AutoParBench: A Unified Test Framework for OpenMP-based Parallelizers

Gleison Souza Diniz Mendonça
Universidade Federal de Minas Gerais
Brazil
gleison.mendonca@dcc.ufmg.br

Chunhua Liao
Lawrence Livermore National Laboratory
USA
liao6@llnl.gov

Fernando Magno Quintão Pereira
Universidade Federal de Minas Gerais
Brazil
fernando@dcc.ufmg.br

Abstract

This paper describes AutoParBench, a framework to test OpenMP-based automatic parallelization tools. The core idea of this framework is a common representation, called a "JSON snapshot", that normalizes the output produced by auto-parallelizers. By converting—automatically—this output to the common representation, AutoParBench lets us compare auto-parallelizers among themselves, and compare them semantically against a reference collection. Currently, this reference collection consists of 99 programs with 1,579 loops. AutoParBench produces graphic or quantitative reports that lead to fast bug discovery. By investigating differences in snapshots produced by separate sources, i.e., tool-vs-tool or tool-vs-reference, we have discovered 3 unique bugs in ICC, 2 in DawnCC, 4 in AutoPar and 2 in Cetus. These bugs have been acknowledged, and at least one of them was repaired as direct consequence of this work.

CCS Concepts: • **Software and its engineering** → **Source code generation**; **Empirical software validation**; • **Compu ting methodologies** → *Parallel programming languages.*

Keywords: Benchmark, Automatic Parallelization, OpenMP

ACM Reference Format:
Gleison Souza Diniz Mendonça, Chunhua Liao, and Fernando Magno Quintão Pereira. 2020. AutoParBench: A Unified Test Framework for OpenMP-based Parallelizers. In *2020 International Conference on Supercomputing (ICS '20), June 29-July 2, 2020, Barcelona, Spain.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3392717.3392744

1 Introduction

The growing prominence of OpenMP [11] has contributed to the appearance of many automatic parallelization tools. By annotating C, C++ or Fortran programs with OpenMP directives, said tools are able to generate parallel code without having to deal with minutiae of computer architectures. Examples of automatic parallelizers based on OpenMP include Intel Compilers (ICC), DawnCC [22, 23], AutoPar [19], Mercurium [6], Pluto [8], TaskMiner [27] and Cetus [3]. The existence of so many tools of similar purpose should, in principle, aid development: by comparing their outputs, developers can find correctness or efficiency issues in their implementations. However, as we explain in Section 2, such is not the case. Each parallelizer produces code that, even when semantically equivalent, can use very different syntax. Furthermore, tools like ICC and Pluto might change the program before annotating it. Also, not every tool is source-to-source—ICC, for instance, produced binary code.

Our Thesis. We claim that it is possible to design a common representation that normalizes the output of OpenMP-based parallelizers; hence, allowing automatic comparison between the annotated programs that they produce.

Our Contributions. To support our thesis, we have designed a representation that normalizes programs annotated with OpenMP 4.5 pragmas. This representation, described in Section 3.2, is based on the JavaScript Object Notation (JSON); thus, we call it a *JSON snapshot*. Centered around this normalized representation, described in Section 3, we built a test framework called *AutoParBench*.

AutoParBench contains the infrastructure necessary to compare the output of different automatic parallelization tools. We have also augmented it with a reference collection of annotated programs, which developers can use as the ground-truth when debugging auto-parallelizers. Thus, AutoParBench allows the direct comparison between tools, or the comparison between a tool and the reference collection. In this process, a set of benchmarks is selected as the baseline, and is used to classify the output of the other. As we will explain in Section 3.4, this classification includes true negatives and positives, plus false negatives and positives. The combination of our concrete test infrastructure and the techniques used during its craft brings forward the following contributions:

1 –**Reference collection** (Sec. 3.1): AutoParBench provides developers with a collection of 99 programs, with 1,579 loops, which have been manually annotated with OpenMP directives. These benchmarks were taken from well-known suites, such as NAS [5, 28], Rodinia[10] and DataRaceBench [18]. All the loops have been manually classified as either positive, i.e., parallelizable, example, or negative, i.e., examples which should not be parallelized.

2 –**Infrastructure**: AutoParBench provides parsers to convert C/C++ sources or ICC's reports into JSON snapshots (Sec. 3.3). It also provides evaluators to semantically compare JSON snapshots (Sec. 3.4). From this comparison, AutoParBench generates graphic or quantitative reports (Sec. 3.5).

3 –**Protocol** (Sec. 4.2): in the effort to debug auto-parallelizers, we have designed a protocol to use AutoParBench. This protocol reduces the amount of user intervention necessary to validate warnings, and sorts warnings by relevance. Developers never have to annotate programs to use AutoParBench.

4 –**Results** (Sec. 4.3): the above protocol was used to discover 3, 2, 4 and 2 bugs in ICC, DawnCC, AutoPar and Cetus, respectively. These bugs have been acknowledged as true problems. At least one bug, in DawnCC, has been fixed. Additionally, AutoParBench allows a direct comparison between the quality of the code produced by the auto-parallelizers (Sec. 4.4).

Software AutoParBench is publicly available at https://github.com/LLNL/AutoParBench, under the BSD 3-Clause License.

ICS '20, June 29-July 2, 2020, Barcelona, Spain

ACM ISBN 978-1-4503-7983-0/20/06…$15.00
https://doi.org/10.1145/3392717.3392744

2 Challenges

Having a common framework that allows testing different automatic parallelization tools is difficult, because tools may generate very different outputs—some not even in textual format. Although distinct, they might represent correct parallelizations of the same program. In this section, we list the major challenges we face when creating AutoParBench.

Challenge 1. A program may be amenable to different parallelization strategies.

There exist different parallel patterns, e.g., data and task parallelism. Target devices can also vary, e.g., CPUs and GPUs. As an example, Listing 1 shows how different targets lead to different parallelizations of the same program.

Listing 1. Loop parallelization using two different strategies: vectorization or GPU acceleration.

```
1  void CPU_vectorization(int *a, int len) {
2    #pragma omp simd
3    for (int i=0; i<len; i++)
4      a[i]= i;
5  }
6  void GPU_parallelization(int *a, int len) {
7    #pragma omp target map(from:a[0:len])
8    #pragma omp teams distribute parallel for
9    for (int i=0; i<len; i++)
10     a[i]= i;
11 }
```

The programs in Listing 1 are semantically different: the loop in `CPU_vectorization` will be vectorized, whereas the loop in `GPU_parallelization` will be accelerated in a GPU. Nevertheless, both are correct; hence, both should be acceptable by an automatic validation tool. Key to solve this challenge is a categorization of potential parallelizations of a program, which we shall discuss in Section 3.1.2.

Challenge 2. Code might be amenable to conditional or multi-versioning parallelization.

Pointer aliasing might hinder parallelization due to potential dependencies. Dependencies occur when pointers dereference overlapping memory regions. A combination of code versioning and conditional checks is a technique adopted by tools like ICC, DawnCC and the LLVM's code vectorizer [1] to avoid dependencies at runtime. As an example, Listing 2 shows code produced by DawnCC.

Listing 2. Conditionalized Parallelization

```
1  void foo (int *dest, int *src, int n) {
2    char ovrlp = ((void*)(dest)<(void*)(src+n));
3    ovrlp &= ((void*)(src)<(void*)(dest+n));
4    #pragma omp parallel for if(!ovrlp)
5    for (int i = 0; i < n; i++)
6      dest[i] = src[i];
7  }
```

Function `foo` contains a loop that is parallel, as long as the two arrays, `dest` and `src`, do not overlap. The guard using `!ovrlp`, at line 4, only allows parallel execution when the two arrays cover disjoint memory regions. Thus, the program in Listing 2 is correct, as long as the guard is present. Section 3.2.1 explains how we evaluate multi-versioned loops.

Challenge 3. Nested loops might be parallelized in a combinatorial number of ways.

Although a loop might be parallelizable, this transformation might not be profitable. This phenomenon happens, for instance, when the work in the loop body is not enough to pay off the cost of creating threads or offloading data. Nested loops may contain multiple levels of parallelizable loops. The cost model of a tool might lead to the parallelization of one, or several of these loops. Thus, the fact that a tool might leave some loops untouched does not necessarily imply that the tool has not been able to identify the potential parallelism. Listing 3 illustrates this issue with an example. In Section 3.2.1 we explain how we represent loops at different granularities, and in Section 3.1.3 we discuss how to deal with unprofitable parallelization.

Listing 3. Nested loops can be parallelized in different ways.

```
1  void both_parallel (int **a, int len) {
2    int i, j;
3    #pragma omp parallel for private(j)
4    for (i=0; i< len; i++)
5      #pragma omp parallel for simd
6      for (j=0; j<len; j++)
7        a[i][j] = (i * len + j + 0.5);
8  }
9  void outer_parallel (int **a, int len) {
10   int i, j;
11   #pragma omp parallel for private(j)
12   for (i=0; i< len; i++)
13     for (j=0; j<len; j++)
14       a[i][j] = (i * len + j + 0.5);
15 }
```

Challenge 4. There are multiple data mapping variants for accelerator offloading.

When parallelizing codes for accelerators, data often need to be transferred between locations. OpenMP provides various directives to specify such data transfers; however, the data mapping pragmas may not be syntactically associated with the offloading directive. Listing 4 illustrates this issue.

Listing 4. Data mapping variants for target directives

```
1  void target_loop (int *a, int len) {
2    int tmp, i;
3    #pragma omp target parallel for private(tmp) map(a[0:len])
4    for (i=0;i<len;i++) {
5      tmp =a[i]+i;
6      a[i] = tmp;
7    }
8  }
9  void target_context (int *a, int len) {
10   int tmp, i;
11   #pragma omp target data map(a[0:len]) {
12     #pragma omp target parallel for private(tmp)
13     for (i = 0; i < len; i++) {
14       tmp = a[i] + i;
15       a[i] = tmp;
16     }
17   }
18 }
```

Function `target_loop` in Listing 4 contains one loop parallelized with the `target` directive combined with a `map` clause. Function `target_context`, in turn, sets up GPU parallelization in two steps, via a combination of directives "`target data`" and "`target parallel for`". A verification tool needs to match the semantics of these two ways of data offloading to a device, as we explain in Section 3.2.2.

Challenge 5. OpenMP clauses can use expressions parameterized by different program symbols.

Several OpenMP clauses are parameterized by program symbols, i.e., user-defined name. For example, the map and the depends clauses receive an array name followed by a memory range. memory ranges are expressions that use program symbols. These expressions complicate the verification of OpenMP clauses because they can be written in an unbounded number of ways, all of which encode a similarly correct semantics.

Listing 5. map clauses with variables

```
void symbolic_map (int *a, int n) {
  int len = 100, i;
  #pragma omp target data map(a[0:len])
  #pragma omp target parallel for private(i)
  for (i = 0; i < len; i++) {
    a[i]++;
  }
}
void numeric_map (int *a, int n) {
  int len = 100, i;
  #pragma omp target data map(tofrom: a[0:100])
  #pragma omp target parallel for private(i)
  for (i = 0; i < len; i++) {
    a[i]++;
  }
}
```

Listing 5 illustrates this challenge. The expressions `len` and `100` are semantically equivalent. A reference output for this program cannot simply settle for one of them, because a tool might use the other, and still deliver correct code. In Section 3.4 we explain how AutoParBench handles differences in symbols.

Challenge 6. Auto-parallelization tools can apply transformations in programs, such as loop-splitting and loop-coalescing.

There is a long list of code transformations that can be used to enable automatic parallelization [30]. Such transformations may render the parallel program very different than its original –sequential– version. Therefore, to be effective, a verification tool must be able to match the original and transformed programs. Listing 6 illustrates this issue.

Listing 6. Example of loop amenable to coalescing

```
int b[1000][1000];
void original_loop(int n, int m) {
  for (int i=0; i<n; i++)
    for (int j=0; j<m; j++)
      b[i][j] = 0.5;
}
void coalesced_loop(int n, int m) {
  #pragma omp parallel for
  for (int index=0; index<(n * m); index++) {
    int i = index / n;
    int j = index % m;
    b[i][j] = 0.5;
  }
}
```

The second routine in Listing 6, the function `coalesced_loop` is produced by ICC, via *loop coalescing* [25]. A common format that allows using function `coalesced_loop` as the reference output for the parallelization of function `original_loop` (Listing 6) must provide hooks to match these two different programs. In Section 3.3 we elaborate on how we deal with such transformations.

Challenge 7. Auto-parallelization tools can produce outputs in different formats.

Automatic parallelization tools can be source-to-source or source-to-binary. The former provide information about the parallelization in source files, via human-readable OpenMP annotations. The latter implement parallelization directly into the binary code. For instance, AutoPar, Cetus and DawnCC are source-to-source: they generate a C/C++ program annotated with OpenMP pragmas. ICC, in turn, produces binary code plus an optional report with debugging information. Listing 7 shows an example of such a report.

Listing 7. Example of optimization report produced by ICC

```
LOOP BEGIN at DRB020-privatemissing-var-yes.c(60,3)
   remark #17109: LOOP WAS AUTO-PARALLELIZED
   remark #17101: parallel loop shared={ .2 } private={ }
        firstprivate={ len a i } lastprivate={ } firstlastprivate
        ={ } reduction={ }
   remark #15540: loop was not vectorized: auto-vectorization is
        disabled with -no-vec flag
   remark #25439: unrolled with remainder by 2
LOOP END
```

The report in Listing 7 contains the information necessary to recover the transformations that ICC has carried out in a given program. A comprehensive test framework should be able to handle different output formats to enable comparison among them. We explain how we deal with different output formats in Section 3.3.

3 The Design of AutoParBench

AutoParBench consists of a reference collection of benchmarks, an intermediate representation (IR) of parallel code, software that translates annotated programs into the IR, a harness that compares tools by normalizing their outputs via the IR, and configurable scripts and reports. This section discusses each one of these parts.

3.1 The Reference Collection

The reference collection is a set of ready-to-use C/C++ benchmarks intended to serve as the ground-truth for automatic parallelization tools. Currently, this collection contains 85 micro-kernels, plus fourteen larger programs from Rodinia and NAS. The provenance of these benchmarks is detailed in Section 4.1, Page 7. The larger benchmarks let AutoParBench compare the performance of parallel programs. However, AutoParBench's main goal is to find correctness bugs in auto-parallelizers, not to measure their performance.

Programs in the reference collection contain a sequential and a parallel version, the latter annotated with OpenMP 4.5 pragmas. Annotations are for correctness, not for efficiency; hence, even small loops, when data-race free, are annotated. We compare results of the two versions and run each annotated program with Intel Inspector[1] to ensure correctness. Most of the annotations were already present in the original benchmarks. We had to annotate race-free loops in the fourteen large programs—details are described in AutoParBench's public distribution.

Notice that users of our framework do not have to generate a reference collection manually. As already hinted in Section 1, AutoParBench allows the direct comparison between two tools. One of them will be considered the reference, when counting false positives and negatives. Indeed, we have used ICC as the reference in some experiments. We provide a reference collection because

[1] https://software.intel.com/en-us/inspector

we realized that some decisions of ICC could be improved to ease debugging. As we shall explain in Section 4.3, with the new reference collection, we have reported three bugs in ICC that were later confirmed.

3.1.1 Extending the Reference Collection. We have designed AutoParBench to allow easy acceptance of external contributions from the community. The addition of new benchmarks to the reference collection is a desirable consequence of this design. To support the addition of new benchmarks, AutoParBench's reference collection is partitioned into self-contained programs. Thus, the addition of new programs does not cause modifications in the structure or composition of the framework. In other words, benchmarks and scripts already there remain untouched. To add a new program to the reference collection, users can either start with a plain-C/C++ program, and annotate it, either manually or via a trusted parallelizer; or they can start with an annotated program, and strip its annotations off, to obtain the sequential version. AutoParBench provides users with a correctness step, which uses Intel Inspector, plus output comparison, to check if the new addition is sound.

3.1.2 Parallelization Strategies. Automatic parallelization tools may apply different parallelization strategies. For example, AutoPar and Cetus deal with multi-core parallelization (CPU Threading); DawnCC targets accelerators (GPU Threading). ICC, in turn, supports both, albeit not at the same time. To accommodate these differences, benchmarks in the reference collection are grouped into the four categories seen in Table 1. Categorization lets us use AutoParBench to verify the output of tools that target different software/hardware features. For instance, the categories "CPU SIMD" and "GPU SIMD" contain the same benchmarks. However, in the first category, benchmarks are annotated with vectorizing pragmas for CPUs; in the second, they target accelerators. This categorization helps AutoParBench to provide a solution to Challenge 1.

Categories	Example OpenMP Directives
CPU Threading	for, parallel, parallel for, task, task loop
CPU SIMD	simd, for simd, parallel for simd task loop simd
GPU Threading	target parallel for, teams distribute target teams distribute parallel for teams distribute parallel for target teams distribute
GPU SIMD	target simd target parallel for simd target teams distribute simd teams distribute parallel for simd target teams distribute parallel for simd

Table 1. Categories of parallel strategies.

3.1.3 Unprofitable Parallelization. The reference collection is not performance-focused. We have strived to annotate every loop that could be parallelized, even when such annotations are clearly unprofitable. We try our best to configure each tool to parallelize as many loops as possible without considering profitability. If a tool, for any reason, refuses to parallelize one of these loops, then AutoParBench reports a false negative. False negatives are not necessarily bugs, although they might account for inefficiencies. This approach, combined with the possibility to execute the program, lets AutoParBench provides a best-effort solution to Challenge 3.

3.2 The Intermediate Representation

The core equipment that AutoParBench uses to compare the output of different tools is an intermediate representation using *JSON snapshots*. A snapshot is a file that represents the parallelization decision of a code region within a benchmark program. Thus, the application of a parallelizer onto a program might yield multiple snapshots—each one representing a particular code region in that program. Snapshots are produced by a *translator*, i.e., a piece of software that parses the output of a tool, and produces the corresponding JSON snapshots. As we shall explain in Section 3.3, currently AutoParBench provides two translators: a general one, that reads C/C++ programs augmented with OpenMP pragmas, and another specific to ICC, which is not a source-to-source compiler. Figure 1 illustrates how snapshots are produced.

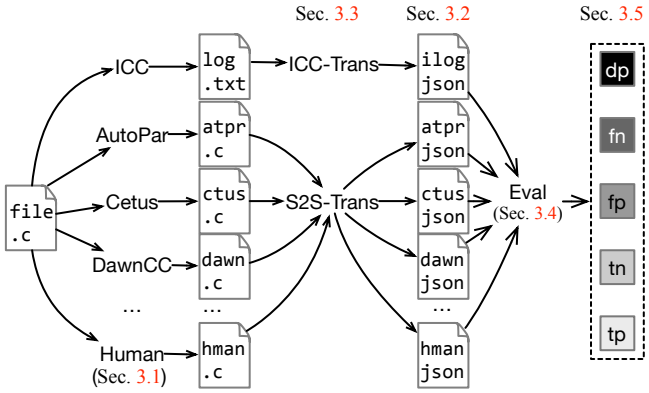

Figure 1. Production and evaluation of Snapshots for C/C++.

3.2.1 Snapshot Objects. A JSON snapshot, that is, the normalized representation of a code region potentially annotated with OpenMP pragmas, is a human-readable text file that contains multiple *objects*. We consider two categories of code regions: loop vs. non-loop regions. The loop regions represents C/C++ loops such as `while`, `do-while` and `for`. Figure 2 shows common features for both types of regions, as well as loop-specific features.

Example 3.1. Figure 3 shows an example of a loop object, together with two different programs that lead to it. Notice that, except for the fields `file` and `line`, the identical object represents both programs.

Non-loop objects represent code regions that can be annotated with OpenMP pragmas, but that are not loops. Example 3.2 shows code that produces this kind of object. We have opted to separate loop and non-loop objects to allow comparing tools' results at a granularity smaller or larger than loop blocks. This strategy lets AutoParBench's evaluator pinpoint the parts of a parallel loop that are identical across tools, and the parts that differ.

Example 3.2. Figure 4 shows the objects extracted from a single loop. The non-loop object denotes the invocation of the `printf` function, which has been annotated with the `ordered` directive. The `atomic` directive, not shown in this example, can also be a source of non-loop objects.

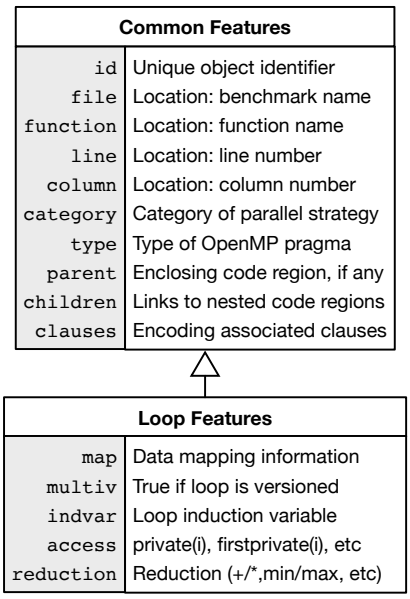

Figure 2. Fields for JSON snapshot objects.

```
1 void main() {
2   int *a = (int*)malloc(400);
3   #pragma omp parallel
4   #pragma omp for
5   for (int i=0; i<100; i++) {
6     a[i] = 1;
7   }
8 }
```
(a)

```
1 void main() {
2   int *a = (int*)malloc(400);
3   #pragma omp parallel for
4   for (int i=0; i<100; i++) {
5     a[i] = 1;
6   }
7 }
```
(b)

```
{
"id":"1"{
"file":"[redacted]",
"function":"main",
"line":"[redacted]",
"column":"3",
"category":"CPU Thr.",
"type":"parallel for",
"multiv":"false",
"indvar":"i"
}
```
(c)

Figure 3. (a-b) Two semantically equivalent parallelizations of the same program. (c) The corresponding loop object.

Notice that our choice of JSON as the intermediate representation is based mostly in our personal taste. We could have used other textual representations that support encoding hierarchical structures, such as XML or YAML, for instance. The advantage of using JSON, over, for instance, designing a domain specific language, is the availability of tools to parse and serialize it in mainstream languages such as Python, Ruby, Java and JavaScript.

Multi-Versioning. The `multiv` field of a JSON object indicates that a loop has been replicated by the auto-parallelizer. Conditional parallelization, as seen in Listing 2, falls into this category. Every loop object that describes one of the multiple versions of the same code has the same `id` field. When evaluating tools, the evaluator (to be discussed in Section 3.4) compares only the parallel version of a multi-versioned loop. This approach provides us with a pragmatic solution to Challenge 2.

3.2.2 Semantic Equivalences. JSON snapshots enable the normalization of syntactically different codes into semantically equivalent classes of annotations. The need for such normalization stems

```
#pragma omp parallel for ordered private(i)
for (i=0; i<len; i++)
  #pragma omp ordered
  printf("%d\n", a[i]);
```

```
{"id":"2",
"file":"[redacted]",
"function":"main",
"line":"72",
"column":"13",
"type":"ordered",
"parent":"1"
}
```

```
{"id":"1",
"file":"[redacted]",
"function":"main",
"line":"70",
"column":"3",
"category":"CPU Threading",
"type":"parallel for",
"children":["2"],
"clause":["ordered"],
"multiv":"false",
"indvar":"i",
"access":"private(i)"
}
```

Figure 4. JSON snapshots extracted from a C program representing a loop and a non-loop object.

from the different syntaxes that the OpenMP standard accepts to represent the same parallelization concept. Presently, we consider four classes of normalizations:

- Joining separate constructions. For instance, "`omp parallel`" and "`omp for`" are combined into "`omp parallel for`", as seen in Example 3.1, or, similarly, "`omp target data`" and "`omp target`" give "`omp target data`". This helps AutoParBench deal with Challenge 4.
- Making explicit every implicit data-sharing clause. For instance, AutoParBench sets as `private` the loop index variable, unless this variable is explicitly annotated with a different data-sharing mode.
- Reduction operator "minus" is normalized into the "plus" operator since they are semantically the same.
- Constant variables are replaced with their values whenever possible: e.g. `len` is replaced with `100` in the snapshot that represents Listing 4.

We use the relative position of a code region within its enclosing function to assign IDs in a snapshot. For example, the first loop in a function will get the ID 1, regardless of its line number. This approach helps matching regions in files produced by tools with those in the reference collection. JSON snapshots also tolerate positional syntactic differences for variables in OpenMP directives, e.g., "`to a, from b`" vs "`from b, to a`", as each one of the occurrences become individual fields within the JSON file.

3.3 The Translators

Translators are needed by AutoParBench to convert the output of an automatic parallelizer into a JSON snapshot. Currently, AutoParBench provides two translators. The first is used by tools that perform source-to-source code annotation, such as DawnCC, AutoPar and Cetus. The second is exclusive to ICC. ICC does not annotate source code; instead, it produces parallel binary code. Additionally, ICC can produce a textual optimization report when used with proper options turned on. Our ICC translator parses this textual report, and produces the JSON snapshot out of it. The translators collectively help address Challenge 7.

The source-to-source translator is implemented as a clang plugin. The ICC translator is implemented as a standalone parser. Both these tools recognize most OpenMP 4.5 clauses; however, at the time of this writing, task-oriented pragmas [2] are not fully supported. This decision was pragmatic: none of the benchmarks in the reference collection contain task-oriented directives. Furthermore, the only task annotator that we are aware of is TaskMiner [27], still a research artifact limited to small programs.

Dealing with Loop Transformations. AutoParBench also deals with Challenge 6 during this translation step. Out of all the tools currently evaluated with AutoParBench, only ICC performs a transformation: loop coalescing. Coalescing consists in merging into a single loop two successive loops that have a common trip count. When parsing ICC's reports, AutoParBench identifies the loops that have been coalesced. By reading the original input file, the translator recovers the identifier of the eliminated loop, as well as its location (line and column). A snapshot is created for each loop that has been eliminated. This object is written in a way to denote the same parallel semantics of the coalesced loop.

3.4 The Evaluator

AutoParBench defines positive and negative tests in the context of automatic parallelization. A positive test contains a parallelizable code region (mostly a loop); a negative test contains a code region that should not be parallelized. For a given comparison, a tool can generate the results below. Notice that these results are relative to a *reference*. This reference can be the reference collection of Section 3.1, or it can be the output of another automatic parallelizer:

- *True Positive (TP)*: a tool parallelized code that is syntactically or semantically equivalent to the reference parallel code.
- *False Positive (FP)*: a tool parallelized code that is not marked as parallel in the reference.
- *True Negative (TN)*: a tool avoided parallelizing code that is not parallel in the reference.
- *False Negative (FN)*: a tool did not parallelize code, although it is parallelizable in the reference.
- *Different Parallelization (DP)*: code produced by a tool is parallel; however, AutoParBench does not (yet) recognize it as semantically equivalent to the reference.
- *Crash*: A tool crashes when parallelizing the code.

These results let us compute four standard metrics for every tool that AutoParBench evaluates: precision = $TP/(TP + FP)$, recall = $TP/(TP + FN)$, accuracy = $(TP + TN)/(TP + TN + FP + FN)$ and the F1-score = $2 \times precision \times recall/(precision + recall)$. JSON objects in the "different parallelization" and "Crash" categories are not included in these metrics, although the latter contributes to compute a *weighted F1-score* (Section 4.2). Warnings reported as different parallelization give us a means to continually evolve AutoParBench. By investigating these warnings, we can refine them further as either true positives or wrong parallelizations (e.g. missing an indispensable data-sharing clause).

Different Parallelizations. Semantic equivalence and positional independence let AutoParBench match many syntactically different annotations as true positives. However, there are annotations that AutoParBench cannot yet classify as equivalent or different. To aid debugging, when reporting an occurrence of "different parallelization", AutoParBench also specifies the type of difference to enable manual investigation. The investigation may lead to a verdict of either true positive or wrong parallelization. An example of wrong parallelization occurs when a tool correctly parallelizes a loop, but misses the insertion of a reduction clause.

Warnings of different parallelization are often due to symbolic expressions. Symbolic expressions specify ranges of data, such as the intervals `[a:len]` and `0:100` at lines 3 and 11 of Listing 4. Proving that general range expressions are equivalent amounts to solving Diophantine Equations, an undecidable problem. To mitigate this issue, AutoParBench compares the program variables used in each expression. Hence, it can report that syntactically different arithmetic expressions are built on the same symbols. This is the strategy currently used to deal with Challenge 5.

3.5 Configurable Scripts and Reports

AutoParBench provides a collection of configurable scripts to evaluate auto-parallelizers. The baseline of evaluation is configurable, because it is possible to use the output of a tool as ground-truth. In other words, although AutoParBench provides a reference collection, which we use as the ground-truth when hunting for bugs, nothing hinders developers from using the output of a trusted tool as the baseline. Such comparisons lead to *actionable items*, that is, indications of bugs or inefficiencies that developers can investigate. The result of a comparison is either a textual or a graphical report. The latter gives developers an easy-to-see idea on how close or distant is the output produced by different tools.

Example 3.3. Figure 5 shows a graphical report produced by AutoParBench. Each cell in the matrix on the right represents the comparison result between the outcome of a parallelizer and the corresponding ground-truth reference. Results are either TP, TN, FP, FN, or DP, indicated by different grayscale shades. Similar reports can be generated for any pair of parallelizers, by treating one of them as the reference.

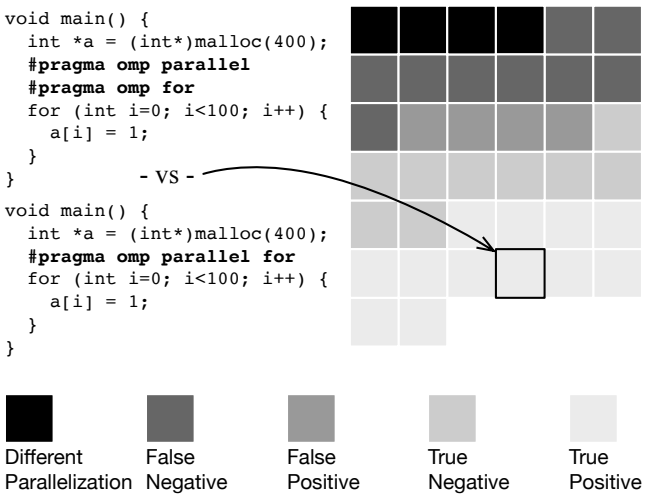

Figure 5. A graphical report produced by AutoParBench. The program on the top was produced by auto-parallelizer; the program on the bottom is part of the reference collection.

4 Experiments

This section describes the experiments used to evaluate AutoParBench Version 1.0. In particular, we:

- provide overall statistics about the benchmarks included in AutoParBench in Section 4.1;
- compare the output produced by different parallelizers in Section 4.2;
- demonstrate that AutoParBench is able to uncover bugs in both academic and industrial tools in Section 4.3;
- carry out a performance comparison between different parallelization approaches in Section 4.4.

Compilers and Tools Selected. We use AutoParBench to evaluate three source-to-source tools: AutoPar (0.9.10.235), DawnCC (3.7.0), and Cetus (1.4.4), and one source-to-binary tool: ICC (19.0.4.243).
Runtime Setup. Results were produced on an 8-core Intel(R) Core (TM) i7-6700T at 3.6GHz with 8GB of RAM running Ubuntu 18.04, featuring a GPU Intel HD Graphics 530.

4.1 The Framework

Provenance. Currently, AutoParBench's reference collection contains 85 programs taken from DataRaceBench v1.2.0, plus 6 programs from the NAS Parallel Benchmark Suite v3.0 and 8 programs from Rodinia v3.1. Together, these 99 programs give us 1,579 loops. Loops are classified as positive or negative. Positive examples are amenable to parallelization via one of the strategies that AutoParBench recognizes (as seen in Table 1). Negative examples are loops which should not be parallelized due to data dependencies.
Size of Benchmarks. Figure 6 (top) groups benchmarks per line of code. Each bucket in the X-axis indicates a range. For example bucket "< 100" includes benchmarks with less than 100 lines of source code. The Y-axis indicates how many benchmarks fall into a given range. Most of the benchmarks are small; however, some performance oriented programs have more than 1,000 lines of code. Figure 6 (bottom) shows a histogram of the number of positive and negative loops per benchmarks. Eight benchmarks contain only one loop. Our largest benchmark, SP, contains 317 loops.

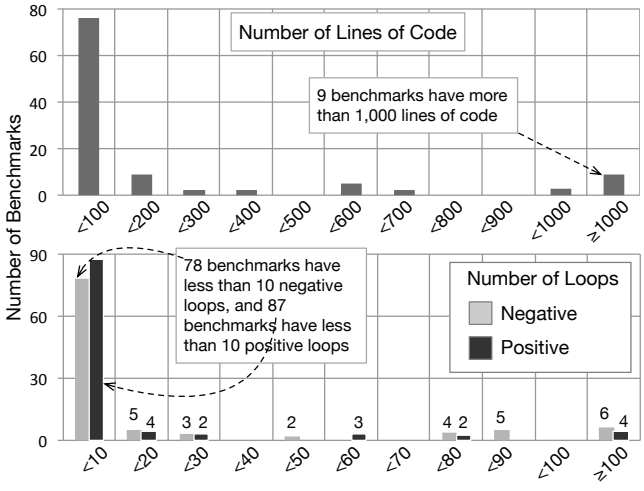

Figure 6. Lines of code and loops per benchmarks.

4.2 Comparing the Output of Tools

Standard Metrics. To generate the metrics defined in Section 3.4, we compare the four auto-parallelizers with the reference collection. Table 2 shows how the tools fare in terms of precision, recall, accuracy and F1-score. It also shows how many programs (Pr) and loops (Tt) were given to each tool, and how many loops (Lp) were properly analyzed without causing a tool to crash or generate error messages. The number of loops analyzed per tool differs in Table 2 for two reasons. First, some benchmarks are specific to particular targets. For instance, DawnCC and ICC-Simd work on data parallel programs. Second, some benchmarks cause crashes in the parallelizer, and their loops are not analyzed.

Tool	Prec.	Rec.	Acc.	F1	Pr/Lp/Tt	WF1
AutoPar	0.85	0.89	0.85	0.87	99/1381/1579	0.76
Cetus	0.92	0.93	0.95	0.93	99/430/1579	0.25
ICC-Cost	0.91	0.28	0.61	0.43	99/1579/1579	0.43
ICC-Full	0.91	0.83	0.88	0.87	99/1579/1579	0.87
DawnCC	1.00	0.30	0.73	0.46	17/63/63	0.46
ICC-Simd	1.00	0.59	0.65	0.74	17/63/63	0.74

Table 2. Summary of results. Pr = number of programs; Lp = number of loops properly handled; Tt = total of loops given to the tool; WF1 = Weighted F1-Score. The higher, the better.

Table 2 considers three different uses of the Intel compiler. ICC-Full parallelizes every loop that ICC deems parallel, regardless of potential runtime benefits. ICC-Cost uses the compiler's default cost model to only parallelize profitable loops. ICC-Simd adds vectorization on top of ICC-Cost. These tools are used with the following flags:

- **AutoPar** -c -w -rose:verbose 0
- **DawnCC** -writeInFile -stats -Emit-GPU=false -Run-Mode= false -Emit-Parallel=true -Emit-OMP=1 -Ptr-licm=true -Ptr-region=true -Restrictifier=true -Memory-Coalescing=true
- **Cetus** -parallelize-loops=2 -ompGen=2 -profitable-omp=0
- **ICC-Cost** -no-vec -fno-inline -parallel -qopt-report-phase= all -qopt-report=5
- **ICC-Full** -par-threshold0 -no-vec -fno-inline -parallel -qopt-report-phase=all -qopt-report=5
- **ICC-Simd** -par-threshold0 -qopt-report-phase=all -qopt-report=5 -vec-threshold0 -fno-inline -parallel

For AutoPar, we used the default flags and enabled warnings. For DawnCC and Cetus, we enabled the available optimizations. ICC-Full uses the compiler's default set of flags and disables the cost model (`-par-threshold0`). ICC-Cost enables the cost model instead. ICC-Simd does not use the cost model, and enables vectorization whenever possible (-vec-threshold0).

The tools do not use all the same baseline, i.e., the same subset of the reference collection. ICC-Full, ICC-Cost, AutoPar and Cetus use, as baseline, the 1,579 loops that, in the reference collection, are in the category *CPU Threading* (see Table 1). ICC-Simd uses as reference the category *CPU SIMD*, and DawnCC uses *GPU SIMD*. These categories comprise 63 loops from 17 programs. These differences are due to the nature of these tools: whereas DawnCC and ICC-Simd transform programs to be used in SIMD-like accelerators, the other tools target multi-core CPUs.
The Weighted F1-Score. The last column of Table 2, WF1, is a *weighted* F1-score. This number, for a given tool, is defined as $WF1 = F1 \times T_t / L_p$, where L_p is the total of loops given to the tool, and T_t is the number of programs that the tool was able to handle. This weighted version of the F1-score gives us a measure of how close the output of a tool is from the reference collection. Example 4.1 illustrates its importance.

Example 4.1 (Weighted F1-Score)**.** Although Cetus and ICC-Full receive the same set of 1,579 loops from the *CPU Threading* category, the former analyzes only 430 of them. Cetus' F1-score, in this universe of 430 programs, is 0.93. Thus, its weighted F1-score is $0.93 \times 430/1,579 = 0.25$. ICC-Full's weighted score is 0.87, the same as its F1-score, because this tool analyzed all the input programs. Thus, ICC-Full is closer to our ground-truth than Cetus.

The WF1-score points out which tool is closer to the reference collection; however, it is not an indication of which tool is better, or more likely to present bugs. A tool that refuses to parallelize every loop will have a WF1-score of zero, but will be bug-free. Precision (Prec. in Table 2) is a better sign of potential for bugs, as it takes the number of false positives in consideration. As we shall see in Section 4.3, false positives mark loops that are likely to expose a tool's bugs.

Example 4.2. ICC-Cost refuses to parallelize several loops, which are deemed unprofitable by its cost model. Such abstentions lead to numerous false negatives; hence, low recall; however, they do not compromise ICC-Cost's precision, which remain high (0.91).

Graphical Comparison. Figure 7 provides a graphical comparison between tools. Each part of the figure is a grid; each cell of this grid is the result of the comparison between the output of a tool and the annotated baseline. We use light grayscale shades to represent true positives and true negatives; thus, the lighter the grid, the closer is the tool's output to the baseline. False positives and different parallelization strategies might demand investigation from developers. These outcomes receive darker shades in Figure 7. Thus, the darker the figure, the larger its potential to present bugs.

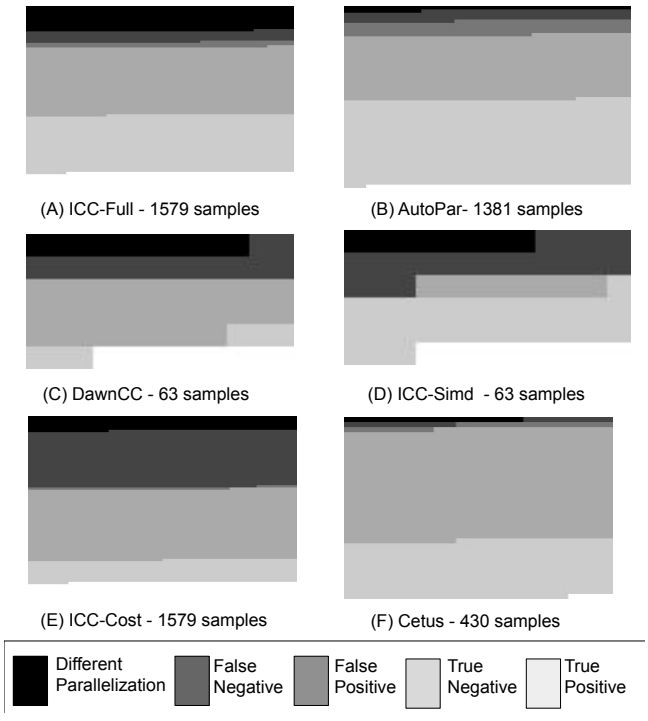

Figure 7. Graphical comparison between tools and baseline.

4.3 Actionable Results

Aided by AutoParBench, we have reported several bugs to developers of compilers and tools. At the time of this writing, we have acknowledgements of 3 bugs uncovered in ICC, 2 bugs uncovered in DawnCC, 4 bugs uncovered in AutoPar, and 2 bugs uncovered in Cetus. We expect that more bugs will emerge, as we are still investigating warnings.

A protocol to investigate results. We have adopted a methodology to rank warnings. Said methodology aims to improve the debugging process, as it prioritizes bugs that are more likely to be fixed by tool developers. We rank suspicious results as follows: 1^{st} tier: tool crashes; 2^{nd} tier: parallel program produces wrong result (Example 4.3); 3^{rd} tier: *false positive* reports (Examples 4.5 and 4.4); 4^{th} tier: reports of the *different parallelization* category (Example 4.6); and 5^{th} tier: *false negative* reports (Example 4.7).

Tool	1^{st} tier	2^{nd} tier	3^{th} tier	4^{th} tier	5^{th} tier
ICC-Cost	0	0	18	126	491
ICC-Full	0	1	45	217	96
ICC-Smid	0	0	0	8	19
Cetus	11	1	11	8	9
AutoPar	5	1	111	33	81
DawnCC	0	0	0	10	14
Total	16	3	185	402	710

Table 3. Number of suspicious results grouped by tiers.

Table 3 shows the number of occurrences of suspicious results in the different tiers, considering the six experiments graphically reported in Figure 7. As it would be natural to expect, most of the warnings are concentrated in the less pressing tiers of the investigation protocol. Consequently, developers can focus on more serious bugs, leaving less severe warnings for posterior inspection. Notice that the number of warnings is not correlated with the number of bugs that we have reported, because the same bug may cause warnings in several different benchmarks.

Examples of confirmed bugs. We describe below some of the confirmed bugs that we have reported. Figure 8 illustrates them.

```
void main(int argc, char *argv[]) {
  int i, len = argc;
  int x = argc > 2 ? len - 2 : 0;
  int* a = (int*)malloc(len * sizeof(int));
  for (i = 0; i < len; i++) {
    a[x] = i; x=i;
  }
  for (i = 0; i < len - 1; i++)
    printf("%d ", a[i]);
  printf("x=%d",x);
}
```
(a)

```
for (i=0;i<len;i++) {
  c[j]+=a[i]*b[i];
  j++;
}
```
(b)

```
#pragma omp parallel \
    for private(i) \
    reduction(+: c[i+j])
for (i=0; i<len; i ++ ) {
  c[i+j]+=(a[i]*b[i]);
}
j+=len;
```
(c)

```
#pragma omp parallel \
    for private(i) linear(j)
for (i=0;i<len;i++) {
  c[j]+=a[i]*b[i];
  j++;
}
```
(d)

Figure 8. (a) Program that caused a false positive in ICC. (b) Sequential program that uncovered bug in Cetus. (c) Code produced by Cetus. (d) Code in reference collection.

Example 4.3 (Parallel program crashes)**.** Figs. 8(b-d) show a bug that was discovered in Cetus. Cetus, when given the program in Figure 8 (b), produces the code in Figure 8 (c). Cetus extracts variable `j` from the loop, and transforms it into a reduction. Said reduction causes a runtime crash.

Example 4.4 (False Positive in ICC)**.** ICC parallelizes the first loop in the program in Figure 8 (a). However, when `argc` is greater than 2, a race condition occurs in `a[len-2]`, caused by a primary race in `x`. This race condition has been found by Intel Inspector when the variable `len` is assigned a value of 16.

Example 4.5 (False Positive in DawnCC)**.** DawnCC has parallelized a doubly nested loop containing the dependence `b[i][j] = b[i-1][j-1]`; hence, leading to a data race. Interestingly, the tool correctly avoids the parallelization when the matrix `b` is given in linear format, i.e., `b[i*n+j]`.

Example 4.6 (Different parallelization in ICC)**.** The variable `sum` in the loop `for (int i=0; i<100; i++) sum+=1;` was marked as `firstlastprivate` by ICC; however, that construction should be a reduction.

Example 4.7 (False Negative in AutoPar)**.** The same program seen in Figure 8 (b) gives us a false negative when submitted to AutoPar. This tool refuses to annotate this loop. However, arrays `a`, `b` and `c` are allocated statically; hence, it is trivial to show that aliasing cannot occur in this case.

4.4 Performance Comparison

The current distribution of AutoParBench has been designed to uncover bugs. However, AutoParBench includes benchmarks taken from the Rodinia and NPB suites, which can be used to evaluate the performance of compilers and hardware. AutoParBench provides a driver to execute these programs. We have used this framework to compare the speed of the code produced by two different auto-parallelization tools: ICC and AutoPar. Figure 9 shows the result of this comparison for six NPB benchmarks, and five Rodinia benchmarks that run when compiled with AutoPar, ICC-Full and ICC-Cost, namely `BFS`, `BPT`=B+Tree, , `E3C`=Euler3D, `E3C`= Euler3D (Double), and `HTW`=Heartwall.

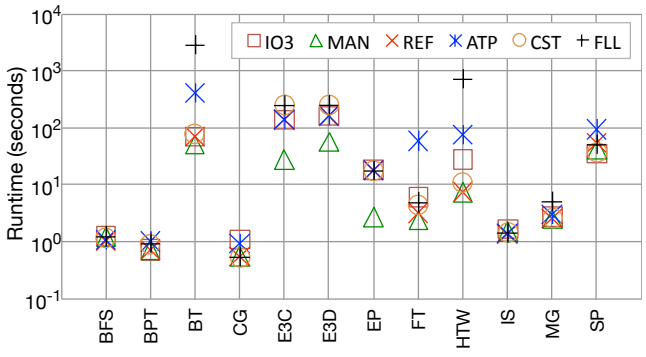

Figure 9. Performance comparison. We use the following keys: `IO3` = sequential code compiled with ICC -O3, `MAN` = manual OpenMP annotations in the original benchmarks, `REF` = the reference collection (every loop annotated), `ATP` = AutoPar, `CST` = ICC-Cost, `FLL` = ICC-Full.

The original benchmarks (`MAN`) have been annotated by their developers with OpenMP pragmas. This is generally the fastest code. In some cases, e.g., `E3C` and `EP`, `MAN` is over 5x faster than the fastest code automatically produced. The reference collection has not been conceived for performance: we have annotated every loop that is parallelizable. Nevertheless, except for NPB's `SP`, the reference is still faster than automatically annotated programs. ICC-Cost improves the runtime of ICC-Full, the unrestricted parallelizer, by using a cost model that rules out potentially unprofitable parallelizations. Such improvement can be dramatic: about 37x for Rodinia's `BPT`. There is no clear winner between AutoPar and ICC-Cost. The former yields statistically significant faster code in three cases; the latter in five. In every case, differences can be elastic: AutoPar's version of Rodinia's `E3C` is 1.8x faster; ICC-Cost's version of NPB's BT is 5.3x faster.

5 Related Work

Benchmarks for Compilers and Tools. The construction of benchmark suites has been a staple of compiler and tool development since its early years. Testimony of this importance is the fact that a few benchmark collections, namely from the SPEC CPU family [17], have been a fundamental driving force behind the design and implementation of static analyses and program optimizations for C, C++ and Fortran compilers, as thoroughly discussed in Patterson and Hennessy's classic textbook [16]. Similar role DaCapo has fulfilled for the Java programming Language and the JVM virtual machine [7]. And, even today, we watch the rise of new suites [15, 29], or the re-edition of old ones [20, 21] to fill niches not yet covered by well-established collections.

Support for the Development of Auto-Parallelizers. AutoParBench contains a reference collection of annotated programs that is similar to other benchmarks for parallel programming. Among those, we count Linpack [13], Rodinia [10], Parsec [4] (and its task-based extensions [9]), NAS [5, 28], SHOC [12] and BOTS [14]. There are also frameworks to evaluate the performance of auto-parallelizers, such as PETRA [24]; however, automatic evaluation is centered on runtime. Recently, Prema *et al.* [26] have pointed out the need for supporting comparisons oriented towards correctness, like the one AutoParBench provides.

Several programs in the reference collection were taken from public suites, namely NAS and Rodinia. The main difference between the present work, and these previous benchmarks is the fact that we provide a program representation that unifies the methodology used to test and verify compilers and tools related to automatic parallelization of programs. All our infrastructure, including programs and their harnesses, have been designed around this intermediate representation. Said representation is built as a meta-language on top of the JSON format. Thus, in contrast to previous benchmark suites, AutoParBench lets us compare different tools using the same framework.

DataRaceBench. A closely related benchmark related to the present paper is DataRaceBench [18]. The goal of DataRanceBench is to test data-race detection tools; hence, this suite contains samples without or with known data-races. it only needs to check a tool's output against a simple true or false reference answer for a given OpenMP input loop. AutoParBench uses OpenMP benchmarks from DataRaceBench and other benchmark suites. However,

the objective of AutoParBench is to provide a unified representation to the output of parallelizers. These tools output programs that, although syntactically different, can be correct parallelizations of the original input code. Thus, AutoParBench uses an intermediate representation to unify the testing process. Such intermediate representation is not part of the design of DataRaceBench.

6 Conclusion

This paper has presented AutoParBench, a framework that allows semantics-aware, quantitative comparison of the output of different automatic parallelization tools. AutoParBench is engineered around a unified representation of OpenMP-based parallel programs, in the JSON format. A suite of supporting translators plus an evaluator are developed to compare programs produced by auto-parallelizers and by humans. We have evaluated AutoParBench by applying it onto four parallelizers. AutoParBench has allowed us to discover several bugs in these tools, many of which were acknowledged by their developers. Several more warnings are still left to be confirmed. As future work, we plan to augment AutoParBench's reference collection with more benchmarks, including SPEC OMP and SPEC ACCEL. We also intend to add to AutoParBench's intermediate representation the ability to encode OpenMP-based Task Directives.

7 Acknowledgement

This work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344 (LLNL-CONF-795158). Work on the design of the test framework was supported by the U.S. DOE Advanced Scientific Computing Program (ASCR SC-21). Experiments were funded through the LLNL-LDRD Program (Project No. 18-ERD-006). Fernando Pereira is sponsored by CNPq (Grant 406377/2018-9) and FAPEMIG (Grant PPM-00193-16). While at UFMG, Gleison Mendonça has received a scholarship from CAPES, under the Brazilian Ministry of Education. We thank the ICS anonymous referees for the time and expertise that they have put into reviewing this paper.

References

[1] Péricles Alves, Fabian Gruber, Johannes Doerfert, Alexandros Lamprineas, Tobias Grosser, Fabrice Rastello, and Fernando Magno Quintão Pereira. 2015. Runtime Pointer Disambiguation. In *OOPSLA* (Pittsburgh, PA, USA). ACM, New York, NY, USA, 589–606. https://doi.org/10.1145/2814270.2814285

[2] Eduard Ayguadé, Rosa M. Badia, Pieter Bellens, Daniel Cabrera, Alejandro Duran, Roger Ferrer, Marc González, Francisco D. Igual, Daniel Jiménez-González, and Jesús Labarta. 2010. Extending OpenMP to Survive the Heterogeneous Multi-Core Era. *International Journal of Parallel Programming* 38, 5-6 (2010), 440–459. https://doi.org/10.1007/s10766-010-0135-4

[3] Hansang Bae, Dheya Mustafa, Jae-Woo Lee, Aurangzeb, Hao Lin, Chirag Dave, Rudolf Eigenmann, and Samuel P. Midkiff. 2013. The Cetus Source-to-Source Compiler Infrastructure: Overview and Evaluation. *Int. J. Parallel Programming* 41, 6 (2013), 753–767. https://doi.org/10.1007/s10766-012-0211-z

[4] Rajive Bagrodia, Richard Meyer, Mineo Takai, Yu-an Chen, Xiang Zeng, Jay Martin, and Ha Yoon Song. 1998. Parsec: A Parallel Simulation Environment for Complex Systems. *Computer* 31, 10 (1998), 77–85. https://doi.org/10.1109/2.722293

[5] D.H. Bailey, E. Barszcz, J.T. Barton, D.S. Browning, R.L. Carter, L. Dagum, R.A. Fatoohi, P.O. Frederickson, T.A. Lasinski, R.S. Schreiber, H.D. Simon, V. Venkatakrishnan, and S.K. Weeratunga. 1991. The NAS Parallel Benchmarks. *Int. J. High Perform. Comput. Appl.* 5, 3 (1991), 63–73. https://doi.org/10.1177/109434209100500306

[6] Jairo Balart, Alejandro Duran, Eduard Gonzalez, Xavier Martorell, Eduard Ayguade, and Jesus Labarta. 2004. Nanos Mercurium: a Research Compiler for OpenMP. In *EWOMP*. IEEE, New York, NY, USA, 103–109.

[7] Stephen M. Blackburn, R. Garner, Chris Hoffmann, Asjad M. Khang, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. E. B. Moss, A. Phansalkar, D. Stefanović, T. VanDrunen, D. von Dincklage, and B. Wiedermann. 2006. The DaCapo Benchmarks: Java Benchmarking Development and Analysis. In *OOPSLA* (Portland, Oregon, USA). ACM, NY, USA, 169–190.

[8] Uday Bondhugula, Albert Hartono, J. Ramanujam, and P. Sadayappan. 2008. A Practical Automatic Polyhedral Parallelizer and Locality Optimizer. In *PLDI* (Tucson, AZ, USA). ACM, New York, NY, USA, 101–113. https://doi.org/10.1145/1375581.1375595

[9] Dimitrios Chasapis, Marc Casas, Miquel Moretó, Raul Vidal, Eduard Ayguadé, Jesús Labarta, and Mateo Valero. 2015. PARSECSs: Evaluating the Impact of Task Parallelism in the PARSEC Benchmark Suite. *ACM Trans. Archit. Code Optim.* 12, 4 (2015), 1–. https://doi.org/10.1145/2829952

[10] Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W. Sheaffer, Sang-Ha Lee, and Kevin Skadron. 2009. Rodinia: A Benchmark Suite for Heterogeneous Computing. In *IISWC*. IEEE, Washington, DC, USA, 44–54. https://doi.org/10.1109/IISWC.2009.5306797

[11] L. Dagum and R. Menon. 1998. OpenMP: An Industry-Standard API for Shared-Memory Programming. *Comput. Sci. Eng.* 5, 1 (1998), 46–55. https://doi.org/10.1109/99.660313

[12] Anthony Danalis, Gabriel Marin, Collin McCurdy, Jeremy S. Meredith, Philip C. Roth, Kyle Spafford, Vinod Tipparaju, and Jeffrey S. Vetter. 2010. The Scalable Heterogeneous Computing (SHOC) Benchmark Suite. In *GPGPU-3*. ACM, New York, NY, USA, 63–74. https://doi.org/10.1145/1735688.1735702

[13] Jack Dongarra. 1988. The LINPACK Benchmark: An Explanation. In *SC*. Springer-Verlag, London, UK, UK, 456–474. http://dl.acm.org/citation.cfm?id=647970.742568

[14] Alejandro Duran, Xavier Teruel, Roger Ferrer, Xavier Martorell, and Eduard Ayguade. 2009. Barcelona OpenMP Tasks Suite: A Set of Benchmarks Targeting the Exploitation of Task Parallelism in OpenMP. In *ICPP*. IEEE, Washington, DC, USA, 124–131. https://doi.org/10.1109/ICPP.2009.64

[15] Breno C F Guimarães, José Wesley de S Magalhães, Anderson Faustino da Silva, and Fernando M Q Pereira. 2019. Synthesis of Benchmarks for the C Programming Language by Mining Software Repositories. In *SBLP*. ACM, New York, NY, USA, 62–69. https://doi.org/10.1145/3355378.3355380

[16] John L. Hennessy and David A. Patterson. 1990. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, San Francisco, CA, USA.

[17] John L. Henning. 2006. SPEC CPU2006 Benchmark Descriptions. *SIGARCH Comput. Archit. News* 34, 4 (Sept. 2006), 1–17. https://doi.org/10.1145/1186736.1186737

[18] Chunhua Liao, Pei-Hung Lin, Joshua Asplund, Markus Schordan, and Ian Karlin. 2017. DataRaceBench: A Benchmark Suite for Systematic Evaluation of Data Race Detection Tools. In *SC*. ACM, New York, NY, USA, 11:1–11:14. https://doi.org/10.1145/3126908.3126958

[19] Chunhua Liao, Daniel J Quinlan, Jeremiah J Willcock, and Thomas Panas. 2010. Semantic-aware automatic parallelization of modern applications using high-level abstractions. *Int. J. Parallel Programming* 38, 5 (2010), 361–378.

[20] Anderson M. Maliszewski, Dalvan Griebler, Claudio Schepke, Alexander Ditter, Dietmar Fey, and Luiz Gustavo Fernandes. 2018. The NAS Benchmark Kernels for Single and Multi-Tenant Cloud Instances with LXC/KVM. In *HPCS*. IEEE, Los Alamitos, CA, USA, 359–366.

[21] Carlos A. F. Maron, Adriano Vogel, Dalvan Griebler, and Luiz Gustavo Fernandes. 2019. Should PARSEC Benchmarks be More Parametric? A Case Study with Dedup. In *PDP*. IEEE, Los Alamitos, CA, USA, 217–221.

[22] Gleison Souza Diniz Mendonca, Breno Campos Ferreira Guimarães, Péricles Rafael Oliveira Alves, Fernando Magno Quintão Pereira, Marcio Machado Pereira, and Guido Araujo. 2016. Automatic Insertion of Copy Annotation in Data-Parallel Programs. In *SBAC-PAD*. IEEE, Los Alamitos, CA, USA, 34–41. https://doi.org/10.1109/SBAC-PAD.2016.13

[23] Gleison Mendonça, Breno Guimarães, Péricles Alves, Márcio Pereira, Guido Araújo, and Fernando Magno Quintão Pereira. 2017. DawnCC: Automatic Annotation for Data Parallelism and Offloading. *ACM Trans. Archit. Code Optim.* 14, 2, Article 13 (May 2017), 25 pages. https://doi.org/10.1145/3084540

[24] Dheya Mustafa and Rudolf Eigenmann. 2015. PETRA: Performance Evaluation Tool for Modern Parallelizing Compilers. *Int. J. Parallel Program.* 43, 4 (2015), 549–571. https://doi.org/10.1007/s10766-014-0307-8

[25] C. Polychronopoulos. 1987. Loop coalescing: A compiler transformation for parallel machines. In *ICPP*. OSTI, Washington, DC, USA, 235–242.

[26] S. Prema, Rupesh Nasre, R. Jehadeesan, and B. K. Panigrahi. 2019. A study on popular auto-parallelization frameworks. *Concurrency and Computation: Practice and Experience* 31, 17 (2019), 1–. https://doi.org/10.1002/cpe.5168

[27] Pedro Ramos, Gleison Souza, Divino Soares, Guido Araújo, and Fernando Magno Quintão Pereira. 2018. Automatic Annotation of Tasks in Structured Code. In *PACT*. ACM, New York, NY, USA, 31:1–31:13. https://doi.org/10.1145/3243176.3243200

[28] Sangmin Seo, Gangwon Jo, and Jaejin Lee. 2011. Performance characterization of the NAS Parallel Benchmarks in OpenCL. In *IISWC*. IEEE Press, Piscataway, NJ, USA, 137–148. https://doi.org/10.1109/IISWC.2011.6114174

[29] Shinichi Shiraishi, Veena Mohan, and Hemalatha Marimuthu. 2015. Test suites for benchmarks of static analysis tools. In *ISSREW*. IEEE, New York, USA, 12–15.

[30] Michael Joseph Wolfe. 1995. *High Performance Compilers for Parallel Computing*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.