UNIVERSIDADE FEDERAL DE MINAS GERAIS
Instituto de Ciências Exatas
Programa de Pós-Graduação em Ciência da Computação

Gleison Souza Diniz Mendonça

AutoParBench: Uma ferramenta para verificação de código paralelo

Belo Horizonte
2020-03

Gleison Souza Diniz Mendonça

AutoParBench: Uma ferramenta para verificação de código paralelo

Versão Final

Dissertação apresentada ao Programa de Pós-Graduação em Ciência da Computação da Universidade Federal de Minas Gerais, como requisito parcial à obtenção do título de Mestre em Ciência da Computação.

Orientador: Fernando Magno Quintão Pereira
Coorientador: Chunhua Liao

Belo Horizonte
2020-03

Gleison Souza Diniz Mendonça

AutoParBench: A Framework for Parallel Code verification

Final Version

Thesis presented to the Graduate Program in Computer Science of the Federal University of Minas Gerais in partial fulfillment of the requirements for the degree of Master in Computer Science.

Advisor: Fernando Magno Quintão Pereira
Co-Advisor: Chunhua Liao

Belo Horizonte
2020-03

Ficha catalográfica elaborada pela bibliotecária Belkiz Inez Rezende Costa
CRB 6ª Região nº 1510

Mendonça,Gleison Souza Diniz.

M539a AutoParBench: a framework for parallel code verification / Gleison Souza Diniz Mendonça. — Belo Horizonte, 2020.
xxvi, 51 f. il.; 29 cm.

Dissertação (mestrado) - Universidade Federal de Minas Gerais – Departamento de Ciência da Computação
Orientador: Fernando Magno Quintão Pereira.
Coorientador: Chunhua Liao

1. Computação – Teses. 2. Benchmarking. 3. Paralelização. 4.Software – Verificação. I. Orientador. II. Coorientador. III. Título.

CDU 519.6*32 (043)

UNIVERSIDADE FEDERAL DE MINAS GERAIS
INSTITUTO DE CIÊNCIAS EXATAS
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

FOLHA DE APROVAÇÃO

AUTOPARBENCH: UMA FERRAMENTA PARA VERIFICAÇÃO DE CÓDIGO PARALELO

GLEISON SOUZA DINIZ MENDONÇA

Dissertação defendida e aprovada pela banca examinadora constituída pelos Senhores:

[illegible]
Prof. Fernando Magno Quintão Pereira - Orientador
Departamento de Ciência da Computação - UFMG

Liaochunhua
Doutor Liao Chunhua - Coorientador
Lawrence Livermore National Laboratory - CASC

[illegible]
Prof. Dorgival Olavo Guedes Neto
Departamento de Ciência da Computação - UFMG

[illegible]
Doutor Pei-Hung Lin
Lawrence Livermore National Laboratory - CASC

Belo Horizonte, 28 de Fevereiro de 2020.

Dedico este trabalho a meus pais, que nunca me abandonaram ante dificuldades.

Acknowledgments

Agradeço aos meus pais, Vânia Helena dos Reis e Hervê Pereira Mendonça por todo o apoio e dedicação ao longo destes anos. Agradeço ao meu orientador Fernando Magno, que me possibilitou construir algo que creio ser relevante para as comunidades de programação paralela e compiladores. Agradeço ao meu mentor Liao Chunhua pela oportunidade de estágio. Agradeço ao meu ex orientador Thiago Ferreira de Noronha e aos meus colegas da maratona de programação, Marcos Almeida e Felipe Arcanjo. Também gostaria de lembrar dos meus colegas de laboratório Péricles Alves, Breno Guimarães, Guilherme Leobas, Késia Andrade, Pedro Ramos, Junio César, Carina Capelão, Henrique Nazaré, Tarsila Bessa e José Wesley, que sempre se mostraram solícitos. Por fim, um agradecimento especial aos professores e todo o corpo docente da Universidade Federal de Minas Gerais, que me possibilitaram melhorar minhas habilidades profissionais e pessoais ao longo de tantos anos.

"*Mais vale um péssimo acordo do que um grande confronto.*"
(Autor desconhecido)

Resumo

Atualmente, existem muitas ferramentas de paralelização que realizam inserção automática de diretivas OpenMP em programas. No entanto, é um desafio comparar automaticamente essas ferramentas quanto a seus pontos fortes e limitações, devido às diversas opções de paralelização de um mesmo programa. Visando suprir esta carência, fornecemos o AutoParBench, uma estrutura de teste que visa mitigar este problema. O AutoParBench consiste em benchmarks e um verificador. Atualmente, os benchmarks incluem 99 programas com 1.579 loops. Um procedimento é definido para permitir a adição rápida e fácil de novos programas. O verificador consiste em uma representação intermediária comum, baseada em JSON, além de todo o mecanismo necessário para converter programas OpenMP em um formato doravante denominado objeto JSON. Os objetos produzidos por diferentes ferramentas permitem a comparação semântica automáticamente de resultados de paralelização sintaticamente diferentes. O AutoParBench é um instrumento eficaz para encontrar bugs. Ao investigar as diferenças nos objetos produzidos por fontes separadas, ou seja, ferramenta versus ferramenta ou ferramenta versus ser humano, relatamos bugs em ferramentas de paralelização selecionadas, como ICC, Cetus, AutoPar e DawnCC. Esses bugs foram todos confirmados.

Benchmark, Parallelization, Tools, VerificationBenchmark, Parallelization, Tools, Verification

Palavras-chave: Testes de referência, Parallelização, Ferramentas, Verificação

Abstract

Abstract

There exist presently many parallelization tools based on the automatic insertion of OpenMP pragmas into programs. However, it is challenging to automatically and quantitatively compare these tools for their strengths and limitations, due to the diverse choices to parallelize a program. This work describes AutoParBench, a test framework aimed to mitigate this problem. AutoParBench consists of benchmarks and a verifier. Benchmarks currently include 99 programs with 1,579 loops.

A procedure is defined to allow quick and easy additions of new programs. The verifier consists of a common intermediate representation, based on JSON, plus all the machinery necessary to convert OpenMP programs into a format henceforth called a JSON snapshot. The snapshots produced by different tools enable automatic semantics-aware comparison of syntactically different parallelization results. AutoParBench is an effective bug-finding instrument. By investigating differences in snapshots produced by separate sources, i.e., tool versus tool or tool versus human, we have reported bugs in selected parallelization tools such as ICC, Cetus, AutoPar and DawnCC, all of which have been confirmed.

Keywords: Benchmark, Parallelization, Tools, Verification

List of Figures

List of Tables

Contents

Chapter 1

Introduction

The growing prominence of OpenMP [12] has contributed to the appearance of many automatic parallelization tools. OpenMP works as a set of annotations embedded into a host language, such as C, C++ or Fortran. By annotating programs with OpenMP directives, said tools are able to generate parallel code without having to deal with minutiae of algorithmic parallelization or details of the target architecture. Examples of automatic parallelizers based on OpenMP include Intel Compilers (ICC), DawnCC [26], AutoPar [23], Mercurium [6], Pluto [8], TaskMiner [32] 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 3, such is not the case. Presently, the only way to compare the output of these tools is via manual inspection. Visual comparison is a hard task: each parallelizer produces code that, even when semantically equivalent, can use very different syntax. Furthermore, tools like ICC or Pluto are able to change the layout of the target program, to enable further parallelization. Additionally, not every tool is source-to-source. ICC, for instance, generates binary code. Developers must read its logs to understand the transformations performed by the compiler.

Our Thesis. We claim that it is possible to design a common representation that normalizes OpenMP directives; hence, allowing direct and automatic comparison between the output of different parallelization tools.

Our Contributions. To support our thesis, we have designed such a representation to enable the verification of OpenMP 4.5 annotations across different tools. Centered around this normalized representation, described in Section 4, we built a test framework called *AutoParBench*. This framework contains, as of today, 103 programs, for a total of 1,594 loops. Some of them were taken from well-known benchmark suites, such as NAS [5] and Rodinia [11]. A procedure is defined to allow quick and easy additions of new programs. AutoParBench also provides harnesses to run the benchmarks and check their outputs. A parser is provided to generate a JSON object out of a C/C++ program with OpenMP directives. This file is called a *JSON snapshot*. To compare the output of two tools, or the output of a tool with the reference collection, it suffices to compare two automatically

generated snapshots. The reference collection comes with positive and negative examples. The former are loops that can be parallelized; the latter are loops that should not be parallelized due to known data dependencies. This separation is helpful to point out bugs and inaccuracies in automatic parallelizers. The combination of benchmarks, parser and unified representation lets us perform semantics-aware comparison of thousands of parallelization results in seconds; thus, generating metrics such as precision, recall and accuracy organized into graphic reports. At the best of our knowledge, AutoPark Bench is the first benchmark suite designed to correctness checking, making its focus a unique contribution to ensure the quality of parallel software.
Our Results. The direct benefit of AutoParBench is bug discovery. We have used this framework to evaluate four different automatic parallelization tools: The Intel Compiler (ICC), DawnCC, Cetus and AutoPar. AutoParBench lets us compare them all, providing a final, unified score for each tool, that gives the user some perspective on their practicality and reach. Using this infrastructure, we have gotten hundreds of warnings and some confirmed bugs in different tools: 2 in DawnCC, 2 in Cetus, 4 in AutoPar and 2 in ICC. These bugs have been acknowledged as true problems, as we report in Section 5. AutoParBench is publicly available at `https://github.com/gleisonsdm/AutoParBench`.

1.1 Publications

Several papers have been published in the context of this work. Some of these papers have inspired this dissertation, whereas at least one stemmed directly from it. Below we provide a list of publications that I have co-authored during my stay at the Compilers Lab:

- Gleison Mendonça, Breno Guimarães, Péricles Rafael Oliveira Alves, Márcio Pereira, Guido Araújo and Fernando Magno Quintão Pereira. *DawnCC: Automatic Annotation for Data Parallelism and Offloading.* ACM Transactions on Architecture and Code Optimization. 14 (2), July 2017.

- Pedro Ramos, Gleison Mendonça, Divino Soares, Guido Araújo and Fernando Magno Quintão Pereira. *Automatic Annotation of Tasks in Structured Code.* PACT. 2018.

- Pedro Ramos, Gleison Souza Diniz Mendonça, Guilherme V. Leobas and Fernando Magno Quintão Pereira. *Taskminer: Automatic identification of tasks*, SBLP, pages 11-18, 2018.

- Gleison Souza Diniz Mendonça, Breno Campos Ferreira Guimarães, Péricles Rafael Oliveira Alves, Márcio Machado Pereira, Guido Araújo and Fernando Magno Quintão Pereira. *Automatic Insertion of Copy Annotation in Data-Parallel Programs.* SBAC. 2016.

- Kézia Correa Andrade Moreira, Gleison Souza Diniz Mendonça, Breno Campos Ferreira Guimarães and Fernando Magno Quintão Pereira. *Paralelização Automática de Codigo com Diretivas OpenACC.* XX Simpósio Brasileiro de Linguagens de Programação. Maringá, Brasil, 2016.

- Breno Campos Ferreira Guimarães, Gleison Souza Diniz Mendonça and Fernando Magno Quintão Pereira. *DawnCC: a Source-to-Source Automatic Parallelizer of C and C++ Programs.* CBSoft Tools. Maringá, Brasil, 2016.

- Douglas Couto, Kézia Andrade, Gleison Souza, and Fernando Pereira. *Etino: Colocação Automática de Computação em Hardware Heterogêneo.* Simpósio Brasileiro de Linguagens de Programação, Belo Horizonte, Brasil, 2015.

- Douglas Couto, Kezia Andrade, Gleison Souza, and Fernando Pereira. *Etino: Colocação Automática de Computação em Hardware Heterogêneo.* CBSoft Tools, pages 65-72, Belo Horizonte, Brasil, 2015.

In addition to this paper, together with Fernando Pereira, my MSc advisor, and Chunhua Liao, my MSc co-advisor, I have written and submitted for evaluation the following work:

- *AutoParBench: A Unified Test Framework for OpenMP-based Parallelizers*

This paper summarizes the findings that this dissertation reports in a greater level of detail.

Chapter 2

Literature Review

Nowadays, there exists much work aimed at the evaluation of parallel programs. Optimizations like parallelization and vectorization are commonly used in this domain to improve the performance of computing systems. Unfortunately, such optimizations may compromise the correctness of programs, if implemented incorrectly. This possibility motivates the design and implementation of benchmark suites to evaluate the effects of automatic parallelization tools. In this chapter, we discuss some of the state-of-the-art benchmark suites that exist in different research fields related to high-performance computing.

2.1 Benchmarks in General

The process of testing software is based on the use of samples representing program inputs. In the context of this work, the software under test is some automatic parallelization tool, and the samples used as inputs are programs that must be parallelized. Relevant features can be verified using a reference collection, comparing the results and checking if they are as expected for a given sample. There are hundreds of benchmarks designed to assess the behavior of algorithms in a given context. These benchmarks can be categorized as Synthetic or Real World and grouped as System-Level or Component-Level.

Synthetic benchmarks are formed by code snippets extracted from other benchmark suites. These samples are usually created combining functions from different applications. This kind of technique can be used to increase the number of benchmarks available. The synthetic references allow the developer to change data structures, providing a more proper evaluation for programs.

In the paper *Fast System Level Benchmarks for Multicore Architectures*, Sen et al. [34] present a framework to generate system-level synthetic benchmarks. These synthetic programs can be used for evaluation when the source code is not available. The artificial samples are smaller and faster than the original programs. However, they approximate the dynamic behavior of real-world code. The System-Level test cases can, not only identify correctness issues within programs, but also reveal the impacts of optimizations on them.

In some cases, benchmarks should represent variety of scenarios that test some common aspect of a program. One example of a benchmark suite designed according to this guideline is HiBench [19]. That benchmark suite was designed to evaluate the implementation of MapReduce. It contains samples of synthetic micro-benchmarks and real-world applications, which constitute a comprehensive collection of functions. That framework includes the computation of throughput, system resources utilization and data access patterns.

Some benchmark suites are categorized as Real-World, because they are extracted from real use cases. In this category, we can cite BigDataBench [38], which was released to evaluate computer architectures using data that represents authentic use cases. The collection on that suite was originally composed of 19 benchmarks, selected for fairly evaluating big data systems.

Component-Level benchmarks focus on evaluating specific aspects instead of testing the behavior of the complete system. Huang et al. [20] describe a benchmark collection for mobile devices that has been shown to facilitate microarchitectural exploration. That collection, called Moby, contains different applications for testing components in domains as diverse as social networking and video streaming. The framework was designed to facilitate the use of full-system architectural simulators.

2.1.1 The role of a reference collection in compiler construction

The use of benchmarks for testing is a reliable way to check if the software works as expected. To ensure correctness and performance improvements, compiler writers often resort to a reference collection. That collection is formed by a number of test cases of known semantics. They are typically accepted as baselines for performance measurements, or correctness. By comparing against that reference collection, developers can track how much improvement has been achieved by a particular optimization or some new static code analysis. As an example, the LLVM compiler [21] provides developers with a reference collection with over 200 programs split into 30 different benchmark suites.

The reference collection can be used for performance verification on optimized programs. By running benchmarks with different input sizes, it is possible to measure how the modifications impact the quality of the code produced by the compiler. Reference collections also allow developers to evaluate characteristics of different architectures, as hardware may affect program execution. Therefore, reference collections guide research efforts, because they contribute to the definition of quality standards.

2.1.2 Some relevant benchmark suites

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 [18], have been a fundamental guiding 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 [28]. Similar role the DaCapo benchmark has fulfilled for the Java programming Language and the JVM virtual machine [7].

Compilers, being complex tools, demand different kinds of tests and verifications. FreeBench [33] and MallocBench [17] are indicated for memory evaluation. BitBench [14] is a collection originally containing six key algorithms that can operate on streaming input data. And, even today, we watch the rise of new suites [31, 36], or the re-edition of old ones [24, 25] to fill niches not yet covered by well-established collections.

2.2 Benchmarks for parallel computing

The design of parallel algorithms is an error-prone task, which demands careful verification due to its complexity. In this context, parallel benchmarks may contain features that are designed to catch side effects of parallelization strategies. In this section we revisit some benchmarks typically used to test parallelization methodologies.

2.2.1 The most important benchmarks

Rodinia [11] is a benchmark suite designed for heterogeneous computing. It provides developers with different programs that implement the same core algorithm. For instance, there are versions of Rodinia that run sequentially, or in GPUs, or in multi-threaded CPUs. That collection provides developers with the means to evaluate program characteristics such as power consumption and synchronization.

It is hard to measure costs for the development and design of large systems. Different alternatives can be considered to pick the best one available, which makes it required to examine systems efficiently. Parsec [4] is a framework designed to help managing simulations in parallel. Bagrodia et al. [4] have presented a parallel simulation language called Parsec. That programing language incorporates a GUI and a runtime system.

Some parallel algorithms exploit intensive data processing in supercomputers. That is a domain that demands very specific reference collections. The NAS [5] suite is an example of benchmark developed for that domain. NAS emulates typical scenarios of data movement and batch computation that are common in supercomputing applications.

Independently of the size of the program, different parallelization strategies can be used by programmers and companies. Such strategies might be based on regular or irregular applications. In the latter category we count recursive function calls, for instance. Programmers can easily exploit irregular parallelism using interfaces like OpenMP or OpenACC. Those systems of annotations facilitate the creation of parallel programs. BOTS [16] is a benchmark suite that contains samples of the OpenMP tasking model, released in the OpenMP (3.0) specification. It was projected to test multicore architectures. Programs in that collection are often recursive, or contain loops that iterate over irregular data-structures.

2.3 Testing in general

Nowadays, there are many options to evaluate applications. Measure the effectiveness of optimizations is essential to avoid issues in algorithms, which makes necessary a collection with samples that can represent real uses. Testing tends to reveal program limitations that should receive attention. These constraints are analyzed by developers, following test reports. From these reports, developers can propose improvements to programs.

2.3.1 How test is used to find bugs in compilers

Testing provides a methodology for correctness checking in compilers. Due to its complexity, that kind of software is prone to generate code with issues. And, contrary to most programs, errors in compilers will surface in secondary code, i.e., the generated program, not in the compiler itself. In this scenario, good reference collections play a fundamental role.

Perhaps the most well-known tool to test compilers is Csmith [40], which generates random C programs. Those programs, once fed to two or more compilers, indicate errors if different compilers produce executables that behave differently. That tool can generate a large subset of programs that represent different patterns from real applications.

2.4 Testing for parallelization tools

This work describes a test framework to compare automatic parallelization tools. There are many relevant parallel benchmarks. Among those, we include Linpack [15], Rodinia [11], Parsec [4] (and its task-based extensions [10]), NAS [5, 35], SHOC [13] and BOTS [16]. There are also frameworks to evaluate the performance of auto-parallelizers, such as PETRA [27]; however, automatic evaluation is centered on runtime. Recently, Prema *et al.* [30] have pointed out the need for supporting comparisons oriented towards correctness, like the one AutoParBench provides.

Several programs distributed with AutoParBench — the focus of this work — were taken from public suites, namely NAS and Rodinia. The main difference between the present work, and those previous benchmark collections is the fact that we provide a program representation that unifies the methodology used to test and verify automatic parallelization compilers and tools. All our infrastructure, including programs and their harnesses, have been designed around that 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, as long as these tools contain annotated reference programs.

2.4.1 DataRaceBench

DataRaceBench ([22]) is a benchmark suite designed at Lawrence Livermore National Laboratory. It was desiged to evaluate the effectiveness of data race detection tools; hence, it contains samples without and with known data-races. The framework only needs to check a tool's output against a simple true or false reference answer for a given OpenMP input loop.

Given its goals and scope, DataRaceBench [22] is closely related to this work. However, AutoParBench was designed to produce a unified representation of the code generated by compilers. These tools output programs that, although syntactically different, can represent correct parallelizations of the original input code. DataRaceBench did not have a focus on comparing different tools. Rather, it was a collection of programs with known semantics meant to detect bugs in auto-parallelizers.

2.4.2 Other kinds of tests for parallelization tools

There exist different ways to design tests for parallelized programs, which provides options to ensure correctness. NDSet ([9]) introduces to the sequential version of the program a form of “controlled nondeterminism“ which emulates different outputs possible if the program runs in parallel. Its insight is recognizing common patterns of writing non-deterministic sequential specifications. It explores a few Java benchmarks comparing the sequential and parallel versions depending on non-deterministic sequential requirements. NDSet’s authors have reported a significant reduction in the number of false positives facing traditional checking when searching for parallelization bugs.

A different verification strategy can explore model checking to verify parallel programs. BlobFlow [37] presents an application that resorts to such methodology. BlobFlow’s authors use formal verification to validate a code snippet, checking for deadlock freedom or functional equivalence to the available versions.

Chapter 3

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.

3.1 A program may be amenable to different parallelization strategies.

A program may be amenable to different parallelization strategies.

There exist different parallel patterns, e.g., data and task parallelism. Targets can also vary, e.g., CPUs and GPUs. As an example, Listing 3.1 shows how different targets lead to different parallelizations of the same program.

Listing 3.1: Loop parallelization using two different strategies: vectorization or GPU acceleration.

```
void CPU_vectorization(int *a, int len) {
  #pragma omp simd
  for (int i=0; i<len; i++)
    a[i]= i;
}
void GPU_parallelization(int *a, int len) {
  #pragma omp target map(from:a[0:len])
  #pragma omp teams distribute parallel for
  for (int i=0; i<len; i++)
    a[i]= i;
}
```

The functions in Listing 3.1 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 4.1.2.

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

Pointer aliasing might hinder parallelization due to potential dependencies. Dependences 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 dependences at runtime. As an example, Listing 3.2 shows code produced by DawnCC.

Listing 3.2: 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 on `ovrlp`, at line 4, only allows parallel execution when the two arrays cover disjoint memory regions. Thus, the program in Listing 3.2 is correct, as long as the guard is present. Section 4.2.1 explains how we evaluate multi-versioned loops.

3.3 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 3.3 illustrates this issue with an example. In Section 4.2.1 we explain how we represent loops at different granularities, and in Section 4.1.3 we discuss how to deal with unprofitable parallelization.

Listing 3.3: Nested loops can be parallelized in different ways.

```
void both_parallel (int **a, int len) {
  int i, j;
  #pragma omp parallel for private(j)
  for (i=0; i< len; i++)
    #pragma omp parallel for simd
    for (j=0; j<len; j++)
      a[i][j] = (i * len + j + 0.5);
}
void outer_parallel (int **a, int len) {
  int i, j;
  #pragma omp parallel for private(j)
  for (i=0; i< len; i++)
    for (j=0; j<len; j++)
      a[i][j] = (i * len + j + 0.5);
}
```

3.4 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 3.4 illustrates this issue.

Listing 3.4: Data mapping variants for target directives

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

Function `target_loop` in Listing 3.4 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 clauses "`target data map`" and "`target parallel for`". A verification tool needs to match the semantics of these two uses of the `target` pragma, as we explain in Section 4.2.2.

3.5 Pragmas can use expressions parameterized by different program symbols.

Several OpenMP directives are parameterized by program symbols, i.e., user-defined names. As an example, the `map` and the `depends` pragmas receive an array and a memory range. Memory ranges are expressions that use program symbols. Such 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 3.5 illustrates this challenge.

Listing 3.5: 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]++;
  }
}
```

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 4.3.1 we explain how AutoParBench reports different parallelizations.

3.6 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 [39]. 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 3.6 illustrates this issue.

Listing 3.6: 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 program in Listing 3.6, the function `coalesced _loop` is produced by

ICC, via *loop coalescing* [29]. A common format that allows using function `coalesced_loop` as the reference output for the parallelization of function `original_loop` (Listing 3.6) must provide hooks to match these two different programs. In Section 4.3 we elaborate on how we deal with such transformations.

3.7 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 implements 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 3.7 shows an example of such report.

Listing 3.7: 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 3.7 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 4.3.

Chapter 4

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 and a harness that compares tools by normalizing their outputs via the IR. This section discusses each one of these parts.

4.1 The Reference Collection

The reference collection is a set of ready-to-use C/C++ benchmarks intended to serve as a ground-truth for automatic parallelization tools. Currently, this collection contains 89 micro-kernels, plus fourteen larger programs from Rodinia and NAS. The provenance of these benchmarks is detailed in Section 5.1. 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. 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. To ensure correctness, we run each annotated program with Intel Inspector[1].

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

4.1.1 Extending the Reference Collection

We have designed AutoParBench considering that community support might play a role into its development. 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.

Figure 4.1 provides a description of how to add new samples to AutoParBench. The process includes manual checking, and scripts to check the output. Step `C2` includes output comparison and Intel Inspector, using scripts to automatize such tasks avoiding unnecessary efforts.

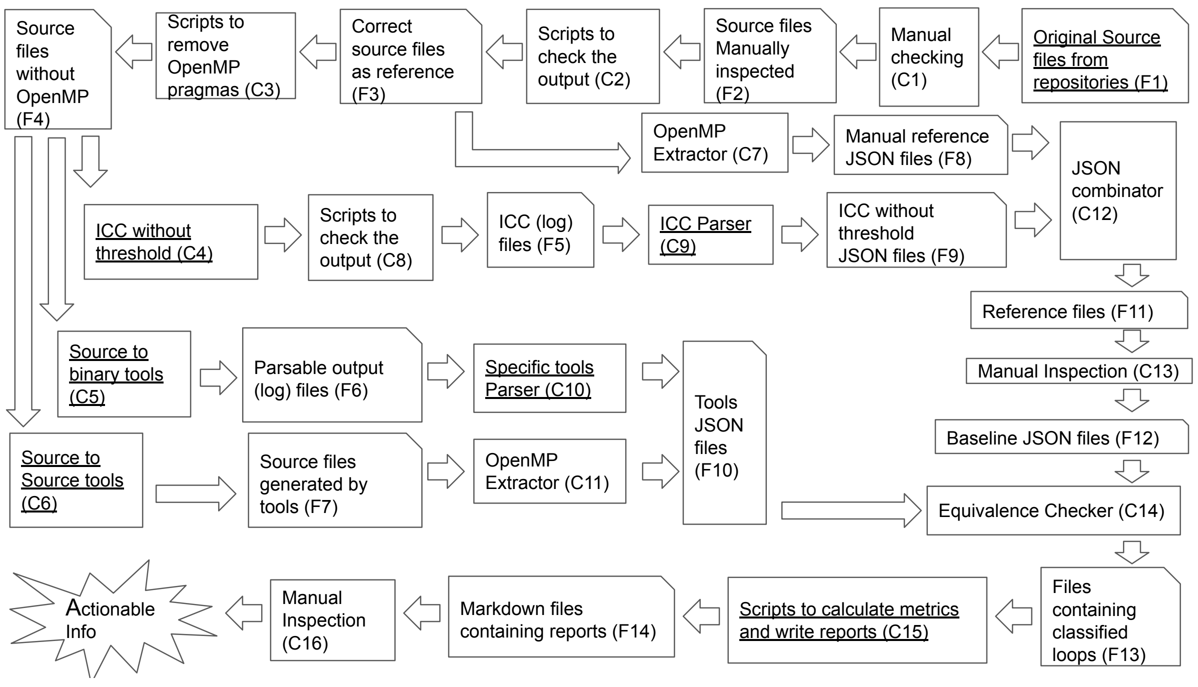

Source files without OpenMP (F4)
Scripts to remove OpenMP pragmas (C3)
Correct source files as reference (F3)
Scripts to check the output (C2)
Source files Manually inspected (F2)
Manual checking (C1)
Original Source files from repositories (F1)
OpenMP Extractor (C7)
Manual reference JSON files (F8)
JSON combinator (C12)
ICC without threshold (C4)
Scripts to check the output (C8)
ICC (log) files (F5)
ICC Parser (C9)
ICC without threshold JSON files (F9)
Reference files (F11)
Manual Inspection (C13)
Baseline JSON files (F12)
Equivalence Checker (C14)
Source to binary tools (C5)
Parsable output (log) files (F6)
Specific tools Parser (C10)
Tools JSON files (F10)
Source to Source tools (C6)
Source files generated by tools (F7)
OpenMP Extractor (C11)
Files containing classified loops (F13)
Scripts to calculate metrics and write reports (C15)
Markdown files containing reports (F14)
Manual Inspection (C16)
Actionable Info

Figure 4.1: Figure Explaining how to add new samples or compilers to AutoParBench.

Reports produced in `F14` include summarized reports, which make it easy to collect qualitative data grouping files with the same benchmark collection. Such info creates easy to evaluate samples based on relevant aspects, which permits the appropriate evaluation of each parallel strategy.

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 4.1: Categories of parallel strategies.

4.1.2 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 4.1. 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 provide a solution to Challenge 3.1.

4.1.3 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. 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.

4.2 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 some benchmark. 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 software that parses the output of a tool, and produces the corresponding JSON snapshots. As we shall explain in Section 4.3, 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 4.2 illustrates how snapshots are produced.

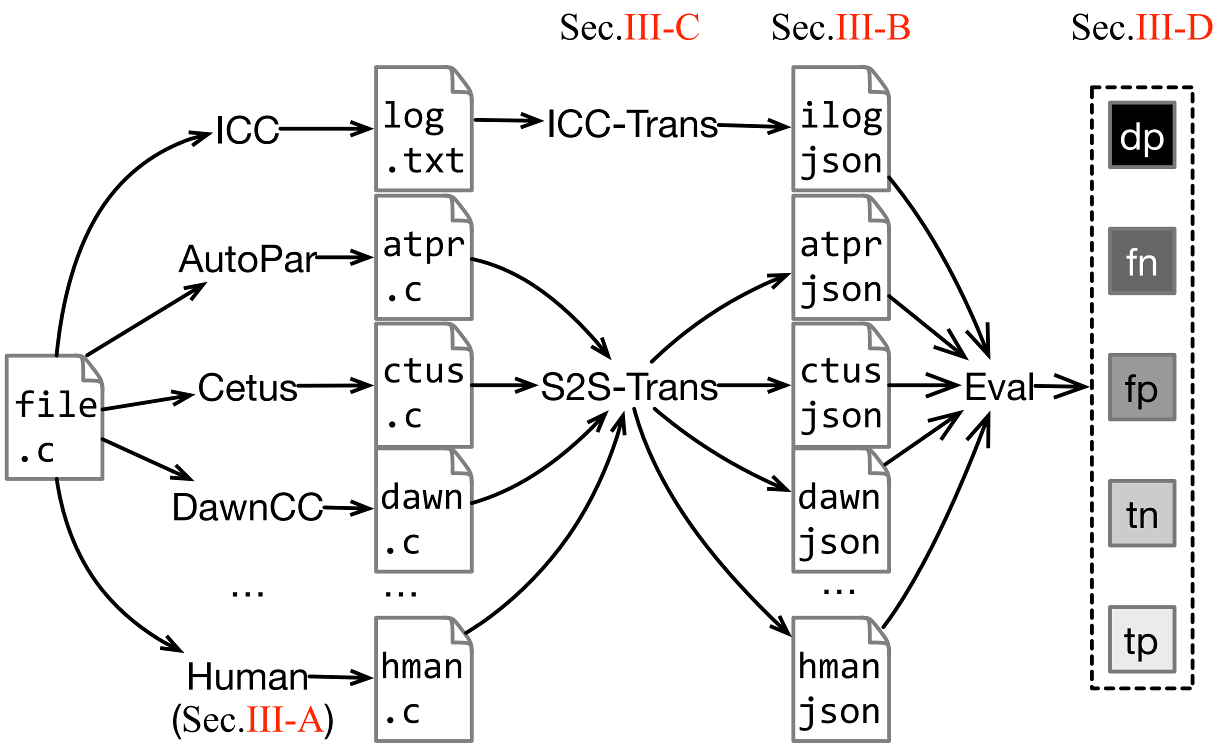
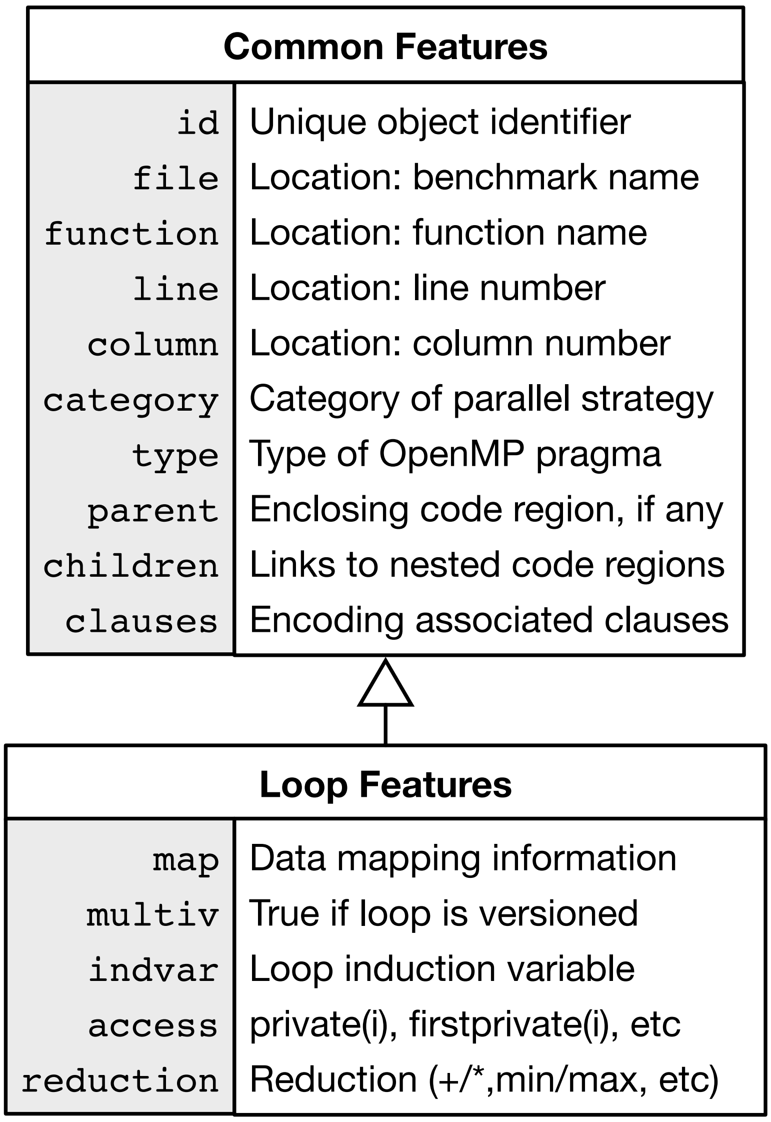


Figure 4.2: Production and evaluation of Snapshots for C/C++.

4.2.1 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*. Figure 4.3 shows the fields present in objects. We consider two subcategories of objects: those that represent loops, and those that represent code regions other than loops. The former category represents loops in the C/C++ programming languages: `while`, `do-while` and `for`. Loops formed by `goto` statements are not considered.



Common Features	
`id`	Unique object identifier
`file`	Location: benchmark name
`function`	Location: function name
`line`	Location: line number
`column`	Location: column number
`category`	Category of parallel strategy
`type`	Type of OpenMP pragma
`parent`	Enclosing code region, if any
`children`	Links to nested code regions
`clauses`	Encoding associated clauses

Loop Features	
`map`	Data mapping information
`multiv`	True if loop is versioned
`indvar`	Loop induction variable
`access`	private(i), firstprivate(i), etc
`reduction`	Reduction (+/*,min/max, etc)

Figure 4.3: Fields for JSON snapshot objects.

Example 1 *Figure 4.4 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 object is identical for both programs.*

Non-loop objects represent code regions that can be annotated with OpenMP pragmas, but that are not loops. Example 2 shows code that produces this kind of object. We have opted to separate loop and non-loop objects to allow comparing tools at a granularity smaller or larger than loop blocks. This strategy lets AutoParBench's

```
void main() {
  int *a = (int*)malloc(400);
  #pragma omp parallel
  #pragma omp for
  for (int i=0; i<100; i++) {
    a[i] = 1;
  }
}
```

(a)

```
void main() {
  int *a = (int*)malloc(400);
  #pragma omp parallel for
  for (int i=0; i<100; i++) {
    a[i] = 1;
  }
}
```

(b)

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

(c)

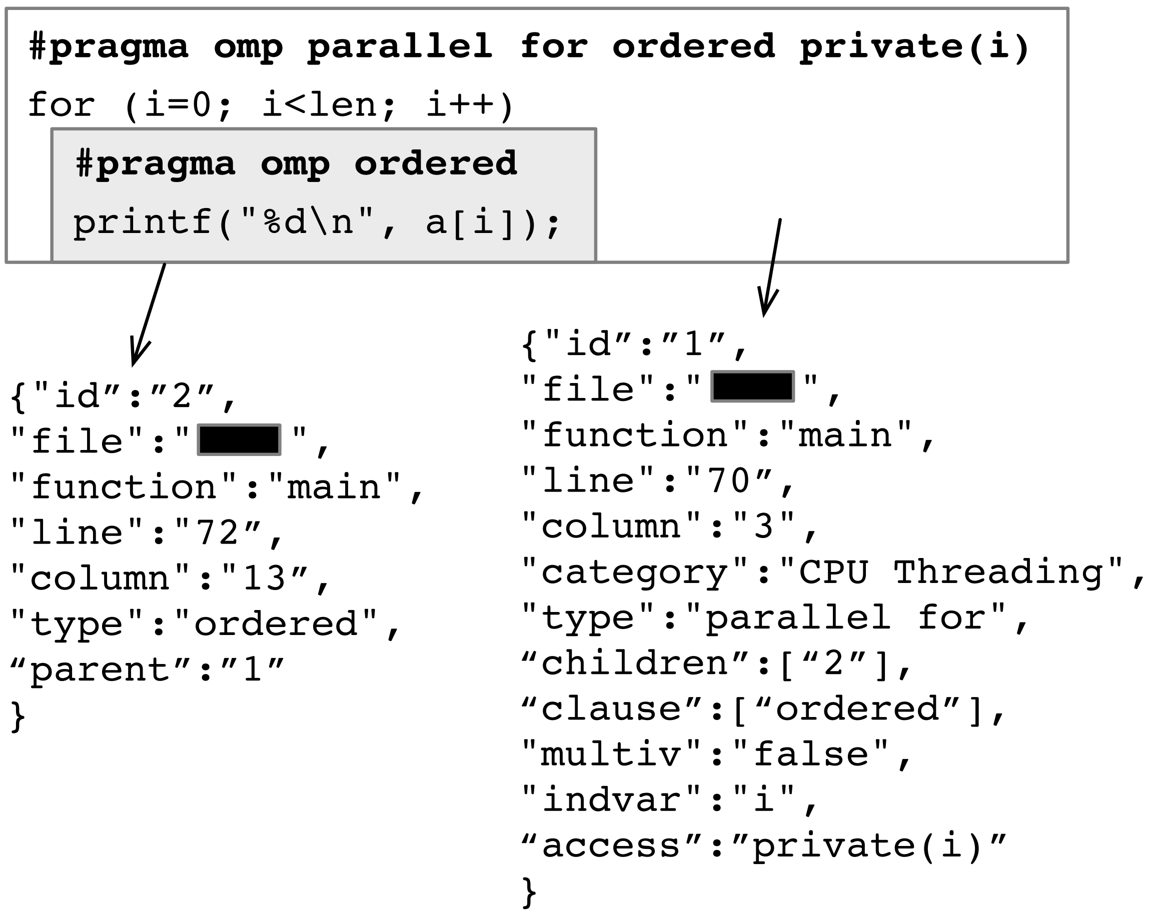
Figure 4.4: (a-b) Two semantically equivalent parallelizations of the same program. (c) The corresponding loop object.

evaluator pinpoint the parts of a parallel loop that are identical across tools, and the parts that differ.

Example 2 *Figure 4.5 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` *pragma, not shown in this example, is a second source of non-loop objects.*

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 3.2, 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 4.3.1) compares only the parallel version of a multi-versioned loop. This approach provides us with a pragmatic solution to Challenge 3.2.



```
#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 4.5: JSON snapshots extracted from a C program representing a loop and a non-loop object.

4.2.2 Semantic Equivalences

JSON snapshots enable the normalization of syntactically different codes into semantically equivalent classes of annotations. The need for such normalization stems 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 1, or, similarly, “`omp target data`“ and “`omp target`“ give “`omp target data`“. This helps AutoParBench deal with Challenge 3.4.
- 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 3.4.

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 becomes an individual field within the JSON file.

4.3 The Translators

Translators are used 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, since it does not annotate source code; instead, it produces parallel binary code. Additionally, when used in debugging mode, ICC produces a textual report. Our ICC translator parses this textual report, and produces the JSON snapshot out of it. The translators collectively help address Challenge 3.7.

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 pragmas. Furthermore, the only task annotator that we are aware of is TaskMiner [32], still a research artifact limited to small programs.

Dealing with Loop Transformations. AutoParBench also deals with Challenge 3.6 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 report, 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.

4.3.1 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 such results are relative to a *reference*. That reference can be the reference collection of Section 4.1, or it can be the output of another automatic parallelizer.

- *True Positive (TP)*: parallelized code generated by a tool 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 $= 2 \times \mathit{precision} \times \mathit{recall}/(\mathit{precision}+\mathit{recall})$. JSON objects in the “different parallelization“ and “Crash“ categories are not included in these metrics. Warnings reported as different parallelization give us a means to continually evolve AutoParBench. By investigating those warnings, we can refine them further as either true positives or wrong parallelizations.

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 3.5. 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 3.5.

4.4 Configurable Scripts and Reports

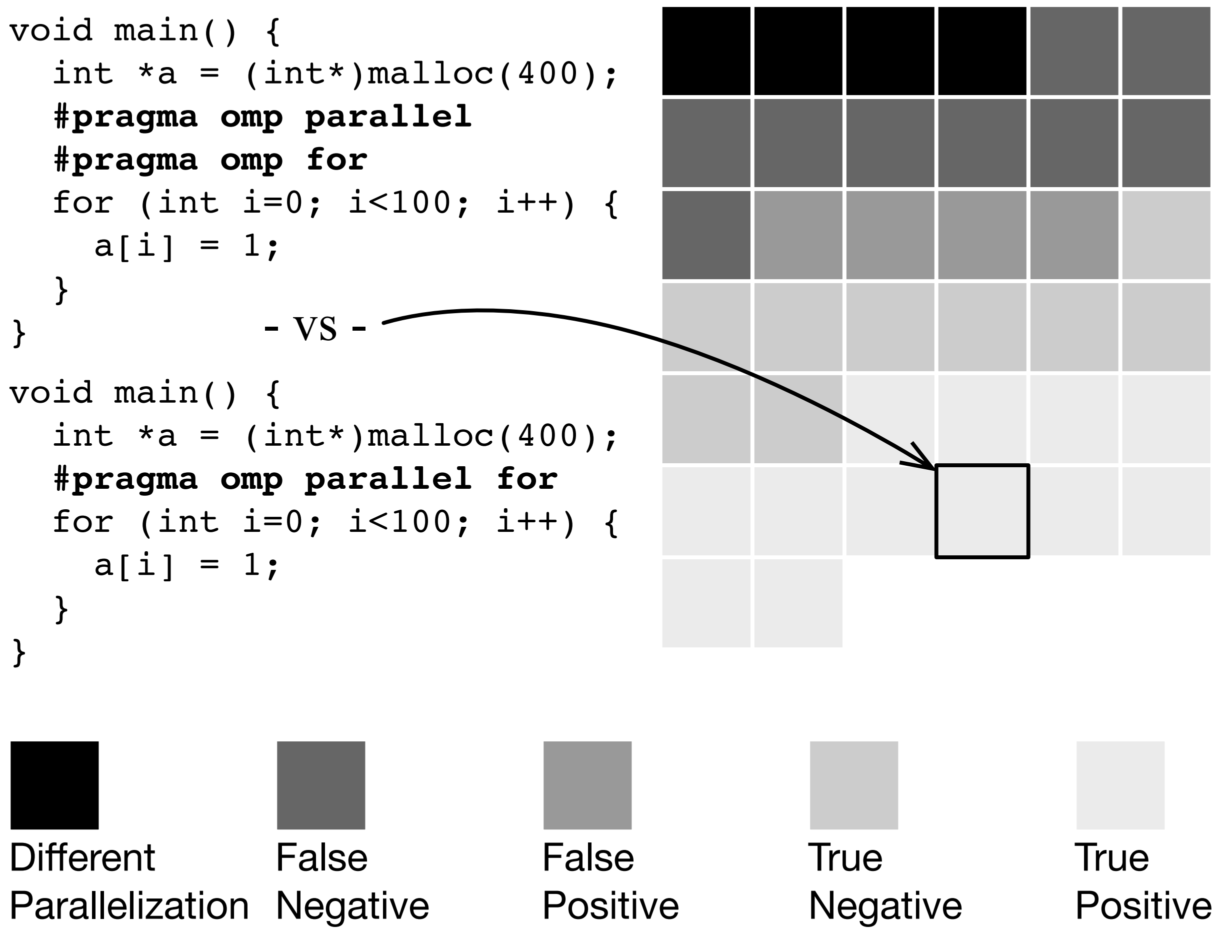
AutoParBench provides a collection of configurable scripts to evaluate auto-parallelizers. The baseline of evaluation is configurable, because it is possible to use the output of a tool as ground-truth. In other words, although AutoParBench provides a reference collection, which we use as the ground-truth 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 are the outputs produced by different tools.

Example 3 *Figure 4.6 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 colors. Similar reports can be generated for any pair of parallelizers, by treating one of them as the reference.*

4.5 Summary of the Solutions

This section provides an overview of the solution proposed in this manuscript.

- The use of different parallelization strategies can deal with the versions generated by different tools. Challenge 3.1 shows why it is necessary to categorize the reference collection, as explained in the Solution 4.1.2
- Loops can be replicated by compilers, which generates multiple versions of the same target source code snippet, as exposed on Challenge 3.2. This framework was designed to compare only the parallel version of the multi-versioned loop.



```
void main() {
  int *a = (int*)malloc(400);
  #pragma omp parallel
  #pragma omp for
  for (int i=0; i<100; i++) {
    a[i] = 1;
  }
}
```

```
void main() {
  int *a = (int*)malloc(400);
  #pragma omp parallel for
  for (int i=0; i<100; i++) {
    a[i] = 1;
  }
}
```

Figure 4.6: A graphical report produced by AutoParBench. Program on top was produced by auto-parallelizer; program on the bottom is part of the reference collection.

- Code can be parallelized considering profitability, which means it is necessary to consider nested parallel regions as described in Challenge 3.3. To figure out this problem, AutoParBench uses a best-effort solution, as explained in Solution 4.1.3.

- Challenge 3.4 expose the necessity of managed multiple versions of data mapping variants for accelerator offloading. To solve this issue, Solution 4.2.2 explains how to produce a standard format, making possible to verify equivalent constructions.

- Different symbolic expressions in programs may mean the same range of data, which produces an undecidable problem to evaluate their equivalence. Challenge 3.5 exposes the impacts of such a normalization problem. AutoParBench reports a category named “different parallelization“ whenever the auto-parallelizer differs from the reference collection, requiring manual inspection.

- Loops may be transformed due to optimizations applied in the source code. AutoParBench also deals with Challenge 3.6 during this translation step, it identifies the loops that have been modified, producing a snapshot for each loop that has been eliminated.

- Compilers can produce outputs in different formats, Challenge 3.7. To solve that, the framework uses different parsers to process files and extract normalized reports into a JSON snapshot. This solution is explained in Section refsub:extract.

Chapter 5

Experiments

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

- provide overall statistics about the benchmarks included in AutoParBench in Section 5.1;
- compare the output produced by different parallelizers in Section 5.2;
- demonstrate that AutoParBench is able to detect bugs in both research and industry tools in Section 5.3;
- carry out a performance comparison between different parallelization approaches in Section 5.4.

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 reported in this section 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.

5.1 The Framework

Provenance. Currently, AutoParBench's reference collection contains 89 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 4.1). Negative examples are loops which should not be parallelized due to data dependencies.

Size of Benchmarks. Figure 5.1 (top) groups benchmarks per lines 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 5.1 (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.

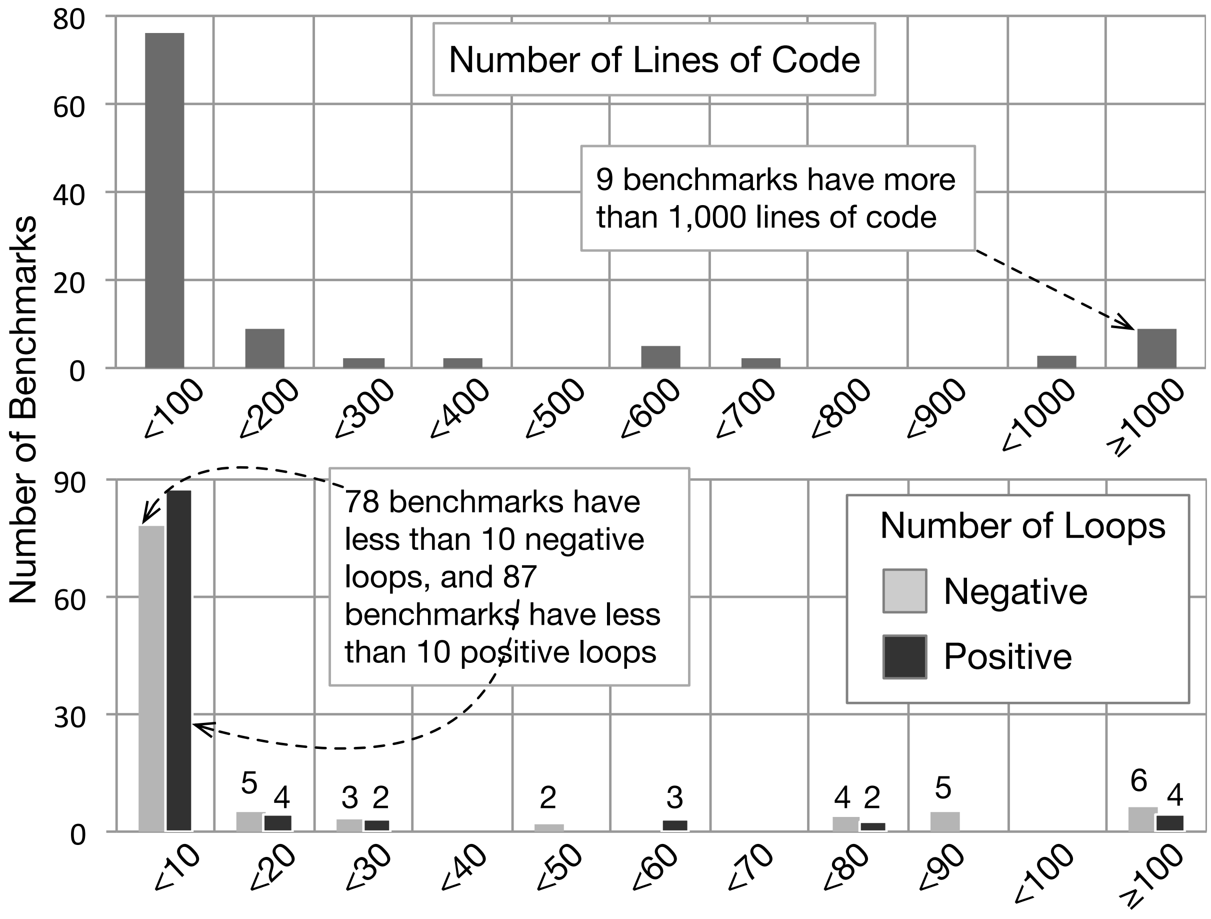


Figure 5.1: Lines of code and loops per benchmarks.

5.2 Comparing the Output of Tools

5.2.1 Standard Metrics.

To generate the metrics defined in Section 4.3.1, we compare the four auto-parallelizers with the reference collection. Table 5.1 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 assigned to each tool, and how many loops were analyzed (Lp). The number of loops analyzed per tool differs in Table 5.1 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	0.11
ICC Simd	1.00	0.59	0.65	0.74	17/63/63	0.74

Table 5.1: Summary of results. Pr = number of programs; Lp = number of loops handled; Tt = total of loops given to the tool; WF1 = Weighted F1-Score. Higher is better.

Table 5.1 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 -Restrictifier=true -Memory-Coalescing=true -Ptr-licm=true -Ptr-region=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 -vec-threshold0 -fno-inline -parallel -qopt-report-phase=all -qopt-report=5

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 Tab.4.1). ICC-Simd uses as reference the category *CPU SIMD*, and DawnCC uses *GPU SIMD*. These categories comprise 63 loops from 17 programs.

The last column of Table 5.1, WF1, is a *weighted* F1-score. This number, for a given tool, is defined as $WF1 = F1 \times P/U$, where U is the total of loops given to the tool, and P is the number of programs that the tool was able to handle. The weighted score gives us a measure of how close the output of a tool is from the reference collection.

Example 4 (Weighted F1-Score) *Although Cetus and ICC-Full receive the same universe 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 programs that it received. 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 indicative 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 5.1) is a better indicative of potential for bugs, as it takes the number of false positives in consideration. As we shall see in Section 5.3, false positives mark loops that are likely to be faulty.

Example 5 *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).*

5.2.2 Metrics for Benchmark suites.

This section discusses the results seen in Table 5.1. We shall split that table per benchmark; hence, showing individual results for DataRaceBench, Rodinia and NAS — the three suites that constitute AutoParBench.

Table 5.2 shows the results for each compiler when parallelizing DataRaceBench's benchmarks. Autopar cannot analyze all of them, because it breaks in some of the benchmarks. In particular, it adds OpenMP directives outside a loop which contains a `return` statement — an action invalid due to the OpenMP specification.

Table 5.3 shows metrics for the Rodinia collection. ICC's cost model has prevented this tool to parallelize all the loops in the Rodinia programs. Because some of these loops are known to be profitable [26], we speculate that the cost model used by ICC still has room for further tuning. Again, AutoPar could not analyze every loop in the target collection, although the ratio of failures is considerably smaller than that observed in Table 5.2.

Tool	Prec.	Rec.	Acc.	F1	Pr/Lp/Tt	WF1
Autopar	0.98	0.93	0.95	0.96	89/269/473	0.57
ICC Cost	01.00	0.18	0.54	0.31	89/473/473	0.31
ICC Full	0.96	0.71	0.83	0.82	89/473/473	0.82
Cetus	0.97	0.96	0.96	0.97	89/236/473	0.50

Table 5.2: Summary of results for DataRaceBench. Pr = number of programs; Lp = number of loops handled; Tt = total of loops given to the tool; WF1 = Weighted F1-Score. The higher, the better.

Tool	Prec.	Rec.	Acc.	F1	Pr/Lp/Tt	WF1
Autopar	0.18	0.28	0.84	0.22	6/178/219	0.18
ICC Cost	00.00	00.00	0.91	0.00	6/219/219	0.00
ICC Full	0.37	0.78	0.87	0.51	6/219/219	0.51
Cetus	0.33	0.66	0.97	0.44	6/171/219	0.34

Table 5.3: Summary of results for Rodinia. Pr = number of programs; Lp = number of loops handled; Tt = total of loops given to the tool; WF1 = Weighted F1-Score. The higher, the better.

Table 5.4 presents results for the NAS Parallel Benchmarks (NPB for short). This collection contains samples that represent classical parallel computations. Cetus is currently able to deal with a very small subset of the loops found in this benchmark suite. The issue seems to be motivated by a shortcoming in its implementation: Cetus does not support compiling multiple source files from different directories. ICC Full and AutoPar, in turn, perform exceedingly well in the NAS collection, analyzing correctly every loop that it contains.

Tool	Prec.	Rec.	Acc.	F1	Pr/Lp/Tt	WF1
Autopar	0.84	0.89	0.83	0.86	8/934/934	0.86
ICC Cost	0.89	0.32	0.57	0.47	8/934/934	0.47
ICC Full	0.94	0.88	0.90	0.91	8/934/934	0.91
Cetus	0.20	0.25	0.69	0.22	8/23/934	0.01

Table 5.4: Summary of results for NAS. Pr = number of programs; Lp = number of loops handled; Tt = total of loops given to the tool; WF1 = Weighted F1-Score. The higher, the better.

5.2.3 Graphical Comparison.

Figure 5.2 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 colors 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 colors in Figure 5.2. Thus, the darker the figure, the larger its potential to present bugs.

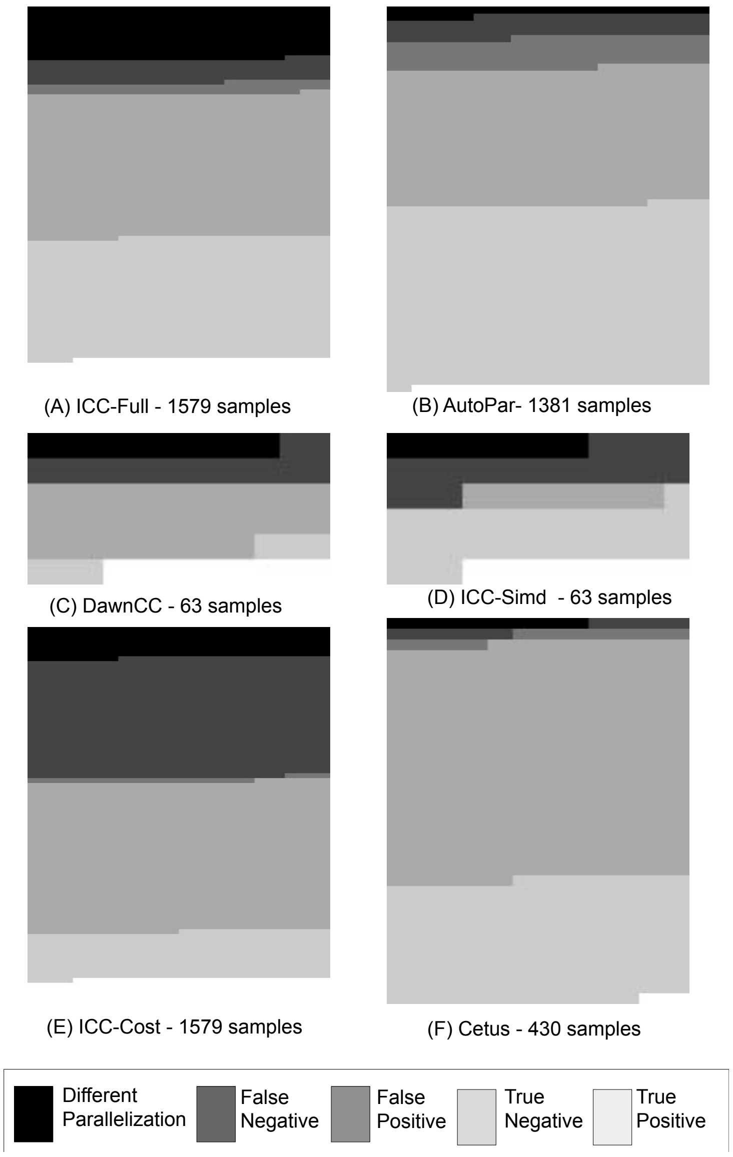


Figure 5.2: Graphical comparison between tools and baseline.

5.3 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.

5.3.1 A protocol to investigate results.

We have adopted a methodology to rank warnings. In our experience, said methodology improves 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 5.3.2); 3^{rd} tier: *false positive* reports (Examples 5.3.2 and 5.3.2); 4^{th} tier: reports of the *different parallelization* category (Example 5.3.2); and 5^{th} tier: *false negative* reports (Example 5.3.2).

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 5.5: Number of suspicious results grouped by tiers.

Table 5.5 shows the number of occurrences of suspicious results in the different tiers, considering the six experiments graphically reported in Figure 5.2. 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.

5.3.2 Examples of confirmed bugs.

Below we describe some of the confirmed bugs that we have reported.

Parallel program crashes

The process described in this work to find bugs could guide the experts to find an invalid generated code.

Listing 5.1: Sequential program that uncovered bug in Cetus.

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

Listing 5.2: Code produced by Cetus.

```
#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;
```

Cetus, when given the program in Listing 5.1, produces the code in Listing 5.2. Cetus extracts variable `j` from the loop, and transforms it into a reduction. Said reduction causes a runtime crash, the possible reason is that the variable `j` is not privatized with the initial value, and becomes shared. The compiler also is modifying the sequential portion of the code, replacing `c[j]` for `c[i+j]`. Listing 5.3 shows the same code present in the reference collection.

Listing 5.3: Code in reference collection.

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

False Positive in ICC

The framework has reported a false positive case when ICC parallelizes the first loop in the program in Listing 5.4.

Listing 5.4: Program that caused a false positive in ICC.

```c
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);
}
```

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 in a setup with `len==16`.

False Positive in DawnCC

DawnCC has parallelized a doubly nested loop with a combination of the directives `target parallel for` and `parallel for`, something that should not be used due to OpenMP specifications.

Listing 5.5: Code in reference collection.

```c
#include <stdio.h>
#include <stdlib.h>
int main(int argc, char *argv[]) {
  int i, j;
  int len = 20;

  double a[20][20];

  for (i = 0; i < len; i++)
    for (j = 0; j < len; j++)
      a[i][j] = (i * len + j + 0.5);
  return 0;
}
```

Listing 5.5 shows the sequential code that produces the sample present at Listing 5.6, wich contains the invalid OpenMP construction.

Listing 5.6: Code in reference collection.

```c
#include <stdio.h>
#include <stdlib.h>
int main(int argc, char *argv[]) {
  int i, j;
  int len = 20;

  double a[20][20];

  #pragma omp target data map(tofrom: a[0:20])
  {
  #pragma omp target parallel for
  for (i = 0; i < len; i++)
    #pragma omp parallel for
    for (j = 0; j < len; j++)
      a[i][j] = (i * len + j + 0.5);
  }
  return 0;
}
```

DawnCC missing private clause

DawnCC is parallelizing loops considering variables that should be private as shared (by default). In these cases, the compiler is introducing a race condition.

Listing 5.7: Code in reference collection.

```
#include <stdlib.h>
#include <stdio.h>
int main(int argc, char *argv[]) {
  int i;
  int tmp;
  int len = 100;
  int a[100];
  #pragma omp target data map(tofrom: a[0:100])
  {
  #pragma omp target parallel for
  for (i = 0; i < len; i++)
    a[i] = i;
}

  #pragma omp target data map(tofrom: a[0:100])
  {
  #pragma omp target parallel for
  for (i = 0; i < len; i++) {
    tmp = a[i] + i;
    a[i] = tmp;
  }
}

  printf("a[50]=%d\n", a[50]);
  return 0;
}
```

Listing 5.7 is the code generated by DawnCC, which inserts all OpenMP directives and clauses presented on it. A possible parallelization can be found at Listing 5.8.

Listing 5.8: Code in reference collection.

```
#include <stdlib.h>
#include <stdio.h>
int main(int argc, char* argv[])
{
  int i;
  int tmp;
  int len=100;
  int a[100];
  #pragma omp target data map(a[0:len])
  #pragma omp target parallel for
  for (i=0;i<len;i++)
    a[i]=i;

  #pragma omp target data map(a[0:len])
  #pragma omp target parallel for private(tmp)
  for (i=0;i<len;i++)
  {
    tmp =a[i]+i;
    a[i] = tmp;
  }

  printf("a[50]=%d\n", a[50]);
  return 0;
}
```

Different parallelization

The variable `sum` in the loop showed in Listing 5.9 was marked as `firstlastprivate` by ICC; however, that construction should be a reduction.

Listing 5.9: Code in reference collection.

```
#include <stdio.h>
int main (void)
{
  int sum=0;
  for (int i = 0; i < 100; i++)
  {
    sum += 1;
  }
  printf ("sum=\%d\n",sum);
  return 0;
}
```

The bug was acknowledged at https://software.intel.com/en-us/comment/1946142. It exposes that, sometimes, the reports from ICC are not accurate. However, these cases can expose an uncovered bug in the compiler, which provides a guide to the possible target region where the problem is coming from.

ICC crashing

When compiling the program in Listing 5.10, ICC produces code that crashes when run in parallel; however, the sequential version of the same program works correctly. The following command line was used to parallelize the program: `icc -w -par-threshold0 -no-vec -fno-inline -parallel -qopt-report-phase=all -qopt-report=3` This bug has been acknowledged at https://software.intel.com/pt-br/forums/intel-c-compiler/topic/822023.

Listing 5.10: Code in reference collection.

```
#include <stdlib.h>
#include <stdio.h>
int main(int argc, char* argv[])
{
  int i,j;
  int n=1000, m=1000;
  double b[1000][1000];

  for (i=0; i<n; i++)
    for (j=0; j<m; j++)
      b[i][j] = 0.5;

  for (i=1;i<n;i++)
    for (j=1;j<m;j++)
      b[i][j]=b[i-1][j-1];

  for (i=0;i<n;i++)
    for (j=0;j<m;j++)
      printf("b[\%d][\%d]=\%f\n", i, j, b[i][j]);
  return 0;
}
```

False Negative

The program seen in Listing 5.11 gives us a false negative report when submitted to AutoPar.

Listing 5.11: Sequential program that reports a false negative to Autopar.

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

Listing 5.12: Code in reference collection.

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

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. Listing 5.12 contains an example o a possible parallelization for this sample.

5.4 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 harness 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 5.3 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.

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.

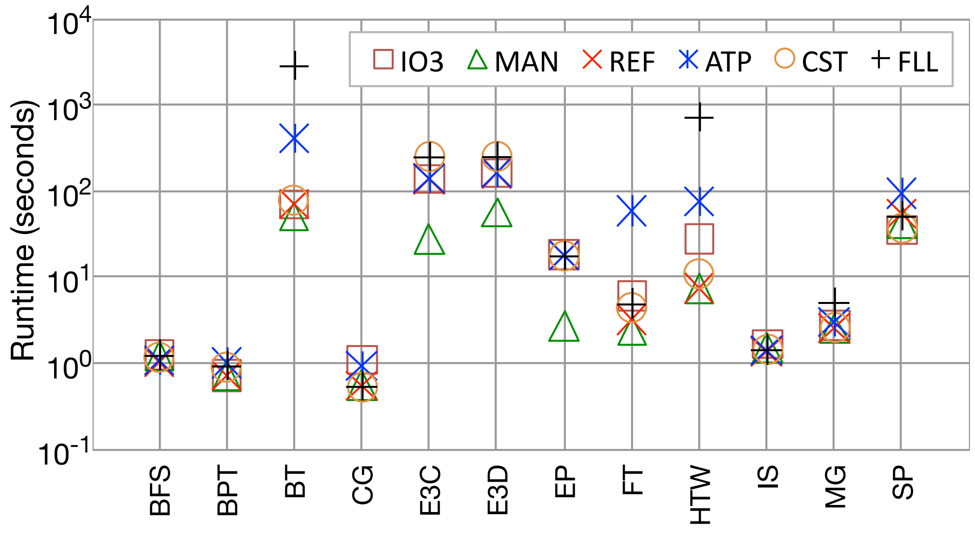


Figure 5.3: Performance comparison. We use the following keys: `IO3` = sequential code compiled with ICC -O3, `MAN` = manual OpenMP annotations in the original benchmarks, `REF` = the reference collection (every loop annotated), `ATP` = AutoPar, `CST` = ICC-Cost, `FLL` = ICC-Full.

Chapter 6

Conclusion

This dissertation 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 and evaluator are developed to enable semantics-aware comparison of 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 developers. Many more warnings are still left to be confirmed.

6.1 Future Work

As future work, we plan to augment AutoParBench's reference collection with more benchmarks, including SPEC OMP and SPEC ACCEL. AutoParBench has been conceived to facilitate the incorporation of new benchmarks into its reference collection. However, we believe that there exists still much room for improvements in the infrastructure necessary to receive new benchmarks. Said improvement consists of more tutorials and instructions about how to add new benchmarks, and the implementation of scripts and supporting tools that ease this task.

We also intend to add to AutoParBench's intermediate representation the ability to encode OpenMP-based Task Directives. Thus far, AutoParBench allows the representation of pragmas mostly related to the creation of data-parallel programs. However, there exist a vast ecosystem of task-parallel related benchmarks publicly available. This ecosystem has grown particularly in the most recent years, due to the addition of task primitives to the OpenMP standard. We believe that the incorporation of such directives into AutoParBench will be a valuable extension of it.

Bibliography

[1] Péricles Alves, Fabian Gruber, Johannes Doerfert, Alexandros Lamprineas, Tobias Grosser, Fabrice Rastello, and Fernando Magno Quintão Pereira. Runtime pointer disambiguation. In *OOPSLA*, pages 589–606, New York, NY, USA, 2015. ACM.

[2] Eduard Ayguadé, Rosa M Badia, Pieter Bellens, Daniel Cabrera, Alejandro Duran, Roger Ferrer, Marc González, Francisco Igual, Daniel Jiménez-González, Jesús Labarta, et al. Extending openmp to survive the heterogeneous multi-core era. *International Journal of Parallel Programming*, 38(5-6):440–459, 2010.

[3] Hansang Bae, Dheya Mustafa, Jae-Woo Lee, Aurangzeb, Hao Lin, Chirag Dave, Rudolf Eigenmann, and Samuel P. Midkiff. The cetus source-to-source compiler infrastructure: Overview and evaluation. *Int. J. Parallel Program.*, 41(6):753–767, 2013.

[4] Rajive Bagrodia, Richard Meyer, Mineo Takai, Yu-an Chen, Xiang Zeng, Jay Martin, and Ha Yoon Song. Parsec: A parallel simulation environment for complex systems. *Computer*, 31(10):77–85, 1998.

[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. The NAS parallel benchmarks. *Int. J. High Perform. Comput. Appl.*, 5(3):63–73, 1991.

[6] Jairo Balart, Alejandro Duran, Eduard Gonzalez, Xavier Martorell, Eduard Ayguade, and Jesus Labarta. Nanos mercurium: a research compiler for OpenMP. In *EWOMP*, pages 103–109. IEEE, 2004.

[7] Stephen M. Blackburn, Robin Garner, Chris Hoffmann, Asjad M. Khang, Kathryn S. McKinley, Rotem Bentzur, Amer Diwan, Daniel Feinberg, Daniel Frampton, Samuel Z. Guyer, Martin Hirzel, Antony Hosking, Maria Jump, Han Lee, J. Eliot B. Moss, Aashish Phansalkar, Darko Stefanović, Thomas VanDrunen, Daniel von Dincklage, and Ben Wiedermann. The DaCapo benchmarks: Java benchmarking development and analysis. In *OOPSLA*, pages 169–190, NY, USA, 2006. ACM.

[8] Uday Bondhugula, Albert Hartono, J. Ramanujam, and P. Sadayappan. A practical automatic polyhedral parallelizer and locality optimizer. In *PLDI*, pages 101–113, New York, NY, USA, 2008. ACM.

[9] Jacob Burnim, Tayfun Elmas, George Necula, and Koushik Sen. Ndseq: Runtime checking for nondeterministic sequential specifications of parallel correctness. *SIGPLAN Not.*, 46(6):401–414, June 2011.

[10] Dimitrios Chasapis, Marc Casas, Miquel Moretó, Raul Vidal, Eduard Ayguadé, Jesús Labarta, and Mateo Valero. PARSECSs: Evaluating the impact of task parallelism in the PARSEC benchmark suite. *ACM Trans. Archit. Code Optim.*, 12(4):1–, 2015.

[11] Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W. Sheaffer, Sang-Ha Lee, and Kevin Skadron. Rodinia: A benchmark suite for heterogeneous computing. In *IISWC*, pages 44–54, Washington, DC, USA, 2009. IEEE.

[12] L. Dagum and R. Menon. OpenMP: An industry-standard API for shared-memory programming. *Comput. Sci. Eng.*, 5(1):46–55, 1998.

[13] Anthony Danalis, Gabriel Marin, Collin McCurdy, Jeremy S. Meredith, Philip C. Roth, Kyle Spafford, Vinod Tipparaju, and Jeffrey S. Vetter. The scalable heterogeneous computing (shoc) benchmark suite. In *GPGPU-3*, pages 63–74, New York, NY, USA, 2010. ACM.

[14] Kyle Daruwalla, Heng Zhuo, Carly Schulz, and Mikko Lipasti. Bitbench: A benchmark for bitstream computing. In *Proceedings of the 20th ACM SIGPLAN/SIGBED International Conference on Languages, Compilers, and Tools for Embedded Systems*, pages 177–187, 2019.

[15] Jack Dongarra. The linpack benchmark: An explanation. In *International Conference on Supercomputing*, pages 456–474, London, UK, UK, 1988. Springer-Verlag.

[16] Alejandro Duran, Xavier Teruel, Roger Ferrer, Xavier Martorell, and Eduard Ayguade. Barcelona OpenMP tasks suite: A set of benchmarks targeting the exploitation of task parallelism in OpenMP. In *ICPP*, pages 124–131, Washington, DC, USA, 2009. IEEE.

[17] Dirk Grunwald, Benjamin Zorn, and Robert Henderson. Improving the cache locality of memory allocation. In *Proceedings of the ACM SIGPLAN 1993 conference on Programming language design and implementation*, pages 177–186, 1993.

[18] John L. Henning. SPEC CPU2006 benchmark descriptions. *SIGARCH Comput. Archit. News*, 34(4):1–17, September 2006.

[19] Shengsheng Huang, Jie Huang, Jinquan Dai, Tao Xie, and Bo Huang. The hibench benchmark suite: Characterization of the mapreduce-based data analysis. In *2010 IEEE 26th International Conference on Data Engineering Workshops (ICDEW 2010)*, pages 41–51. IEEE, 2010.

[20] Yongbing Huang, Zhongbin Zha, Mingyu Chen, and Lixin Zhang. Moby: A mobile benchmark suite for architectural simulators. *2014 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 45–54, 2014.

[21] Chris Lattner and Vikram Adve. Llvm: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization*, CGO '04, pages 75–, Washington, DC, USA, 2004. IEEE Computer Society.

[22] 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 *SC*, pages 11:1–11:14, New York, NY, USA, 2017. ACM.

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

[24] Anderson M. Maliszewski, Dalvan Griebler, Claudio Schepke, Alexander Ditter, Dietmar Fey, and Luiz Gustavo Fernandes. The NAS benchmark kernels for single and multi-tenant cloud instances with LXC/KVM. In *HPCS*, pages 359–366, Los Alamitos, CA, USA, 2018. IEEE Computer Society Press.

[25] Carlos A. F. Maron, Adriano Vogel, Dalvan Griebler, and Luiz Gustavo Fernandes. Should PARSEC benchmarks be more parametric? A case study with dedup. In *PDP*, pages 217–221, Los Alamitos, CA, USA, 2019. IEEE Computer Society Press.

[26] Gleison Mendonça, Breno Guimarães, Péricles Alves, Márcio Pereira, Guido Araújo, and Fernando Magno Quintão Pereira. DawnCC: Automatic annotation for data parallelism and offloading. *ACM Trans. Archit. Code Optim.*, 14(2):13:1–13:25, May 2017.

[27] Dheya Mustafa and Rudolf Eigenmann. PETRA: Performance evaluation tool for modern parallelizing compilers. *Int. J. Parallel Program.*, 43(4):549–571, 2015.

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

[29] C. Polychronopoulos. Loop coalescing: A compiler transformation for parallel machines. In *International Conference on Parallel Processing*, pages 235–242, Washington, DC, USA, 1987. OSTI.

[30] S. Prema, Rupesh Nasre, R. Jehadeesan, and B. K. Panigrahi. A study on popular auto-parallelization frameworks. *Concurrency and Computation: Practice and Experience*, 31(17), 2019.

[31] Aleksandar Prokopec, Andrea Rosà, David Leopoldseder, Gilles Duboscq, Petr Tůma, Martin Studener, Lubomír Bulej, Yudi Zheng, Alex Villazón, Doug Simon, Thomas Würthinger, and Walter Binder. Renaissance: Benchmarking suite for parallel applications on the jvm. In *PLDI*, pages 31–47, New York, NY, USA, 2019. ACM.

[32] Pedro Ramos, Gleison Souza, Divino Soares, Guido Araújo, and Fernando Magno Quintão Pereira. Automatic annotation of tasks in structured code. In *PACT*, pages 31:1–31:13, New York, NY, USA, 2018. ACM.

[33] Peter Rundberg and Fredrik Warg. The freebench v1. 0 benchmark suite. *URL: http://www. freebench. org*, 2002.

[34] Alper Sen, Gokcehan Kara, Etem Deniz, and Smaïl Niar. Fast system level benchmarks for multicore architectures. *2014 17th Euromicro Conference on Digital System Design*, pages 635–638, 2014.

[35] Sangmin Seo, Gangwon Jo, and Jaejin Lee. Performance characterization of the NAS parallel benchmarks in opencl. In *IISWC*, pages 137–148, Piscataway, NJ, USA, 2011. IEEE Press.

[36] Shinichi Shiraishi, Veena Mohan, and Hemalatha Marimuthu. Test suites for benchmarks of static analysis tools. In *2015 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*, pages 12–15. IEEE, 2015.

[37] Stephen F Siegel and Louis F Rossi. Analyzing blobflow: A case study using model checking to verify parallel scientific software. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, pages 274–282. Springer, 2008.

[38] Lei Wang, Jianfeng Zhan, Chunjie Luo, Yuqing Zhu, Qiang Yang, Yongqiang He, Wanling Gao, Zhen Jia, Yingjie Shi, Shujie Zhang, Chen Zheng, Gang Lu, Kent Zhan, Xiaona Li, and Bizhu Qiu. Bigdatabench: A big data benchmark suite from internet services. *2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA)*, pages 488–499, 2014.

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

[40] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. Finding and understanding bugs in c compilers. *SIGPLAN Not.*, 46(6):283–294, June 2011.