Computing or Multithreading, OpenMP, Data Races or Race Conditions and Computation Graphs. The response results are shown in Table 2. The self-reported proficiency ratings indicate that differences in individual skill in parallel computing had little impact on the user efficacy study.

Table 2. Average Self-Declared Proficiency

Group	Part	Average Proficiency by Topic			
		Parallel Computing	OpenMP	Data Races	Computation Graphs
Control	1	6.4	4.8	6.1	4.4
Experimental	1	6.125	5	6.75	4.75

Part 2: For 5 programs (Q1 ~ Q5), participants were asked to identify whether the reported races were true reports or false alarms. **Part 3 (optional)**: For 3 programs (B1 ~ B3), we showed the first line involved in the race; participants needed to identify the second line. We analyzed the responses using two metrics. 1) The percentage of correct responses and 2) The average response time.

Table 3. Accuracy, average time elapsed (seconds), and participation rate

Group	Part	Accuracy and Average Response time					Participation		
		Q1	Q2	Q3	Q4	Q5	Q1 ~ Q5		
Control	2	50% (119s)	38% (115s)	50% (118s)	75% (82s)	50% (64s)	100%		
Experimental	2	50% (53s)	50% (145s)	63% (45s)	88% (26s)	63% (31s)	100%		
		B1	B2	B3			B1	B2	B3
Control	3	17% (60s)	0% (106s)	0% (81s)	N/A	N/A	60%	50%	50%
Experimental	3	50% (117s)	50% (65s)	25% (50s)	N/A	N/A	100%	100%	100%

Table 3 reveals interesting trends. For part 2, a required section, the experimental group demonstrated equivalent or superior accuracy to the control group across all five questions while spending significantly less time overall (except for Q2). For part 3, an optional section, all participants in the experimental group finished all questions, in contrast to the control group's 50% completion rate. Control group members may encounter challenges using traditional log reports, even if the reports contain stack traces and line numbers. These challenges could contribute to the observed lower completion rates and accuracy.

After part 3, both groups rated the difficulty of understanding data races on a scale from 1 to 10. Results indicate that participants in the control group (avg. 8.4) found data races challenging to comprehend. Upon viewing a screenshot of our tool, they strongly agreed (avg. 8) that visualization would facilitate their understanding of data races. Conversely, the experimental group rated the difficulty of understanding data races as fair (avg. 4.6) and strongly agreed (avg. 7.3) that visualization aids comprehension.

These results suggest that our visualization tool offers the advantage of enhanced productivity. It enables programmers to gain a deeper understanding of data races and expedites the process of identifying root causes, thereby reducing time spent. Moreover, the tool helps programmers recognize a race detector's limitations by visualizing false negatives. By utilizing the graph, developers can more confidently identify false results reported. Any tool's ultimate goal is to assist programmers, and the user study indicates that our tool, although not perfect, is clearly on the right track.

7 Extending Beyond Data Races: Visualizing Data Mapping Issues

OpenMP programs suffer from various correctness issues. For example, previous work [24] has shown that many bugs can arise from misunderstandings of data movement behavior. These bugs include data races, uninitialized memory, stale data, and more. We currently support host-device data movement in our visualization. In an ongoing project, we are integrating Arbalest [24] to report bugs caused by incorrect use of data movement. Fig. 6 shows our current visualization. OpenMP programmers can use the *target offloading* feature to migrate code onto heterogeneous devices for execution. Data is moved between the host (CPU) and the target/device (GPU) at the begin/end of an offloading. Users can specify options to control this movement. For example, the *to* option copies data from host to device at the beginning but not the other way around at the end.

```
#pragma omp target map(to:a[0:N]) map(from:c[0:N]) map(alloc:b[0:N*N]) device(0)
```

However, no current OMPT callbacks accurately model host-device data movements. Two possible but inferior solutions using existing callbacks exist. The first callback handles *data mapping*, a high-level abstraction of data movement. Unfortunately, data mapping does not necessarily lead to data movement. This may happen when the same data mapping is declared multiple times in a GPU kernel. Tool developers could handle data movement in this callback, but they must track if a movement is incurred every time, leading to significant time

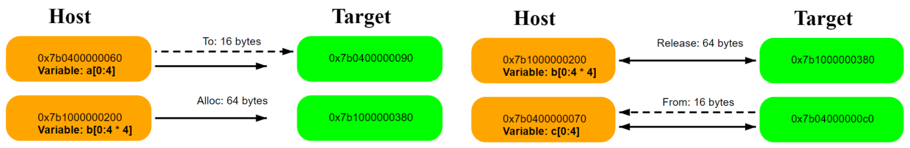


Fig. 6. Data movement code and visualization. The left-hand side shows transferring data from host to target when the offloading begins. The right-hand side shows transferring data back to host when the offloading ends. Variable address, name, and size are included for each transfer.

overhead. The second callback handles *data operations*, which are low-level GPU memory operations such as allocation. These operations are used to implement data mapping. The challenge is that it is up to the runtime to determine how to utilize operations to accomplish a mapping, which means this callback may not have enough information about the movement.

We introduced a new OMPT callback `device_mem` to model host-device data movement precisely. This callback is associated with each data mapping and combines all data operations conducted. Therefore, it avoids problems faced by existing callbacks and provides debugging information for the mapped variable, including name, size, file name, and line number. These details are unavailable with existing callbacks for data mappings and data operations.

8 Related Work

Using visualization to explore performance bottlenecks is an active research field. Muddukrishna et al. proposed grain graphs [12], which considers grain (a task or a parallel for-loop) as the unit for performance measuring. Langdal et al. implemented three OMPT callbacks to build grain graphs [10]. Reissmann and Muddukrishna proposed a strategy to aggregate grain graphs to ease navigation [15]. Agrawal et al. visualized OpenMP task dependencies using OMPT to analyze performance [1]. Aftermath is a performance analysis tool for detecting performance bottlenecks in task parallel programs [4]. A subsequent work extends Aftermath by providing an instrumented version of OpenMP [3]; similar to our work, the authors applied a language-centric approach. AfterOMPT [21] further extends Aftermath by incorporating OMPT events to trace programs.

The work mentioned above was designed to provide insights into performance issues. A handful of projects have also contributed to debugging visualization. DAGViz is a tool that captures and visualizes computation graphs [7]. However, users must write programs using a generic model, which only supports three parallel constructs. Temanejo [2] acts at the task level and gives users the dependency graph while debugging the program in its GUI. ThreadScope visualizes multithreaded applications [20]. The graph generated shows memory operations, and the authors propose that users can identify graph-based problems from it. However, all previous works have merely stopped at "presenting the graph" without displaying any correctness problems, leaving users to debug on their own. In contrast, our work allows existing tools to report issues and incorporates dynamic animation and interaction. Additionally, we adopt a language-centric approach to relate the graph to the OpenMP context. These innovations are crucial in helping programmers understand the problems without getting lost in the debugging process.

9 Conclusion

In this paper, we studied how visualization can help programmers understand data races in OpenMP programs. Our contribution lies in formulating techniques to correctly and efficiently construct computation graphs. A new OMPT callback is introduced by us to model host-device data movements accurately. The ensuing implementation of our graph builder, tailored for OpenMP programs, provides a robust foundation for visualizing correctness-related aspects. The interactive visualization interface allows programmers to review data races and data movements. To our knowledge, our tool is the first to explore and highlight correctness issues in OpenMP programs through a customized visualization interface. It is also a trailblazer for future successors because the graph builder and visualization are independent of any debugging tool; developers can integrate their tool with our work. A key opportunity for future work is to develop an improved aggregation approach to facilitate easier graph navigation. Loop parallelism can also be supported by utilizing OMPT events to collect information and visualizing in the graph.

References

1. Agrawal, V., Voss, M.J., et al.: Visualization of openmp* task dependencies using intel® advisor–flow graph analyzer. In: 14th International Workshop on OpenMP. pp. 175–188. Springer (2018)
2. Brinkmann, S., Gracia, J., Niethammer, C.: Task debugging with temanejo. In: Tools for High Performance Computing 2012. pp. 13–21. Springer (2013)
3. Drebes, A., Bréjon, J.B., et al.: Language-centric performance analysis of openmp programs with aftermath. In: 12th International Workshop on OpenMP. pp. 237–250. Springer (2016)
4. Drebes, A., Pop, A., et al.: Interactive visualization of cross-layer performance anomalies in dynamic task-parallel applications and systems. In: 2016 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS). pp. 274–283. IEEE (2016)
5. Duran, A., Teruel, X., et al.: Barcelona openmp tasks suite: A set of benchmarks targeting the exploitation of task parallelism in openmp. In: 2009 international conference on parallel processing. pp. 124–131. IEEE (2009)
6. Feng, M., Leiserson, C.E.: Efficient detection of determinacy races in cilk programs. In: Proceedings of the ninth annual ACM symposium on Parallel algorithms and architectures. pp. 1–11 (1997)
7. Huynh, A., Thain, D., et al.: Dagviz: A dag visualization tool for analyzing task-parallel program traces. In: Proceedings of the 2nd Workshop on Visual Performance Analysis. pp. 1–8 (2015)

8. Jin, F., Jacobson, J., et al.: Minikokkos: A calculus of portable parallelism. In: 2022 IEEE/ACM Sixth International Workshop on Software Correctness for HPC Applications (Correctness). pp. 37–44. IEEE (2022)
9. Jin, F., Yu, L., Cogumbreiro, T., et al.: Dynamic determinacy race detection for task-parallel programs with promises. In: 37th European Conference on Object-Oriented Programming (ECOOP 2023). Schloss-Dagstuhl-Leibniz Zentrum für Informatik (2023)
10. Langdal, P.V., Jahre, M., Muddukrishna, A.: Extending ompt to support grain graphs. In: 13th International Workshop on OpenMP. pp. 141–155. Springer (2017)
11. 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 (1991)
12. Muddukrishna, A., Jonsson, P.A., et al.: Grain graphs: Openmp performance analysis made easy. In: Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming. pp. 1–13 (2016)
13. Protze, J., Hahnfeld, J., et al.: Openmp tools interface: Synchronization information for data race detection. In: 13th International Workshop on OpenMP. pp. 249–265. Springer (2017)
14. Raman, R., Zhao, J., et al.: Scalable and precise dynamic datarace detection for structured parallelism. Acm Sigplan Notices **47**(6), 531–542 (2012)
15. Reissmann, N., Muddukrishna, A.: Diagnosing highly-parallel openmp programs with aggregated grain graphs. In: Euro-Par 2018: Parallel Processing: 24th International Conference on Parallel and Distributed Computing, Turin, Italy, August 27-31, 2018, Proceedings 24. pp. 106–119. Springer (2018)
16. Shannon, P., Markiel, A., et al.: Cytoscape: a software environment for integrated models of biomolecular interaction networks. Genome research **13**(11), 2498–2504 (2003)
17. Surendran, R., Sarkar, V.: Dynamic determinacy race detection for task parallelism with futures. In: International Conference on Runtime Verification. pp. 368–385. Springer (2016)
18. Utterback, R., Agrawal, K., et al.: Efficient race detection with futures. In: Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming. pp. 340–354 (2019)
19. Voss, C., Cogumbreiro, T., Sarkar, V.: Transitive joins: a sound and efficient online deadlock-avoidance policy. In: Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming. pp. 378–390 (2019)
20. Wheeler, K.B., Thain, D.: Visualizing massively multithreaded applications with threadscope. Concurrency and computation: Practice and experience **22**(1), 45–67 (2010)
21. Wodiany, I., Drebes, A., et al.: Afterompt: An ompt-based tool for fine-grained tracing of tasks and loops. In: 16th International Workshop on OpenMP. pp. 165–180. Springer (2020)
22. Xu, Y., Singer, K., Lee, I.T.A.: Parallel determinacy race detection for futures. In: Proceedings of the 25th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming. pp. 217–231 (2020)
23. Yu, L., Jin, F., et al.: Leveraging the dynamic program structure tree to detect data races in openmp programs. In: 2022 IEEE/ACM Sixth International Workshop on Software Correctness for HPC Applications (Correctness). pp. 54–62. IEEE (2022)
24. Yu, L., Protze, J., et al.: Arbalest: dynamic detection of data mapping issues in heterogeneous openmp applications. In: 2021 IEEE International Parallel and Distributed Processing Symposium (IPDPS). pp. 464–474. IEEE (2021)