AutoParBench: A Unified Test Framework for OpenMP-based Parallelizers

Gleison Souza Diniz Mendonça[1], Fernando Magno Quintão Pereira[1], Chunhua Liao[2]

[1]Departamento de Ciência da Computação
Universidade Federal de Minas Gerais (UFMG) – Brazil

[2]Lawrence Livermore National Laboratory – CASC – USA

{gleison.mendonca,fernando}@dcc.ufmg.br,liao6@llnl.gov

Abstract. *After decades of advances in techniques of automatic parallelization, software developers can count, today, on many different tools that transform a program, so that it runs in parallel. Yet, most of these tools are still considered research artifacts. They contain latent bugs, often consequence of a sparsity of testing frameworks for autoparallelizers. This dissertation describes one such framework: AutoParBench—the product of a cooperation between UFMG's Compilers Lab, and Lawrence Livermore National Laboratory. AutoParBench is today publicly available, and has been successfully used to find three zero-day bugs in the Intel C Compiler. Its usage also uncovered problems in more research-oriented tools: 2 bugs in DawnCC, 4 in Rose AutoPar and 2 in Cetus. All these bugs have been confirmed, and some of them have been already fixed as an aftermath of this work.*

1. Introduction

Much effort has been spent in the craft of automatic parallelization tools. These tools adapt programs to run in parallel. Examples of automatic parallelizers include the Intel C Compiler (ICC), DawnCC [Mendonca et al. 2016, Mendonça et al. 2017], AutoPar [Liao et al. 2010], Pluto [Bondhugula et al. 2008], TaskMiner [Ramos et al. 2018b] and Cetus [Bae et al. 2013]. Yet, the usage of these tools is not widespread. Developers are reluctant to employ them, mostly because it is difficult to find bugs in these artifacts [Liao et al. 2017]. The goal of this dissertation was to mitigate these difficulties. To this end, we have delivered two contributions.

The First Contribution: DawnCC. The first of these contributions was DawnCC [Mendonca et al. 2016, Mendonça et al. 2017], an automatic parallelization tool that annotates programs with OpenMP pragmas. Programs thus annotated can be compiled to graphics processing units, an embarrassingly parallel hardware. Innovations tested in DawnCC were later put to new use in the design of TaskMiner, an automatic parallelization tool for multi-core CPUs [Ramos et al. 2018b]. Both DawnCC and the TaskMiner are today publicly available through online interfaces: users feed them with programs, and they give back parallel versions of said codes[1].

The Second Contribution: AutoParBench. The second of our contributions was AutoParBench: a test framework to support the design and the development of automatic

[1]DawnCC is available at `http://cuda.dcc.ufmg.br/dawn/`. The TaskMiner is available at `http://cuda.dcc.ufmg.br/taskminer/`

parallelization tools[2]. AutoParBench is centered around a representation that normalizes programs annotated with OpenMP 4.5 pragmas. This representation is based on the JavaScript Object Notation (JSON); thus, we call it a *JSON snapshot*. The main benefit of this common representation is the possibility to compare programs produced by different automatic parallelization tools. AutoParBench comes 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. Using this methodology, AutoParBench has let us discovered zero-day bugs in different automatic parallelizers: three bugs were confirmed in the Intel C Compiler, two in DawnCC, four in Rose AutoPar [Liao et al. 2010] and two in Cetus [Bae et al. 2013].

A Partnership between Industry and Academia. This dissertation is the product of a cooperation between the industry and the university. The development of DawnCC was sponsored by LG Electronics to support the parallelization of software running on smartphones. Thus, DawnCC has been used to transform sequential C programs into code that runs in GPUs of smartphones [Mendonça et al. 2017]. After the first release of DawnCC, LG has ported its technology to multicore CPUs. From this effort, the TaskMiner was produced, through a project financed at UFMG and at UNICAMP. The release of DawnCC drew the attention of researchers from Lawrence Livermore National Laboratory, who later contacted us at UFMG. From this contact, I, Gleison Mendonça, was invited for an internship at LLNL. Once there, I constructed AutoParBench under the supervision of Chunhua Liao.

A Work of Many Hands. As mentioned previously, this dissertation was part of a project financed by LG Electronics, CAPES and the U.S. Department of Energy. Hence, I have worked in close cooperation with many colleagues, from UFMG's Compilers Lab, from the Computer Systems Lab (UNICAMP) and from Lawrence Livermore National Lab. The bulk of the implementation of DawnCC is my work. DawnCC reuses a static analysis built by Péricles Alves [Alves et al. 2015], to infer the sizes of arrays. He also designed DawnCC's algorithm to find parallel loops. To support DawnCC's annotation system, Breno Guimarães [Mendonca et al. 2016] has implemented an algorithm to harmonize information from Clang (front-end) and LLVM's IR (back-end). I have worked in the implementation of TaskMiner; however, most of this effort was Pedro Ramos'. The design of AutoParBench is fruit of my effort and my mentors', but I have implemented the framework alone.

2. First Contribution: DawnCC

Directive-based programming models, such as OpenACC and OpenMP, allow developers to convert a sequential program into a parallel one with minimum human intervention. However, inserting pragmas into production code is a difficult and error-prone task, often requiring familiarity with the target program. This difficulty restricts the ability of developers to annotate code that they have not written themselves. The first part of the dissertation describes a suite of compiler-related methods to mitigate this problem. Such techniques rely on symbolic range analysis, a well-known static technique, to achieve two

[2]AutoParBench is available at `https://github.com/LLNL/AutoParBench`.

```
1  void
2  saxpy_serial(int n, float alpha, float *x, float *y) {
3      for (int i = 0; i < n; i++) {
4          y[i] = alpha*x[i] + y[i];
5      }
6  }
```

(a)

```
1  __global__ void
2  saxpy_parallel(int n, float alpha, float *x, float *y) {
3      int i = blockIdx.x * blockDim.x + threadIdx.x;
4      if (i < n) {
5          y[i] = alpha * x[i] + y[i];
6      }
7  }
8  ...
9  // Allocate x and y, and copy them to the GPU ...
10 // Invoke the parallel kernel:
11 int nblocks = (n + 255) / 256;
10 saxpy_parallel <<<nblocks, 256>>>(n, 2.0, x, y);
11 // Copy y back to the CPU ...
```

(b)

Figure 1. (a) Standard C implementation of the Single Precision $AX + Y$ (SAXPY) kernel. (b) Same algorithm written in C for CUDA.

```
1  void saxpy_serial(int n, float alpha, float *x, float *y) {
2    long long int tmp[2];
3    tmp[0] = n -1;
4    tmp[1] = ((tmp[0] > 0) ? tmp[0] : 0); // upper bound
5
6    char x_y_alias_free = ((x >= y + tmp[1] + 1) ||
7                           (y >= x + tmp[1] + 1));
8
9    #pragma acc data pcopy(y[0:tmp[1]]) \
10                  pcopyin(x[0:tmp[1]]) \
11                  if(x_y_alias_free)
12   #pragma acc kernels loop independent \
13                  if(x_y_alias_free)
14   for (int i = 0; i < n; i++)
15     y[i] = alpha*x[i] + y[i];
16 }
```

(a)

```
1  void saxpy_serial(int n, float alpha, float *x, float *y) {
2    long long int tmp[2];
3    tmp[0] = n -1;
4    tmp[1] = ((tmp[0] > 0) ? tmp[0] : 0); // upper bound
5
6    char x_y_alias_free = ((x >= y + tmp[1] + 1) ||
7                           (y >= x + tmp[1] + 1));
8
9    #pragma omp target data map(to:x[0:tmp[1]]) \
10                  map(tofrom:y[0:tmp[1]]) \
11                  if(x_y_alias_free)
12   #pragma omp parallel for if(x_y_alias_free)
13   for (int i = 0; i < n; i++)
14     y[i] = alpha*x[i] + y[i];
15 }
```

(b)

Figure 2. (a) SAXPY annotated with OpenACC pragmas. (b) SAXPY annotated with OpenMP pragmas. The gray area denotes code created automatically by DawnCC.

purposes: populate source code with data-transfer primitives and to disambiguate pointers that could hinder automatic parallelization due to aliasing. We have materialized our ideas into a tool, DawnCC, which can be used standalone, or through an online interface. To demonstrate its effectiveness, we have shown how DawnCC can annotate the programs available in PolyBench without any intervention from users. Such annotations lead to speedups of over 100x in an Nvidia architecture, and over 50x in an ARM architecture.

We use the *Single Precision AX + Y* (SAXPY) kernel in Figure 1 (a) to illustrate the contributions of this paper. This kernel runs in linear time on a sequential machine. However, it is $O(1)$ in the *Parallel Random-Access Machine* (PRAM) model, because there is no dependency between different iterations of the loop. Figure 1 (b) shows a direct translation of SAXPY to C for CUDA. CUDA's syntax is very similar to C's; however, its semantics is substantially different. Part of it, e.g., lines 1-7 is meant to run on a GPU in parallel; the rest, e.g., lines 9-11, is meant to run on a host CPU.

To make GPUs more accessible to the everyday developer, the high-performance computing community has designed a number of *annotation systems*. An annotation system is a meta-language that changes the semantics of a host language. In our setting, the host language is C, and the meta-language is either OpenACC or OpenMP. Figure 2 shows the sequential SAXPY kernel annotated with (a) OpenACC and (b) OpenMP prag-

mas. Our DawnCC compiler inserts these pragmas, plus all the code necessary for them to work, automatically. The tool is still restricted to regular loops. It is not able, for instance, to parallelize an algorithm that traverses a graph represented as a mesh of pointers.

3. Second Contribution: AutoParBench

The second contribution of this work, AutoParBench, is a framework to compare the output of different automatic parallelization tools. Its goal is to uncover bugs in these tools. Testimony to its effectiveness are the bugs confirmed as consequence of its usage. AutoParBench has found 3, 2, 4 and 2 bugs in ICC, DawnCC, AutoPar and Cetus, respectively. Figure 3 shows the main components of AutoParBench. These components are briefly summarized below.

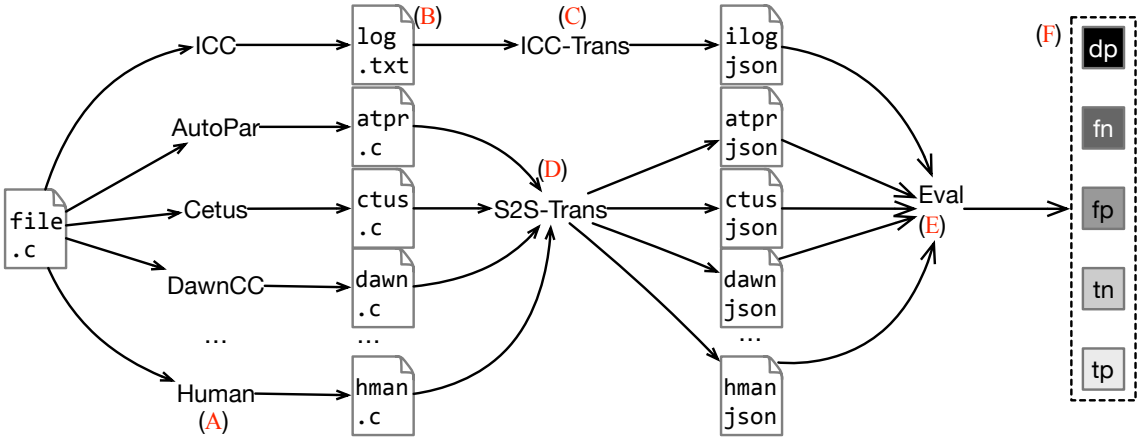

Figure 3. An overview of AutoParBench.

AutoParBench provides developers with a reference collection of 99 programs (A). These programs contain 1,579 loops, which have been manually annotated with OpenMP directives. These benchmarks were taken from well-known collections, such as NAS [Seo et al. 2011] and Rodinia [Che et al. 2009]. AutoParBench provides front-ends to different annotation systems used on C/C++ programs and on logs produced by Intel's ICC (B). The goal of these front ends is to extract the loops and the corresponding annotations from C source files. AutoParBench contains two translators to convert annotated programs into JSON snapshots. One of these translators read ICC's logs (C), for ICC does not produce annotated sources, but rather binary files. The other parser reads C source files and extracts OpenMP annotations from them, to then convert them into JSON objects (D).

The most intricate part of AutoParBench are the snapshot evaluators. These tools (E) semantically compare JSON snapshots. From this comparison, AutoParBench generates graphic or quantitative reports (F). Said reports classify the loops annotated by tools as true positives or negatives (annotations deemed correct), or as false positives or negatives (annotations with high probability to be bugs). In the effort to debug auto-parallelizers, we have designed a protocol to verify these reports. This protocol reduces the amount of user intervention necessary to validate warnings, and sorts warnings by relevance. We emphasize that developers never have to annotate programs to use AutoParBench. The only human intervention that we require is the verification of warnings, once they are issued. This pass is necessary to confirm or refute bugs.

Examples of confirmed bugs. ICC parallelizes the first loop in the program in Figure 4 (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 can be exercised by Intel Inspector when the variable `len` is assigned a value of 16. Figs. 4(b-d) show a bug that was discovered in Cetus. Cetus, when given the program in Figure 4 (b), produces the code in Figure 4 (c). Cetus extracts variable `j` from the loop, and transforms it into a reduction. Said reduction causes a runtime crash. The version of this program in the reference collection appears in Figure 4 (d). By comparing JSON snapshots produced for Figure 4 (c) and (d), AutoParBench warns the developer about the potential bug.

```
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)

```
#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)

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

Figure 4. (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.

In addition to comparing tools for correctness, AutoParBench also allows us to compare them for performance. To this effect, AutoParBench contains a set of curated inputs for long-running benchmarks in its reference collection. The original description of AutoParBench [Mendonça et al. 2020] contains a comparison of six different versions of parallel programs—five of them produced automatically by different tools.

4. Conclusion: Summary of Constributions

This MSc dissertation has helped to advance the design and implementation of automatic parallelization tools. Such help comes from AutoParBench, a framework that supports the testing of automatic parallelization tools, and from DawnCC, which is, itself, one such a tool. In addition to software, from this work resulted papers and bug reports—contributions that we summarize as follows:

Papers: The main publication describing DawnCC appeared in the ACM Transactions on Architecture and Code Optimization (TACO) [Mendonça et al. 2017]. AutoParBench is described in a paper published on the International Conference on Supercomputing (ICS) [Mendonça et al. 2020]. In addition to these two works, several other papers have been published as a consequence of this research [Mendonca et al. 2016, Ramos et al. 2018a, Ramos et al. 2018b].

Software: DawnCC is accessible at `http://cuda.dcc.ufmg.br/dawn/`. AutoParBench is available at `https://github.com/LLNL/AutoParBench`. TaskMiner can be used at `http://cuda.dcc.ufmg.br/taskminer/`.

Bugs: Several bugs have been reported as a consequence of this work. Some of them have been confirmed and posteriorly fixed[3]. Others are still awaiting for confirmation at the time of SBC's CTD.

References

Alves, P., Gruber, F., Doerfert, J., Lamprineas, A., Grosser, T., Rastello, F., and Pereira, F. M. Q. a. (2015). Runtime pointer disambiguation. In *OOPSLA*, page 589–606, New York, NY, USA. ACM.

Bae, H., Mustafa, D., Lee, J.-W., Aurangzeb, Lin, H., Dave, C., Eigenmann, R., and Midkiff, S. P. (2013). The cetus source-to-source compiler infrastructure: Overview and evaluation. *International Journal of Parallel Programming*, 41(6):753–767.

Bondhugula, U., Hartono, A., Ramanujam, J., and Sadayappan, P. (2008). A practical automatic polyhedral parallelizer and locality optimizer. In *PLDI*, page 101–113, New York, NY, USA. ACM.

Che, S., Boyer, M., Meng, J., Tarjan, D., Sheaffer, J. W., Lee, S.-H., and Skadron, K. (2009). Rodinia: A benchmark suite for heterogeneous computing. In *IISWC*, pages 44–54, Washington, DC, USA. IEEE.

Liao, C., Lin, P.-H., Asplund, J., Schordan, M., and Karlin, I. (2017). Dataracebench: A benchmark suite for systematic evaluation of data race detection tools. In *SC*, pages 11:1–11:14, New York, NY, USA. ACM.

Liao, C., Quinlan, D. J., Willcock, J. J., and Panas, T. (2010). Semantic-aware automatic parallelization of modern applications using high-level abstractions. *Int. J. Parallel Programming*, 38(5):361–378.

Mendonca, G. S. D., Guimarães, B. C. F., Alves, P. R. O., Pereira, F. M. Q., Pereira, M. M., and Araujo, G. (2016). Automatic insertion of copy annotation in data-parallel programs. In *SBAC-PAD*, pages 34–41, Los Alamitos, CA, USA. IEEE.

Mendonça, G., Guimarães, B., Alves, P., Pereira, M., Araújo, G., and Pereira, F. M. Q. a. (2017). DawnCC: Automatic annotation for data parallelism and offloading. *ACM Trans. Archit. Code Optim.*, 14(2):13:1–13:25.

Mendonça, G. S. D., Liao, C., and Pereira, F. M. Q. a. (2020). Autoparbench: A unified test framework for openmp-based parallelizers. In *ICS*, New York, NY, USA. ACM.

Ramos, P., Souza, G., Leobas, G., and Pereira, F. M. Q. (2018a). Taskminer: Automatic identification of tasks. In *SBLP*, page 11–18, New York, NY, USA. ACM.

Ramos, P., Souza, G., Soares, D., Araújo, G., and Pereira, F. M. Q. a. (2018b). Automatic annotation of tasks in structured code. In *PACT*, New York, NY, USA. ACM.

Seo, S., Jo, G., and Lee, J. (2011). Performance characterization of the NAS parallel benchmarks in opencl. In *IISWC*, pages 137–148, Piscataway, NJ, USA. IEEE Press.

[3]For an example of discussion that our bug reports have raised, see this thread in the Intel mailing list: `https://community.intel.com/t5/Intel-C-Compiler/ICC-19-0-4-243-parallelized-loop-with-confirmed-Race-Condition/m-p/1173793`.