Enhancing DataRaceBench for Evaluating Data Race Detection Tools

Gaurav Verma[1,3] gaurav.verma@stonybrook.edu
Yaying Shi[1,4] yshi10@uncc.edu
Chunhua Liao[1] liao6@llnl.gov
Barbara Chapman[2,3] barbara.chapman@stonybrook.edu
Yonghong Yan[4] yyan7@uncc.edu

[1]Lawrence Livermore National Laboratory, Livermore, California, USA
[2]Brookhaven National Laboratory, Upton, New York, USA
[3]Stony Brook University, Stony Brook, New York, USA
[4]University of North Carolina at Charlotte, Charlotte, North Carolina, USA

***Abstract*—DataRaceBench is a dedicated benchmark suite to evaluate tools aimed to find data race bugs in OpenMP programs. Since its initial release in 2017, DataRaceBench has been widely used by tool developers to find the strengths and limitations of their tools. The results also provide an apple-to-apple comparison of the state-of-the-art of data race detection tools. In this paper, we discuss our latest efforts to enhance DataRaceBench. In particular, we have added support for Fortran language and some of the newest OpenMP 5.0 language features. We have also added new kernels representing new patterns from literature and other benchmarks (e.g., NAS Parallel Benchmark). To reduce duplicated code patterns in the benchmark suite, we have designed a distance-based code similarity analysis, combining both static and dynamic code features. Finally, we dockerize tools and streamline the entire benchmarking process to quickly generate a dashboard showing the state-of-the-art of data race detection of OpenMP programs. The enhanced DataRaceBench is released as v 1.3.0, with 222 newly added benchmarks. 56 of them are in C, and the remaining 166 are in Fortran, reproducing the C programs' nature. Our experiments show that this new version can spot more limitations of the current data race detection tools, with significantly reduced user efforts needed to run experiments.**

***Index Terms*—Benchmarks, OpenMP, Data Races, Tools,**

I. Introduction

DataRaceBench (DRB) is a dedicated benchmark suite to evaluate tools aimed to find data race bugs in OpenMP programs. It is an open-source benchmark suite designed to systematically and quantitatively assess the effectiveness of data race detection tools. DataRaceBench includes a set of microbenchmark programs with or without data races, written in OpenMP, the popular programming model for multi-threaded applications. Since its initial release in 2017, DataRaceBench has been widely used by tool developers to find strengths and limitations of their tools and to facilitate tool development [30, 3, 27, 6, 12, 11, 18]. We also publish the latest results on GitHub to provide an apple-to-apple comparison of the state-of-the-art of data race detection tools.

However, the latest version of DataRaceBench v 1.2 still lacks many desirable features. It supports only C/C++ and OpenMP 4.5. It is still labor-intensive to use the benchmark suite to evaluate different tools. Last but not least, it is challenging to add new benchmark programs without introducing duplicated test cases.

In this paper, we discuss our efforts to enhance DataRaceBench to address its limitations mentioned above. In particular, this paper has the following new contributions:

- We have added equivalent Fortran versions for existing C/C++ benchmark programs.
- We have added additional benchmark programs to cover some of the latest OpenMP 5.0 language features.
- We have collected new kernels representing new patterns from literature and other benchmarks (e.g., NAS Parallel Benchmark [4]).
- To reduce duplicated benchmark programs, we explored a simple distance-based code similarity analysis, combining both static and dynamic features of OpenMP loops.
- The test process has been improved to allow users to select customized subsets of benchmarks to evaluate their tools. We have also dockerized the tools to streamline the entire benchmarking process to generate a dashboard presenting the state-of-the-art.
- Using the enhanced DataRaceBench, we have re-evaluated several available data race detection tools and generated more comprehensive results showing their strengths and limitations.

A new release of v 1.3.0 of DataRaceBench contains 338 benchmarks, compared to 116 in v 1.2.0, including newly added 56 C and 166 Fortran kernels. These 338 benchmarks include 42 GPU-based kernels, 4 kernels extracted from the NAS Parallel Benchmark, and 46 kernels representing OpenMP 5.0 features. Our experiments using the enhanced DataRaceBench shows that it can spot more limitations of the current data race detection tools, with significantly reduced user efforts needed to run experiments.

II. Enhancements to DRB

DataRaceBench was created as a common and comprehensive benchmark suite to provide an apple-to-apple comparison of data race detection tools for OpenMP. By design, DataRaceBench includes both positive and negative tests. Positive tests are OpenMP programs with known data races, while negative tests are programs which are data race free. With both positive and negative tests, one can easily check if a given data race detection tool can generate an expected positive

978-0-7381-1044-8/20/$31.00 ©2020 IEEE
DOI 10.1109/Correctness51934.2020.00008

or negative report for each test. For example, if a tool correctly identifies and reports the known data races in a positive test, we record a true positive for the tool. Similarly, we can record true negative, false positive, and false negative for every test result of a given tool. With these positive and negative counts, standardized quality metrics, such as precision, recall, and accuracy, can be calculated for each tool being evaluated. More details about how we calculate these metrics are given in Section. III-B.

This section discusses different enrichments to the DataRaceBench. We also give details about the attempt to reduce redundant benchmark programs and to simplify the testing workflow using dockers.

A. Inclusion of Fortran 95 support

In the field of HPC, there are three prominent languages - C, C++, and modern Fortran (Fortran 90/95/03/08). The popular OpenMP and MPI libraries for parallelizing code are developed for these languages only. Further, many legacy codes in physics are written in Fortran. The ease of use and huge codebase in Fortran bolsters a need for a benchmarking framework to evaluate various tools for their correctness in identifying data races in OpenMP kernels.

To implement Fortran support to the existing DataRaceBench Framework, we have replicated the current C/C++ versions of the microbenchs from the DataRaceBench. Additionally, we have added features like WORKSHARE, which are only supported in Fortran. DataRaceBench contains three kernels from the PolyBench suite written in C. Since converting the complete program into Fortran is a tedious task, we have converted those C programs' nature into similar smaller Fortran kernels. In this way, we have included all the possible kernels present for the C/C++. Entirely, we have added 166 Fortran programs.

We tried to find tools that can automatically convert OpenMP C programs to equivalent OpenMP Fortran programs, but we could not find any such tools. Therefore we used manual translation instead. While writing Fortran kernels, we encountered several hurdles as follows.

1) Initially, for an N-Dimensional array, Fortran's index begins with 1, whereas in C, it starts at 0. To address this, Fortran presents a medium to begin indexing with 0 using `INTEGER ARR(0:2)`. In a few straightforward scenarios, we have incremented the values by 1 to get the lower and upper bounds for an array.

2) Unlike in C, where most of the default passing method is pass-by-value, it is by default pass-by-reference in modern Fortran. This default passing method can be switched using the `%VAL` built-in function or `VALUE` attribute and the `%REF` built-in function in the argument list of a `CALL` statement or function reference [31].

3) Due to the lack of `static` keyword in Fortran, we have represented static objects using `save` attributes to preserve the value of a variable local to a subroutine across subroutine calls. For example, `integer, save :: var`.

4) Additionally, we can not assign an `Integer` to a `Character` array in Fortran, like in C (line 6 in Listing 1). To accomplish this, we had to use `write()` to cast `Integer` first into a `String` (line 8 in Listing 2) and then assign it to a `Character` array (line 9 in Listing 2). This explicit conversion may lead to data race in `DRB047-doallchar-orig-no.f95`. To overcome this differing nature between the C version and the Fortran version, we privatized the `str` variable in the Fortran version.

Listing 1: C: DRB047 - One dimension array computation with finer granularity

```
1 char a[100];
2 int main() {
3   int i;
4   #pragma omp parallel for
5   for (i=0;i<100;i++)
6     a[i]=a[i]+1;
7   return 0;
8 }
```

Listing 2: Fortran: DRB047 - One dimension array computation with finer granularity

```
1  character(len=100), dimension(:), allocatable :: a
2  character(50) :: str
3  integer :: i
4  allocate (a(100))
5
6  !$omp parallel do private(str)
7  do i = 1, 100
8    write( str, '(i10)' )  i
9    a(i) = str
10 end do
11 !$omp end parallel do
```

5) The added paramount challenge we faced was array-ordering in C and Fortran. While C arrays are predominantly stored in row-major order, Fortran arrays are stored in column-major order. We have made the required adjustments to support these features. For a 2-D array, we have warranted correct indexing by converting the row-indexing to column-indexing and vice versa.

Listing 3: C: DRB014 - Data race caused due to out-of-bound access

```
1 double b[100][100];
2 #pragma omp parallel for private(j)
3   for (i=1;i<n;i++)
4     for (j=0;j<m;j++)
5       b[i][j]=b[i][j-1];
```

Listing 4: Fortran: DRB014 - Data race caused due to out-of-bound access

```
1 allocate (b(100,100))
2 !$omp parallel do private(i)
3   do j = 2, n
4     do i = 1, m
5       b(i,j) = b(i-1,j)
6     end do
7   end do
8 !$omp end parallel do
9 deallocate(b)
```

The above listings present an example, where we took care of the column-major indexing in Fortran while translating a

C kernel. Listing 3 is a C version of DRB014, where the outermost loop is parallelized (line 4). But the inner level loop has out of bound access for `b[i][j]` when `i=2` and `j=0`. The `b[i][j-1]` on line 5 will cause memory access to a previous row's last element, `b[1][3]`, and hence the data race. To elucidate the same behavior in Fortran, presented in Listing 4, we took care of the column-major ordering in Fortran. We ensured that there are out-of-bound access and loop-carried dependency on line 5, causing the data race when `i=1` and `j=3`. It will access `b[i-1][j]`, resulting in `b[0][3]`, which in turn is `b[4][2]` due to linearized column-major storage of the 2-D array.

6) Lastly, pointers in Fortran are intricate. For example, a procedure pointer must not be referenced unless it is a pointer associated with a target procedure. An entity with the `POINTER` attribute must not have the `ALLOCATABLE`, `INTRINSIC`, or `TARGET` attribute, and it must not be a `coarray`. We have employed `ASSOCIATED` intrinsic function to find the association status of a pointer. Ultimately, while writing kernels in Fortran, we have accounted for the eccentricities, as mentioned above.

B. Inclusion of OpenMP 5.0 and GPU specific kernels

Many novel features have been added to the OpenMP 5.0 while DataRaceBench v 1.2 only supports some of OpenMP 4.5. Below are a few listed OpenMP 5.0 features [17] which may lead to data races caused by human errors:

- Extended **`teams`** construct for host execution
- **`taskwait`** with dependences
- **`mutexinoutset`** task dependences
- Multidependence Iterators (in **`depend`** clauses)

Additionally, we have included various possible GPU offloading kernels. It is to incorporate data races on hardware accelerators. We have also introduced kernels having data race from the DataRaceOnAccelerator [27] benchmark suite. The extension will outfit DataRaceBench to recognize data races and notify when a kernel is executed on GPU in contrast to a CPU. There is considerable work going on in the form of ARCHER [2], PRUNER [26], ARCHERGEAR [30], SWORD [3] to identify the data races in parallel programs. We have taken care of extracting the example OpenMP programs mentioned by these papers too, which were not covered earlier in our benchmark suite.

To maintain the completeness of the DataRaceBench, we have included all new kernels in both C/C++ and Fortran 95. It will enhance the framework's capability and extend the outreach, and DRB can be applied across data race detection tools to evaluate them more exhaustively. Also, data race kernels on hardware accelerators will check for unsynchronized behavior across host and device. We have ensured that there are positive and negative scenarios for a kernel. Continuing the existing Property Labels presented in the DataRaceBench [20], we have included them for all the newly added kernels in C/C++ and Fortran95. Those kernels which are using accelerator directives, we have placed them under label Y5, `Accelerator data races`, and N5, `Using accelerator directives`.

Listing 5: DRB144 - Data race caused by missing synchronization across `teams`

```
1 #pragma omp target map(tofrom:var) device(0)
2 #pragma omp teams distribute parallel for
3 for (int i=0; i<100; i++){
4   #pragma omp critical
5   var++;
6 }
```

We give an example of the newly added tests in Listing 5. In this example, the map-type at line 1 is `tofrom`. The value of the `var` is always copied from the device environment to the host. The `team` construct will generate a league of teams in which the first thread will be liable for the associated region's execution. The `distribute parallel` at line 2 will enable parallel execution by multiple threads that are members of multiple teams. Since, `critical` construct synchronizes only within a team, there will be a data race at line 5 in `var` in a Non-uniform memory access (NUMA) architecture. To circumvent this, one should use `reduction` clause to have a per team instance of `var`.

The microbenchs must be accurate in themselves. There should not be any accidental or human errors. The exactness is maintained by executing the corresponding kernel on a range of compilers with warning options turned on. We ran them through correctness tools, like Valgrind, to assure no memory leaks, to confirm the correctness.

C. New kernels representing new patterns from benchmarks

Some well-known OpenMP benchmarks, such as the NAS Parallel Benchmark [23], are data race free code. However, when we investigated those benchmarks via dynamic data race detect tools (e.g., Intel Inspector [16], ROMP [12], Archer [2], ThreadSanitizer [28], and so on), a few tools reported data races in those benchmarks. For example, Inter Inspector reported data race in the function `lhsx()`, `lhsy()` and `lhsz()` in the SP program. ThreadSanitizer missed to recognize the `omp barrier` directive and Inter Inspector could not deal with the `omp master` directive in a convoluted code employing conditional compilation using the OpenMP macro, `_OPENMP`.

Many of these code patterns were not covered in the original release of DataRaceBench. We decided to extract them and add them to our benchmark suite. We ensured that the original and our extracted kernel would cause the same data race detection result for four tools.

Listing 6 is a simplified version of the code extracted from the NPB SP program. When we ran NPB through data race detection tools, Intel Inspector and ThreadSanitizer reported data race in function lhsx(), lhsy(), and lhsz(). Archer and ROMP reported no data race, which is expected. After extracting this kernel and reducing it to a simplified version by removing non-contributing lines to the data race, we reran the tools. As expected, Intel Inspector and ThreadSanitizer reported the False Positive at line number 7 and 11. On

manually investigating, we found that there is no data race since there is an explicit barrier between critical and single sections.

Listing 6: DRB172 - Simplified version of a loop pattern from NPB's SP program

```
#pragma omp parallel default(shared)
{
  #pragma omp for
  for (i = 0; i < 10; i++) q[i] += qq[i];

  #pragma omp critical
  q[9] += 1.0;

  #pragma omp barrier
  #pragma omp single
  q[9] = q[9] - 1.0;
}
```

D. Distance-based similarity analysis

With the addition of more test programs to DRB, there is a plausibility that several kernels in various benchmarks have similar patterns causing the data race. To avoid duplicated benchmark programs, we explored a simple distance-based similarity analysis. The distance-based similarity analysis relies on feature vectors representing loops. A feature vector contains static information (such as OpenMP directives and clauses), dynamic information generated by data race detection tools (e.g. Archer, ROMP, ThreadSanitizer, and Intel Inspector), and ground truth. The feature vector is very flexible to include an arbitrary number of fields. Currently, the feature vector is defined using the following formula:

$$\vec{A} = (Directive, Clause, Archer\ result, Intel\ result, Romp\ result, Tsan\ result, Ground\ Truth) \quad (1)$$

Table I presents the detailed information about each feature in a vector. To encode all the OpenMP directives (in total, 87 directives) one obvious choice is to use a single field with a value range of [0, 86], stored in either a binary or integer field. However, this single field encoding for all directives is problematic. Firstly, the distance between two different directives (e.g., pairs encoded as 1 and 85 vs. 2 and 10) would be very different, while we want the distance to be the same for any pair of different directives. Secondly, single field encoding can not handle multiple occurrences of different directives.

To overcome these problems, we encoded each directive (and clause) into one dedicated field. Each field has either 0 or 1 as value. If a loop has any directive, the respective field in the feature vector will have an integer value 1. Else, it will be 0 by default. The same rule applies to the clause's field.

For tools result features, we encode four possible outcomes in one field with a value range of [-2, 1]. We do want to give more weight to segmentation fault and time out errors since they are an unusual situation. A loop causing such errors will have a longer distance compared to ground truth when we calculate cosine distance.

To generate the feature vectors, we have used OpenMP Extractor [22] to get the loop information for each kernel.

Feature	Value Range	Encoding Fields	Description
Directive	[0,1]	[E0-E86]	87 directives are flattened into the first 87 elements in the vector. The existing directive's value is set to 1, else 0.
Clause	[0,1]	[E87-E122]	The next 36 integer elements are 36 flattened clauses. If the value is 1, the test case contains that clause.
Archer	[-2,1]	E123	One integer element for data race result by Archer. -2 represents time out. -1 represents the segmentation fault. 0 represents no data race. 1 represents the data race.
Intel	[-2,1]	E124	Data race result by Intel Inspector. Same as Archer.
Romp	[-2,1]	E125	Data race result by Romp. Same as Archer.
Tsan	[-2,1]	E126	Data race result by Threadsanitizer. Same as Archer.
Ground Truth	[0,1]	E127	Ground Truth, whether a loop has a data race or not.

TABLE I: Feature vector's definition and encoding methods

The OpenMP Extractor is a tool built using Clang. It stores the information about OpenMP loops in a JSON format. We read the JSON file and convert contained pragma and clause information into the fields of our feature vectors. We also add the data race result from four tools as additional features of our feature vectors. If the tool detects data races, the value will be 1. Otherwise, it will be 0, signifying false by default. When a tool reports a segmentation fault on compile time (CSF) or during runtime (RSF), the value should be -1. If a tool reports runtime time out (RTO), the value should be -2.

For example, `DRB162-nolocksimd-orig-gpu-no.c` has one OpenMP loop. That loop has `target teams distribute parallel for` directive and `map` clause information. The OpenMP Extractor will extract the following information in the following Json format.

Listing 7: OpenMP Extractor information

```
"parallel loop 1":{
"file":"micro-benchmarks/DRB162-nolocksimd-orig-gpu-
    no.c",
"function":"main",
"loop id":"1",
"pragma type":"target teams distribute parallel for
    ",
"offload":"true",
"multiversioned":"false",
"map tofrom":["var"],
}
```

Referring to Listing 7, DRB162 has one directive and one clause. In the feature vector, the 68th element should be set to 1 for the `target teams distribute parallel for` directive, and the 114th element set to 1 for the `map` clause. For the data race detection result, four tools reported results of CSF, TN, TN, and FP, respectively. The corresponding values should be -1, 0, 0, and 1. Since DRB162 has no data race, the ground truth should be set to 0. Combining all the information, the feature vector for

DRB162-nolocksimd-orig-gpu-no.c will be as follows:

$$(0, ..., 0, 1, 0, ..., 0, 1, 0, ..., 0, -1, 0, 0, 1, 0) \quad (2)$$

There are many ways to calculate vector distances. For simplicity, we select Cosine Distance, which is a way to calculate the distance between two non-zero vectors of an inner-product space. It shows the similarity of two vectors by cosine value, theta (θ). The range of Cosine Distance is from [-1,1]. Since we used the integer value for vector, the dot product value could be negative of two vectors. If two vectors are more similar, the cosine distance will be closer to 1, and the degree for two vectors will be closer to 0°. Otherwise, it will be closer to -1, and the degree will be closer 180°. It is derived from the Euclidean dot product formula and is defined as follows:

$$\cos(\theta) = \frac{\vec{A} \cdot \vec{B}}{|\vec{A}||\vec{B}|} = \frac{\sum_{i=1}^{n} A_i B_i}{\sqrt{\sum_{i=1}^{n} A_i^2} \sqrt{\sum_{i=1}^{n} B_i^2}} \quad (3)$$

We save existing benchmark programs' feature vectors as a reference. Before adding any new kernel, similarly, we generate the feature vector for the new kernel and correlate against the previously saved ones by analyzing the distance of two vectors to determine the similarity.

The distances among existing loops, in the current benchmark suite, can also be calculated. A n⋆n distance matrix (n is the total number of loops in DataRaceBench) can be generated. We convert the matrix into a heat map to show the distance among these loops. If two loops are similar, the color is a darker shade of blue. Otherwise, the color is redder. This helps us identify similar loops within the current DataRaceBench.

E. Improving the Workflow for DRB

Users have reported that it is difficult to install data race tools. It is also difficult to use the tools properly to process the files of DataRaceBench to generate final results. They also want the flexibility to test an arbitrary set of benchmarks, instead of always testing all of them. We have enhanced our workflow to address these user requirements.

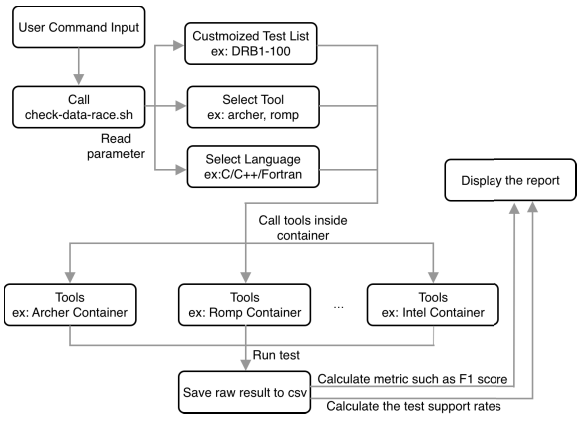

Fig. 1: The workflow of DataRaceBench (DRB)

Figure 1 presents the high-level enhanced workflow. A user only needs to submit a single command with a parameter to call the check-data-race.sh script. The script will automatically read the parameters given by the user. The parameters are used to select a set of test cases, languages, and tools. We added new features that allow the user to perform partial tests of DRB microbenchmarks. For example, if a user only wants to test the unique features of OpenMP5.0 with a specific tool, he/she only needs to add required OpenMP5.0 test files into a configuration file and define the tools that the user wants. As a result, users have full control over which set of tests and tools to be used.

Another script, test-hardness.sh, will automatically run the specified tests with the selected tools. It also saves generated binary files, output, and log files into a result folder. After running the data race detection tools on the test programs, the raw results are saved as a CSV file. At last, the script will automatically calculate the statistical metrics and remove all trash files.

Each data race detection tool has different requirements for the supportive compilers, library dependencies, and command-line options, and so on. To relieve the users from installing and configuring each tool, instead, we have dockerized the DRB with each tool's image in the docker container. By using the Docker container, users can use, modify, and develop DRB rapidly. Without bothering about installing tools, they can download and evaluate the tools anywhere on any machine with the DRB docker container.

We also updated the Tool Evaluation Dashboard of DRB to show the state-of-the-art. The new DRB reports the tool's failure rates and the statistic analysis metrics in the final report. Users can evaluate the test file support rate of the tool intuitively. The enhanced DRB can automatically calculate the False Positive, True Positive, False Negatives, and True Negative. It also reports the five standard metrics: Recall, Specificity, Precision, Accuracy, and F1 score to evaluate the performance of tools. The metrics are calculated based on four possible results of a tool, as shown in Table II.

III. Experiments

This section details the software bundled in the docker images, the tools we selected, the hardware we use for our experiments using DataRaceBench v 1.3.0.

A. Software and Hardware Configurations

We ran DataRaceBench on Lassen [15] cluster hosted at Livermore Computing Center. The machine has 2 Sockets, 16 Core(s) per socket, 4 thread(s) per core, 128 CPU(s), and NVIDIA Tesla V100 GPU. There are 4 GPUs and 2 IBM POWER9 processors (dual-socket) with hyperthreading support per compute node. To obtain results of Intel Inspector, we also ran the DataRanceBench on the Carina machine at the University in North Carolina. This machine has 2 Intel(R) Xeon(R) Gold 6230N CPUs and an NVIDIA Tesla V100 GPU.

We evaluated DataRaceBench on four data race detection tools: Archer, Intel Inspector, Romp, and ThreadSanitizer.

Tool Result	Ground Truth		Recall	Specificity	Precision	Accuracy	F1 Score
	True	False					
True	TP	FP	TP / (TP + FN)	TN / (TN + FP)	TP / (TP + FP)	(TP+TN) / (TP + FP + TN+ FN)	2 * (P * R) / (P + R)
False	FN	TN					

TABLE II: Definition of metrics (Recall, Specificity, Precision, Accuracy and F1 Score)

Table III shows the information about the versions, compilers, and flags used specifically for each tool. For C/C++ test cases, we used Clang/LLVM(6.0) as a compiler for Archer. For Intel Inspector, we used icc compiler with max source option using `-collect ti3 -knob scope=extreme -knob stack-depth=16 -knob use-maximum-resources=true` flag. The ROMP used the GCC compiler and Threadsanitizer used `Clang/LLVM(10.0)` with the flags mentioned in the table III.

To evaluate these four tools using DRB on Fortran test cases was a bit convoluted. Since Romp is race detection tool based on binary instrumentation, we used `gfortran` compiler with flag `-fopenmp -lomp -ffree-line-length-none`. In some cases, like `DRB043-adi-parallel-no.F95`, program uses C header and library files. We use the GCC compiler with a `-c` option to generate the object file and link it with gfortran. For Intel Inspector, we used `ifort` to compiler the Fortran test cases with flag `-qopenmp -free -Tf`. We used `icc` to compile it with `-c` flag to generate object file and cross-link with `ifort`. Threadsanitizer uses Clang as compiler, so we used gfortran with `-ffree-line-length-none -fopenmp -c -fsanitize=thread` flag to generate object files. And then used clang with `-lgfortran` flag to link all object file together. In Archer, we used GCC to generate an object file and link it with Clang. Archer is similar to Threadsanitizer. We used gfortran to the compiler with the `-c` option to generate an object file. And then, use the clang-archer link and compiler with `-larcher` flag.

We have executed each test program in DRB using eight threads for the five iterations. From the past research work, it was found that the tools perform reasonably well with eight threads. The result does not change significantly with more number of threads. Five iterations ensure that there is no bias introduced by a tool. Ultimately, we apply a union policy to decide on the presence of data race. The union policy refers to a scenario where a tool reports data race presence at least once in five iterations. In that case, the tool is deemed to report a data race. In this paper, we didn't manually check the four tool's output line by line. We only compare the four tool's output with ground truth of each test file in DRB while performing the statistical analysis. The total time for running the DRB varies from half hour(Archer) to four hours(Romp).

B. Metrics and Results

We have summarized the result for each test program and published the detailed results for all the programs and tools on our Tool Evaluation Dashboard [1]. It contains the tools' execution result for C/C++ and Fortran test cases. The True Positive (TP) are ones where tools reported data race, and there is a data race(s) in the program. False Positive (FP) represents cases where tools reported data race(s), and there is no data race(s) in the program. True Negative (TN) are scenarios where a tool did not report any data race, and there was no data race in the program. Finally, False Negative (FN) represents scenarios where a program has a data race(s), but a tool did not report it.

To address the cases where a particular tool could not compile (due to unsupported feature(s) specific to a version) or encountered runtime error, we have the following tags. We use CSF for Compile-time Segmentation Fault, CTO for Compilation Time-Out, RSF for Runtime Segmentation Fault, and RTO for Runtime Time-Out.

Table IV shows the statistical metrics. It contains the TN/FN and TP/FP results for the four tools in addition to the metrics such as recall, precision, specificity, and so on. We also added a new metric test support rate(TSR). The TSR is the ratio of how many test files are successfully compiled and executed by a tool. The adjusted F-1 score is the F-1 score multiplied by the TSR. The adjusted F-1 score can show the true ability of a tool.

From Table IV, we can see that, for C/C++ test programs, Intel Inspector has the highest TP, lowest FN, and best recall. Archer and ThreadSanitizer have the lowest FP. Also, ThreadSanitizer has the highest TN and best specificity, precision, accuracy, and Adjusted F1. ROMP has the best test support rate. For Fortran test programs, Intel Inspector and ThreadSanitizer have the highest TN. Also, Intel Inspector has the highest TP and adjusted F1. Along with ROMP, it shares the best test support rate(TSR) score. Archer has the lowest FP, and best recall, specificity, precision, and accuracy. ThreadSanitizer has the lowest FP and FN. It also has the best specificity, and precision. However, Archer and ThreadSanitizer's Test Support Rates are low. Compared with previous results, Archer and Romp have more CSF test cases. Overall, with more test cases, the accuracy and F-1 score reduced since tools can not correctly detect the data race(s) in new test cases. Thus, FP and FN increased, and performance has declined. Besides, we used union policy for five-run iterations resulting in increased False Positive. If a tool reports data race even once out of five runs, we consider it FP. Especially, when we investigated Intel Inspector, it indicates the min-max data race from 0 to 1. Due to the union policy, we perceived it as FP.

Table IV also shows that the tools' accuracy has a range of [0.7073,0.9024] for C/C++, while the range for Fortran

[1] https://github.com/LLNL/dataracebench/wiki/Tool-Evaluation-Dashboard

Tool	Languages	Version	Compiler	Flag(s)/Environment var
Archer	C/C++	release_60	Clang/LLVM 6.0	export TSAN_OPTIONS="ignore_noninstrumented_modules=1"; -larcher -fopenmp
Intel Inspector		2020(build 603904)	Intel Compiler 19.1.0.166	-collect ti3 -knob scope=extreme -knob stack-depth=16 -knob use-maximum-resources=true -fopenmp
ROMP		20ac93c	GCC 7.4	-g -std=c++11 -fopenmp -lomp
ThreadSanitizer		10.0	Clang/LLVM 10.0	export TSAN_OPTIONS="ignore_noninstrumented_modules=1"; -fopenmp -fsanitize=thread
Archer	Fortran	release_60	Clang/LLVM 6.0/gfortran 10.1.0	-fopenmp -fsanitize=thread -lgfortran; -larcher (cross-compiled)
Intel Inspector		2020(build 603904)	Intel Compiler 19.1.0.166	-free -qopenmp -qopenmp-offload=host -Tf
ROMP		20ac93c	gfortran 7.4	-fopenmp -lomp -ffree-line-length-none
ThreadSanitizer		10.0	Clang/LLVM 10.0.1/gfortran 10.1.0	-ffree-line-length-none -fopenmp -c -fsanitize=thread -lgfortran

TABLE III: Tools Information: version, compiler and flags used.

Tool	Languages	TP	FP	TN	FN	Recall	Specificity	Precision	Accuracy	TSR	Adjusted F1
Archer	C/C++	63	**1**	80	17	0.7875	0.9877	0.9844	0.8882	0.9360	0.8190
Intel Inspector		**71**	40	45	**8**	**0.8987**	0.5294	0.6396	0.7073	0.9535	0.7126
ROMP		59	11	73	18	0.7590	0.8876	0.8630	0.8256	**1.0000**	0.8077
ThreadSanitizer		64	**1**	**84**	15	0.8101	**0.9882**	**0.9846**	**0.9024**	0.9545	**0.8375**
Archer	Fortran	53	**0**	63	16	**0.8154**	**1.0000**	**1.0000**	**0.9063**	0.7711	0.6927
Intel Inspector		**63**	9	**65**	16	0.7975	0.8784	0.8750	0.8366	**0.9217**	**0.7691**
ROMP		62	12	61	18	0.7750	0.8356	0.8378	0.8039	**0.9217**	0.7465
ThreadSanitizer		52	**0**	**65**	**13**	0.8000	**1.0000**	**1.0000**	0.9000	0.7831	0.6961

TABLE IV: Results of Individual Data Race Detection Tools. There are 172 C/C++ test programs and 166 Fortran test programs. 83 C/C++ and 84 Fortran test cases have data race, 89 C/C++, and 82 Fortran test cases do not have data race.

is [0.8039,0.9063]. The accuracy range of C/C++ is significantly worse than a previously reported accuracy range of [0.8500,0.9500] generated by DRB v1.2.0 [21]. For the adjusted F-1 scores, the range is [0.7126, 0.8375] for C/C++ and [0.6927, 0.7691] for Fortran. The range of C/C++ is also worse compared to the earlier range [0.7860, 0.8510] reported by DRB v1.2.0. These range numbers indicate that the new version is indeed more difficult for tools to process.

There were some test programs, listed in Table V for which all the four tools failed to either compile/run successfully or detect the data race rightly. We have already reported in [20] that most of the tools cannot recognize `simd` directive-based data races correctly. Besides, the `mutexinoutset` is an OpenMP 5.0 feature addition employed to enforce mutually exclusive execution with dependencies. There is no data race in `DRB135-taskdep-mutexinoutset-orig-no.c` as a memory location is never accessed concurrently. But two tools reported a false positive. Similarly, in an accelerator-based kernel, the `distribute` directive in `DRB160-nobarrier-orig-gpu-yes.c` does not have an implicit barrier, but all tools missed to report the data race correctly.

Also, none of the tools could identify the incorrect use of the `mergeable` clause in `DRB129-mergeable-taskwait-orig-yes.f95` and reported FN. Primarily Archer encountered CSF for all the accelerator-based kernels in Fortran. One reason for this behavior is that the compiler's version used in these tools does not support GPU offloading in Fortran. It is interesting to see none of the tools could run `taskwait` with `Dependences` based test programs either in C or Fortran and errored out with CSF. Again it is an OpenMP 5.0 feature.

C. Similarity Analysis

We experimented with our similarity analysis based on the Cosine Distance, using the feature vector defined in Sec. II-D. We define the extent of the likeness by its degree value. Table VI is the reference table for our similarity analysis. On investigating, we observed that there are 593 unique loops among the test programs. We calculated the similarity indices by computing the pair-wise distance between loops and compiled them in the table VI.

We have 351,649 (593*593) cosine values. Out of which, 593 pairs are self-to-self distance comparison. We have ignored such scenarios. We used 0.87 as a threshold and investigated the test pairs to check if there are any similar test cases. For example, DRB042, a program from PolyBench, has 408 loops, which are identical to each other. After investigation, we reduced the 408 redundant loops to 2 unique loops. We investigated another similar test pair, DRB001, and DRB002. They are identical in clause, directive, and the tool test results except for the input array types (fixed size vs. varying size). That means they are different by design. But our similarity analysis methodology could not detect this and reported a false positive (false similar). Similarly, DRB011 and DRB016 are reported identical in directives and clauses. But the update statement and the loop body were found to be different. Our similarity analysis methodology currently does not encode the loop bodies, and hence, generated another false positive. Another similar pair discovered is DRB006 and DRB008.

			Archer		Intel		Romp		Tsan	
ID	Name	R	Race	Type	Race	Type	Race	Type	Race	Type
24	DRB024-simdtruedep-orig-yes.c	Y	0	FN	0	FN	0	FN	0	FN
135	DRB135-taskdep-mutexinoutset-orig-no.c	N		CSF		CSF	0-1	FP	2-4	FP
160	DRB160-nobarrier-orig-gpu-yes.c	Y	0	FN	0	FN	0	FN	0	FN
73	DRB073-doall2-orig-yes.f95	Y	0	FN	0	FN	0	FN	0	FN
95	DRB095-doall2-taskloop-orig-yes.f95	Y		CSF	0	FN	0	FN	0	FN
115	DRB115-forsimd-orig-yes.f95	Y	0	FN	0	FN	0	FN	0	FN
129	DRB129-mergeable-taskwait-orig-yes.f95	Y	0	FN	0	FN	0	FN	0	FN
138	DRB138-simdsafelen-orig-yes.f95	Y	0	FN	0	FN	0	FN	0	FN
144	DRB144-critical-missingreduction-orig-gpu-yes.f95	Y		CSF	0	FN	0	FN		CSF
148	DRB148-critical1-orig-gpu-yes.f95	Y		CSF	0	FN	0	FN		CSF
150	DRB150-missinglock1-orig-gpu-yes.f95	Y		CSF	0	FN	0	FN		CSF
160	DRB160-nobarrier-orig-gpu-yes.f95	Y		CSF	0	FN	0	FN		CSF
165	DRB165-taskdep4-orig-omp50-yes.f95	Y		CSF		CSF		CSF		CSF

TABLE V: Selected problematic benchmark programs causing tools to misreport the data race or have compile-time failures.

Cosine Distance	Degree	Similarity Index	Pair Number
[0.87-1]	0°-30°	Identical	159,345
[0.5-0.87)	30°-60°	Moderately Identical	111,316
[-1-0.5)	60°-180°	Distinct	80,988
Total			351,649

TABLE VI: Cosine distance, similarity degree and test pairs.

Though they are alike in all the aspects, an index array has only one element with different values in the two source files. By design, these two are varying test programs, but our analysis does not check the value difference in input arrays. So, it reported a false positive.

Based on the experimental results, our similarity analysis methodology can find some identical patterns. But it has certain limitations. It does not encode information about a loop's body, input data types, and values. Such limitations present an avenue for us to extend our current work to have an enhanced feature vector in the future.

IV. Related Work

There are plenty of OpenMP system's performance evaluation benchmarks available, such as NPB [23], SPEComp [1], and OmpSCR [10]. At the same time, research for data race detection in parallel benchmark has also extended for the Java language. JBench [11] is one of the popular JAVA data race benchmarks with 48 JAVA test cases and three data race tools support. However, no benchmark is specially designed for extensive OpenMP programs' data race detection. DataRaceBench v1.2.0 contains 116 test cases intended for OpenMP data race detection with dynamic data race tools support. In this paper, we included more test cases, extend language support for Fortran, and support customizable partial tests.

Statistical Similarity of Binaries [8] introduced a new analytical approach to detect and analyze the similarity of procedure in the stripped binaries. It uses the concept that compares the resemblance of code by their compositions. By using a control flow graph representation of a procedure, the authors have divided the code into smaller fragments that are small enough as a strand. When there are two sets of strands, they use the proportion of matched value between the query state and the target state as a statistical metric, VCP, signifying the similitude ratio for the semantics similarity. The Tracelet-Based Code Search in Executables [9] introduced a new static method to find similar functions in the code-base. The author divided the function into tracelets, which is a short, continuous, partial trace of execution. For the two sets of tracelets, they use the rewrite method to modify one tracelet to another tracelet and keep the number of rewrite operations as edit distance. Finally, they calculate the tracelet match score by aligning tracelet with LCS variation (edit distance). These two methods analyze the binary code similarity by a similar approach.

V. Conclusion

In this paper, we have presented our latest efforts to extend DataRaceBench to have more comprehensive and up-to-date tests. We have added 56 new C/C++ tests and 166 Fortran tests, covering new GPU and OpenMP 5.0 features, as well as new patterns extracted from literature and other benchmarks. We re-evaluated several tools using the new version of DataRaceBench (v1.3.0) and found that it exposes more limitations of the tools. We also explored a simple similarity analysis methodology to reduce the number of redundant code patterns in the benchmark suite.

In the future, we will enhance our similarity analysis methodology to encode more information about loops and data, such as control-flow and data-flow information. We will add more static data race detection tools [6] into our experiments and expand the feature vector to include the results of new tools also. Since the current kernel extraction process is manual, we will explore developing source-to-source tools to automatically extract kernels. Depending on the interests of the community, we can also add OpenACC and CUDA versions of tests.

Acknowledgment

This work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344, and partially supported by the U.S. Dept. of Energy, Office of Science, ASCR SC-21. LLNL-CONF-813563

REFERENCES

[1] Vishal Aslot, Max Domeika, Rudolf Eigenmann, Greg Gaertner, Wesley B Jones, and Bodo Parady. "SPEComp: A new benchmark suite for measuring parallel computer performance". In: *International Workshop on OpenMP Applications and Tools*. Springer. 2001, pp. 1–10.

[2] Simone Atzeni, Ganesh Gopalakrishnan, Zvonimir Rakamaric, Dong H Ahn, Ignacio Laguna, Martin Schulz, Gregory L Lee, Joachim Protze, and Matthias S Müller. "ARCHER: effectively spotting data races in large OpenMP applications". In: *2016 IEEE international parallel and distributed processing symposium (IPDPS)*. IEEE. 2016, pp. 53–62.

[3] Simone Atzeni, Ganesh Gopalakrishnan, Zvonimir Rakamaric, Ignacio Laguna, Gregory L Lee, and Dong H Ahn. "Sword: A bounded memory-overhead detector of OpenMP data races in production runs". In: *2018 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE. 2018, pp. 845–854.

[4] David H Bailey, Eric Barszcz, John T Barton, David S Browning, Robert L Carter, Leonardo Dagum, Rod A Fatoohi, Paul O Frederickson, Thomas A Lasinski, Rob S Schreiber, et al. "The NAS parallel benchmarks summary and preliminary results". In: *Supercomputing'91: Proceedings of the 1991 ACM/IEEE conference on Supercomputing*. IEEE. 1991, pp. 158–165.

[5] Prithayan Barua, Jun Shirako, Whitney Tsang, Jeeva Paudel, Wang Chen, and Vivek Sarkar. "OMPSan: Static Verification of OpenMP's Data Mapping Constructs". In: *International Workshop on OpenMP*. Springer. 2019, pp. 3–18.

[6] Utpal Bora, Santanu Das, Pankaj Kureja, Saurabh Joshi, Ramakrishna Upadrasta, and Sanjay Rajopadhye. "LLOV: A Fast Static Data-Race Checker for OpenMP Programs". In: *arXiv preprint arXiv:1912.12189* (2019).

[7] THE FLASH CODE. *http : // flash . uchicago . edu / site / flashcode/*.

[8] Yaniv David, Nimrod Partush, and Eran Yahav. "Statistical similarity of binaries". In: *ACM SIGPLAN Notices* 51.6 (2016), pp. 266–280.

[9] Yaniv David and Eran Yahav. "Tracelet-based code search in executables". In: *Acm Sigplan Notices* 49.6 (2014), pp. 349–360.

[10] Antonio J Dorta, Casiano Rodriguez, and Francisco de Sande. "The OpenMP source code repository". In: *13th Euromicro Conference on Parallel, Distributed and Network-Based Processing*. IEEE. 2005, pp. 244–250.

[11] Jian Gao, Xin Yang, Yu Jiang, Han Liu, Weiliang Ying, and Xian Zhang. "Jbench: a dataset of data races for concurrency testing". In: *Proceedings of the 15th International Conference on Mining Software Repositories*. 2018, pp. 6–9.

[12] Yizi Gu and John Mellor-Crummey. "Dynamic data race detection for OpenMP programs". In: *SC18: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE. 2018, pp. 767–778.

[13] Yizi Gu and John Mellor-Crummey. "Dynamic data race detection for OpenMP programs". In: *SC18: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE. 2018, pp. 767–778.

[14] Ok-Kyoon Ha, In-Bon Kuh, Guy Martin Tchamgoue, and Yong-Kee Jun. "On-the-fly detection of data races in OpenMP programs". In: *Proceedings of the 2012 workshop on parallel and distributed systems: Testing, analysis, and debugging*. 2012, pp. 1–10.

[15] 2020. Livermore Computing Lassen System. (2020). Retrieved June 2020 from https://hpc.llnl.gov/hardware/platforms/lassen.

[16] *Intel® Inspector*. Mar. 2020. URL: https://software.intel.com/en-us/inspector.

[17] OpenMP Application Programming Interface. *https : // www . openmp.org/wp-content/uploads/openmp-examples-5.0.0.pdf*.

[18] Marc Jasper, Malte Mues, Maximilian Schlüter, Bernhard Steffen, and Falk Howar. "RERS 2018: CTL, LTL, and Reachability". In: *International Symposium on Leveraging Applications of Formal Methods*. Springer. 2018, pp. 433–447.

[19] Mun-Hye Kang, Ok-Kyoon Ha, Sang-Woo Jun, and Yong-Kee Jun. "A tool for detecting first races in openmp programs". In: *International Conference on Parallel Computing Technologies*. Springer. 2009, pp. 299–303.

[20] Chunhua Liao, Pei-Hung Lin, Joshua Asplund, Markus Schordan, and Ian Karlin. "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*. 2017, pp. 1–14.

[21] Pei-Hung Lin, Chunhua Liao, Markus Schordan, and Ian Karlin. "Exploring Regression of Data Race Detection Tools Using DataRaceBench". In: *2019 IEEE/ACM 3rd International Workshop on Software Correctness for HPC Applications (Correctness)*. IEEE. 2019, pp. 11–18.

[22] Gleison Souza Diniz Mendonça, Chunhua Liao, and Fernando Magno Quintão Pereira. "AutoParBench: a unified test framework for OpenMP-based parallelizers". In: *Proceedings of the 34th ACM International Conference on Supercomputing*. 2020, pp. 1–10.

[23] *NAS Parallel Benchmarks 3.0*. https://github.com/benchmark-subsetting/NPB3.0-omp-C.

[24] Mohammad Norouzi, Felix Wolf, and Ali Jannesari. "Automatic construct selection and variable classification in OpenMP". In: *Proceedings of the ACM International Conference on Supercomputing*. 2019, pp. 330–341.

[25] Robert O'callahan and Jong-Deok Choi. "Hybrid dynamic data race detection". In: *Proceedings of the ninth ACM SIGPLAN symposium on Principles and practice of parallel programming*. 2003, pp. 167–178.

[26] Kento Sato, Ignacio Laguna, Gregory L Lee, Martin Schulz, Christopher M Chambreau, Simone Atzeni, Michael Bentley, Ganesh Gopalakrishnan, Zvonimir Rakamaric, Geof Sawaya, et al. "PRUNERS: Providing reproducibility for uncovering non-deterministic errors in runs on supercomputers". In: *The International Journal of High Performance Computing Applications* 33.5 (2019), pp. 777–783.

[27] Adrian Schmitz, Joachim Protze, Lechen Yu, Simon Schwitanski, and Matthias S Müller. "DataRaceOnAccelerator–A Micro-benchmark Suite for Evaluating Correctness Tools Targeting Accelerators". In: *European Conference on Parallel Processing*. Springer. 2019, pp. 245–257.

[28] Konstantin Serebryany and Timur Iskhodzhanov. "ThreadSanitizer: data race detection in practice". In: *Proceedings of the workshop on binary instrumentation and applications*. 2009, pp. 62–71.

[29] José M Soler, Emilio Artacho, Julian D Gale, Alberto García, Javier Junquera, Pablo Ordejón, and Daniel Sánchez-Portal. "The SIESTA method for ab initio order-N materials simulation". In: *Journal of Physics: Condensed Matter* 14.11 (2002), p. 2745.

[30] Samuel Thayer, Ganesh L Gopalakrishnan, Ian Briggs, Michael Bentley, Dong H Ahn, Ignacio Laguna, and Gregory L Lee. "ArcherGear: data race equivalencing for expeditious HPC debugging". In: *Proceedings of the 25th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. 2020, pp. 425–426.

[31] Passing arguments by reference or by value. *https://www.ibm.com/ support/ knowledgecenter/ SSGH4D_15. 1. 0/ com. ibm. xlf151.aix.doc/proguide/interlang-passargs.html*.

APPENDIX

A. ARTIFACT DESCRIPTION

A.1 Overview

A.1.1 How software can be obtained (if available)?

DataRaceBench v1.3.2 can be found at Github, https://github.com/LLNL/dataracebench.

Alternatively, it can be pulled as a Docker Image directly from the Docker Hub, https://hub.docker.com/repository/docker/yshixyz/dataracebench

A.1.2 Hardware dependencies.

The computation node we used is from the Lassen IBM POWER9 nodes with NVIDIA Volta GPUs as described at https://hpc.llnl.gov/hardware/platforms/lassen . The nodes are machines running Linux operating systems. To obtain results of Intel Inspector, we also ran the DataRanceBench on the Carina machine at the University in North Carolina. This machine has 2 Intel(R) Xeon(R) Gold 6230N CPUs and an NVIDIA Tesla V100 GPU.

Any similar hardware supporting execution of OpenMP programs should be supported. For example, the DRB can be run on AWS Cloud-based GPU instances.

A.1.3 Software dependencies.

As a benchmark suite, DataRaceBench is provided as a Dockerized Service. The images contain Archer, ROMP, and ThreadSanitizer tools (Intel Inspector need a valid license, hence its image is not provided) and DataRaceBench. One may need to update DataRaceBench (DRB) to the latest Github version and mount it into the docker container. The image details are as follows:

- ThreadSanitizer, version 10.0, and compiler support for Clang/LLVM 10.0
- Archer, release version release_60, and compiler support for Clang/LLVM 6.0
- ROMP, version 20ac93c, and compiler support for GCC/gfortran 7.4.0.

We have used Intel Inspector, version 2020(build 603904), and compiler support for Intel Compiler 19.1.0.166 and the latest v1.3.2 version of DRB [2].

A.1.4 Datasets.

All programs in DataRaceBench have built-in data sets. No additional input files are needed. A configuration file is used to specify different sizes of arrays or to run selected programs only.

A.2 Installation

Follow the below steps to use the DataRaceBench:

1) ThreadSanitizer: sudo docker pull yshixyz/-dataracebench:Tsan
2) Archer: sudo docker pull yshixyz/dataracebench:archer
3) ROMP: sudo docker pull yshixyz/dataracebench:romp
4) Intel Inspector: A valid license is required to install and use Intel Inspector.

After downloading the docker image, a user needs to create containers for all the four tools.

1) Archer: docker run -it –name drb_archer yshixyz/-dataracebench:archer
2) ThreadSanitizer: sudo docker run -it –name drb_tsan yshixyz/dataracebench:Tsan
3) ROMP: sudo docker run -it –name drb_romp yshixyz/-dataracebench:romp
4) Intel Inspector: Assume user have built their own Intel image - sudo docker run -it –name drb_intel bash

DRB is automated with the provided scripts which are covered in the next section.

A.3 Evaluation Workflow

Once the above setup is completed, one can start and enter the respective docker container using one of the below commands:

- docker start drb_archer; sudo docker exec -it -u root drb_archer bash
- docker restart drb_Tsan; sudo docker exec -it -u root drb_tsan bash
- docker stop drb_romp; sudo docker exec -it -u root drb_romp bash
- docker rm drb_intel; sudo docker exec -it -u root drb_intel bash

DataRaceBench's execution script, check-data-races.sh, has builtin support for all the four tools. Once we enter a container, we need to set the environment for ROMP and Intel Inspector. To use ROMP, we need to run the following commands to set the context:

Listing 8: Environment setup - ROMP

```
source /home/drb/modules/init/bash
module use /home/drb/spack/Modules/modules/linux-ubuntu18.04-haswell
module load gcc-7.4.0-gcc-7.5.0-moe6s7c
module load llvm-openmp-romp-mod-gcc-7.4.0-cs7qzdu
module load glog-0.3.5-gcc-7.4.0-gqqsqah
module load dyninst-10.1.2-gcc-7.4.0-ohvswfl
export DYNINSTAPI_RT_LIB=/home/drb/spack/Modules/packages/linux-ubuntu18.04-haswell/gcc-7.4.0/dyninst-10.1.2-ohvswflc5hmntqwldkswrmwexnb56hzm/lib/libdyninstAPI_RT.so
export ROMP_PATH=/home/drb/spack/Modules/packages/linux-ubuntu18.04-haswell/gcc-7.4.0/romp-master-i4tglb74pfvppyxbq42iljsrcxmexnrv/lib/libromp.so
export PATH=/home/drb/spack/Modules/packages/linux-ubuntu18.04-haswell/gcc-7.4.0/romp-master-i4tglb74pfvppyxbq42iljsrcxmexnrv/bin:$PATH
```

For Intel Inspector, we need to run the following code to set the environment:

Listing 9: Environment setup - Intel Inspector

```
source /opt/intel/parallel_studio_xe_2020.0.088/bin/psxevars.sh
export PATH=/opt/intel/bin:$PATH
```

Double-check the file location and the added path for the correct environment variables setup. To run the DRB, use:

[2]https://github.com/LLNL/dataracebench

```
./check-data-race.sh --toolname language
(./check-data-race.sh --romp fortran)
```

Use below to see all the possible options: `#show more helpful information for this script`

```
./check-data-races.sh --help
```

We can even run partial test programs using `--customize` flag. One should enter the test programs to run in the `list.def` and tools to test in the `tool.def` file. Rest all the steps remains the same and can be referred to from the above `--help` option.

A.4 Evaluation and Results

Running check-data-races.sh generates csv files stored in a sub-directory, named results, containing multiple lines of information. Each line indicates results of one or several experiments of a given tool under a certain configuration, with fields for:

- The name of the evaluated tool.
- The filename of the microbenchmark.
- True or false (indicating whether the microbenchmark is known to have a data race).
- The number of threads being used for execution.
- Varying length array size (reports N/A if the microbenchmark has no variable length array(s)).
- How many data races the tool reports for this experiment.
- The elapsed time reported in seconds in the experiment.