Bottleneck Analysis of Parallel Programs with Profiling tools

Scott Chu, Qiyuan Huang, Yiwei Shao, Juexiao Zhang
{zc2396,qh2086,ys5070,jz4725}@nyu.edu
New York University
NY, USA

ABSTRACT

Identifying bottlenecks in a parallel program is an important and challenging task. In this project, we designed and implemented a bottleneck prediction system that leverages popular profiling tools and can hierarchically analyze and report potential bottlenecks for OpenMP parallel programs. We implemented a set of programs with different characteristics as benchmarks to test and showcase the functionality and effectiveness of the system. Experiments showed that the system can accurately measure benchmark programs and report their potential bottlenecks. We believe that the hierarchical bottleneck analysis proposed by our system, with further refinement, can help developers analyze bottlenecks and improve their parallel programs. Our project can be found at: our repository.

KEYWORDS

Bottleneck Analysis, OpenMP, Parallel Programming, Profiling

1 INTRODUCTION

With Moore's Law gradually coming to an end, the significance of obtaining performance improvements from the system and software side, i.e. "room at the top" [4], has been drawing greater attention in the computer science community. With the rise of multicore processors and distributed computing systems, writing parallel programs has become one of the biggest sources of performance boost from the "top". This has made parallel programming more important than ever before. However, designing efficient parallel programs is a challenging task that requires a deep understanding of the underlying hardware architecture, software frameworks, and algorithms. One of the main challenges in parallel programming is identifying and mitigating bottlenecks, which are the parts of a program that limit its performance and scalability. Therefore, building a system that can identify or predict potential bottlenecks will be very helpful for analyzing and improving parallel programs.

In an effort to address this challenge, we designed and implemented a bottleneck prediction system that can identify a list of potential bottlenecks in an OpenMP program. The system is based on popular profiling tools such as Valgrind [5] and omptracing [6]. It measures the program's execution traces and hierarchically identifies patterns that indicate performance bottlenecks. Developers can optimize their OpenMP program based on the list of bottlenecks suggested by our system.

In this course project, we designed and implemented the system, and implemented a set of parallel programs with different characteristics and workloads to evaluate the system. We ran the programs with different parameters such as the number of threads and scheduling schemes to investigate their impact and showcase the functionality and effectiveness of our system.

The rest of this paper is organized as follows. Section 2 provides a brief literature background on parallel programming, profiling, and bottleneck detection techniques. Section 3 describes the design and implementation of our system. Section 4 presents the experimental setup as well as the benchmark. Section 5 discusses the experimental results. Finally, Section 6 concludes the paper, summarizes the main contributions, and points out the limitations.

2 LITERATURE

When designing a bottleneck prediction system, defining and identifying specific types of bottlenecks is always challenging. In this paper [7], the authors defined a set of bottlenecks and analyzed the performance using OpenMP. However, their work does not propose a standard for identifying these bottlenecks in a given program. In another paper [3], the author formulates the bottleneck finding problem as an information overload challenge and a data collection challenge. The former is that to find the bottleneck, one has to decide what information is useful among the various metrics and visualizations obtained from measuring performance. The latter is a dilemma where it is difficult to choose what data to collect from analyzing a program without knowing the bottleneck, but to isolate the bottleneck, one has to collect data. To address this challenge, they came up with two solutions to find the most useful information among the various performance metrics and collect the required data for analysis as efficiently as possible. They also developed a library of potential bottlenecks which they termed as hypotheses.

Our system draws wisdom from theirs and is conceptually similar. We define a set of potential bottlenecks and collect corresponding data for them. However, we focus on OpenMP programs and design a hierarchical framework to organize the data collection and analysis of each potential bottleneck. Furthermore, our system is built on more recent progress in profiling parallel programs. Valgrind [5] and Cachegrind [2] provide powerful tools for analyzing a program's memory access pattern. Recent progress in OpenMP programming has enabled a handy tracing tool [6] for OpenMP programs. Our system takes advantage of these effective tools and builds an information collection and analysis workflow on top of them. By doing so, we can develop a system that can accurately measure programs and report their potential bottlenecks.

3 METHODS

In this section, we first present the overall framework of our system and how the profiling tools are incorporated into our system. Then

we discuss in detail the specific features we implement in our system to test and predict the potential bottlenecks.

3.1 Overall framework

The overall framework of our system is illustrated in the figure 1. Since the problem size has a great impact on the running time of a parallel program, as well as the measurements, we find that it is more reasonable to take the problem size as an input to the system. But no matter the problem size, the system will do two kinds of measure. One is cache check and we leverage the output from profiling tools such as the Cachegrind in Valgrind. The other is thread check, in which we use omptracing to monitor, trace and record the running time of the whole program and each of the threads. The thread check is further divided into scaling check, sync check and work region check. They are performed in a hierarchical manner to identify the specific bottlenecks. We will introduce these features in detail below.

3.2 Features

In this subsection, we present each of the bottleneck checks implemented in the system. They form the key features of our bottleneck predicting system.

Cache check: The cache is the most important part for both read and write. When data is not found in the cache, the program has to fetch it from a slower memory level, which increases the latency and slows down the execution. So we use the Cachegrind[2] in Valgrind to get the cache info, including the overall *cache miss times* and specific function's *cache miss times*. Then we calculate the *cache miss rate* in each memory level. We set several thresholds to judge if a program is influenced by the cache miss bottleneck. The thresholds are summarized through the experiments of testing different programs on NYU CIMS machine. It is below the theoretical threshold which is 5 percents.[1] Those cache miss rates are also correlated to the *Limited Scalability*, which means as the number of threads in a parallel program increases, the likelihood of cache contention and cache coherence traffic also increases.

Scaling check: Our system checks if the tested program's implementation is not optimized enough to approach the theoretical speedup. First we just do the strong scalability check, simply increasing the threads number and check if we can gain the speedup proportional to the thread number. If it scales close to theoretical upper bound, there is no bottleneck.

Parallel region check: If the program does not scale well, our system goes on to identify the possible cause for that as is shown in figure 1. According to Amdahl's Law,

$$S(N) = \frac{1}{(1-P) + P/N} \tag{1}$$

we cannot get more speedup than $\frac{1}{1-P}$. It measures the program using omptracing. By setting the number of threads to 1, we can get the percentage of the parallel region. Small P suggests that there is not much potential performance gain from this program by parallelization, thus indicates the bottleneck is lack of parallelism.

Sync check: If the parallel region takes the dominating part but not scaling well, our system continue to perform the synchronization check (sync check). From the omptracing generated json file, our system is able to analyze and obtain the program's *parallel region*, and each thread's *thread region*, marking the begin and the end of thread and *work region*, marking the begin and the end of work regions like loop, taskloop, sections, workshare, single regions, measured by the running time. Based on ratio of the work regions in the parallel regions, the system gets to know how much of the parallel part is dedicated to actually perform parallel work. If the work region has already taken the most part of a parallel region, while the program scales poorly, our system will speculate that the bottleneck is lack of parallelism. This suggest that the parallel part is likely not very sub-optimal, instead, the developers should try to optimize the parallel part, making it smaller or making the parallel part larger. We still leverage the measurements obtained from omptracing. By calculating the difference between each thread's work region and the parallel regions, we get to estimate the time spent on overhead such as creating and joining threads and threads synchronization. By identifying what overhead takes a relatively larger part of the non-work region, we can accordingly speculate the bottleneck to be load imbalance or threads overhead.

4 EXPERIMENTS

In this section, we discuss the setup of our experiments. We present the benchmarks for testing the bottlenecks in 4.1. Our programs are tested on NYU CIMS machines as well as a Docker. The system is able to run in a Linux system with C and C++ compiler and Python with several basic libraries. The specific configuration for the environment can be found at our repository.

4.1 Benchmark

In order to test our systems and showcase the bottlenecks, we designed and implemented several benchmark parallel programs based on OpenMP. We design the benchmark programs according to the different components of our system, allowing us to showcase the functionality of our system in a more targeted manner. The benchmarks can be divided into the following sets:

Cache access and cache miss: We select several matrix computation programs involving high cache read and write to measure the cache miss. Those programs also take different input sizes and thread numbers, which we used to make comparison tests. These programs demonstrates a high cache miss rate due to poor cache locality and cache coherence issues that can lead to performance bottlenecks in parallel programming. We also use different methods to do the same calculation: matrix multiply to compare if there is a improvement in cache locality. The results are presented in 5.1.

Parallelism and scaling: This part of the system aims to test if the tested program can be re-implemented with better parallelism. Our analysis compares the theoretical maximum speedup of the tested program against the actual speedup to get how much potential optimization can still be applied to the current implementation. Large gaps between the theoretical speedup and the actual speedup

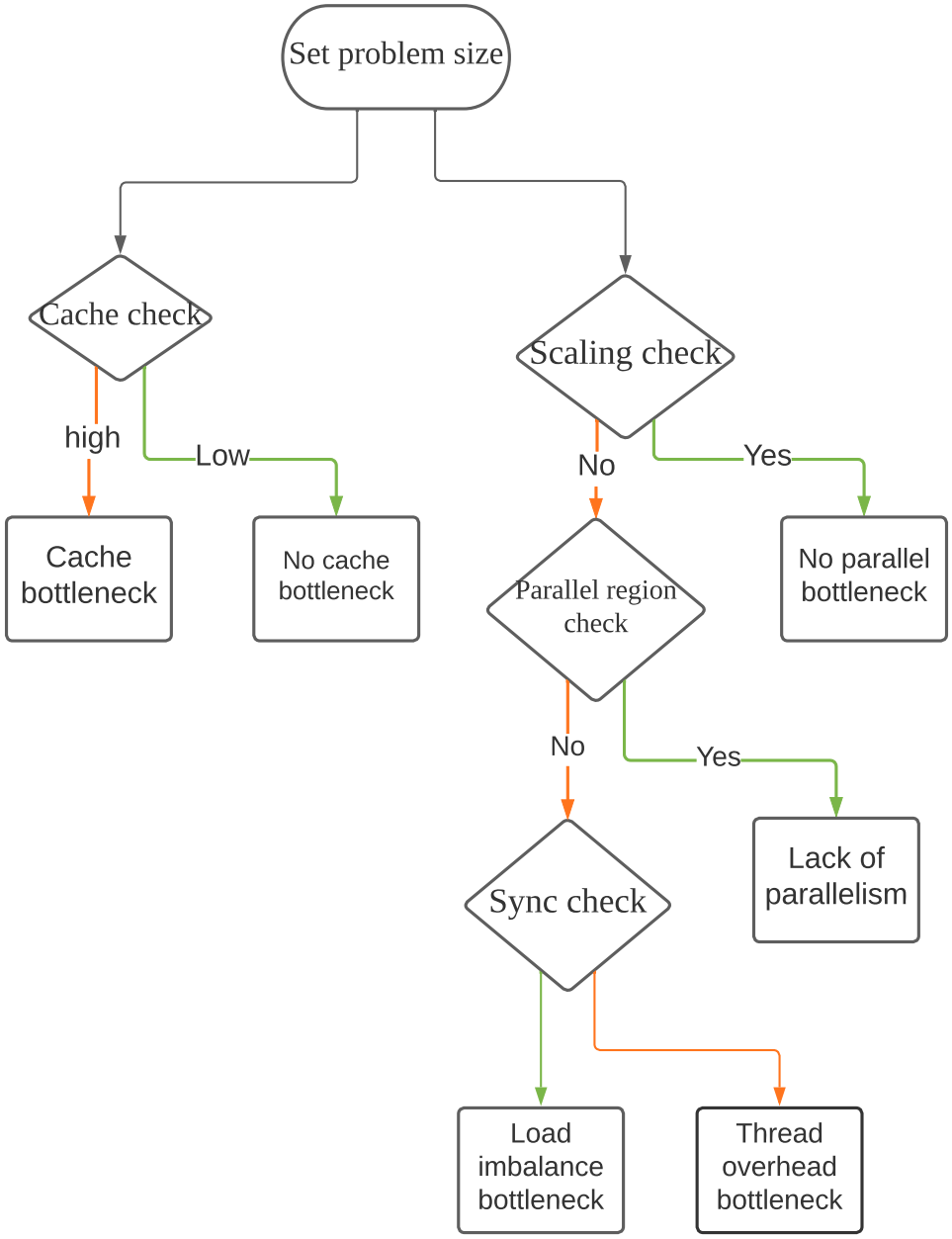

Figure 1: Overall framework of the system.

implies the tested program lacks parallelism.

Synchronization and load imbalance: This set of programs aims to test the system's functionality of conducting *Work Region Check* and *Sync Check*. The programs derive from a simple program that has two for loops where each iteration simply perform an independent function $f(x)$ that runs in time proportional to x. For the first for loop x is set to be the iteration indicator i and for the second it is $n - i$ where n is the total number of iterations. In this way we create a naturally imbalance workload for naive omp scheduling.

Extended from this program, we write several variants with different scheduling mechanism to compare the performances and showcase the potential bottleneck.

The details of the benchmark programs can be found in our repository.

5 RESULTS

In this section we present the results obtained from the experiments on the benchmark programs. We will analyze the measurements obtained with the profiling tools and discuss the bottlenecks reflected by them.

5.1 Cache access and cache miss

For each program, we do experiments with different thread numbers to check its speedup. The figure 2 shows the program of matrix multiplication using direct methods(3 loops). The program was run on NYU CIMS Crunchy server. We can see there is no speedup in execution, even when the matrix size is big(over 500).

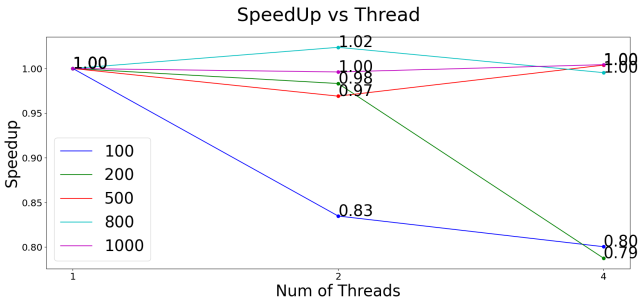

Figure 2: Speedup for matrix multiplication

Then we use valgrind to check its cache access. Results are shown in figure 3. We divide the Level 1 Data Read Miss to the Overall Data Read. The figure 3 shows when the matrix size is over 500, The Miss rate is saturated at 4.6 percent.

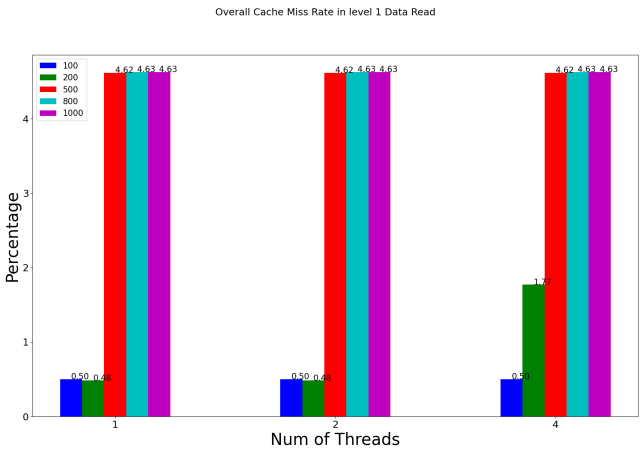

Figure 3: Level 1 Overall Cache Bottleneck

To ensure that this high miss rate is made by the computation part which is paralleled using OpenMP, we also check the cachegrind specific report and gather the following data in figure 4.

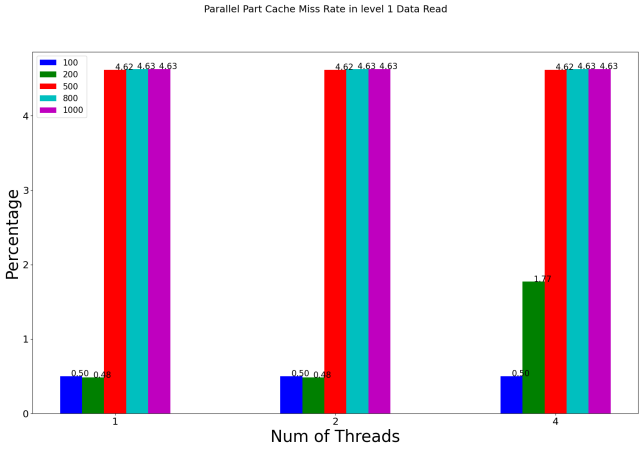

Figure 4: Level 1 Parallel Part Cache Bottleneck

We can see that it yields the same number of cache miss rates in this part, which proves that the high cache miss rate is caused by the parallel computation part. The high cache miss rate in this program occurs because of the way the matrices are accessed during multiplication. When a core accesses a particular element in the matrix, it is likely that the surrounding elements will be accessed soon after. However, due to the nested loop structure and matrix layout, the surrounding elements are not always accessed in an optimal order, leading to cache misses and cache coherence issues.

To strengthen our conclusion, we also use another way to do the matrix multiplication. We split each matrix into blocks to optimize cache locality and reduce the cache miss rate. We fix the matrix size to 1024*1024, and try to use different block size.

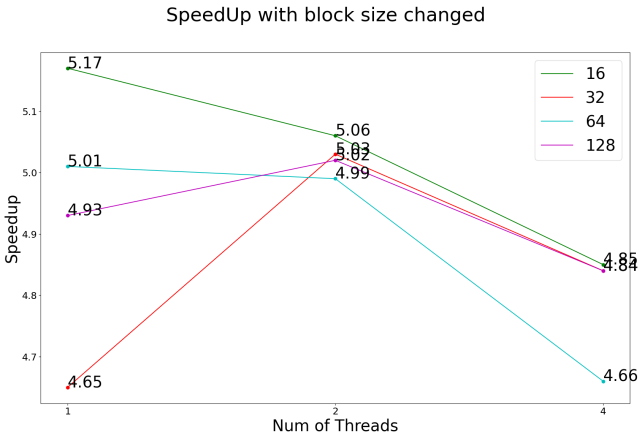

Figure 5: Change block size in matrix multiplication. Block-sizes are listed in the legend.

As is shown in figure 5, Block size 16 here outperforms the others. This is because it fits the L1 cache best in our experiment environment. All programs have a better speedup (>1) compared to the previous simple algorithm program. And we also check the cache miss rate of these programs to see if it has strong relation with the speedup.

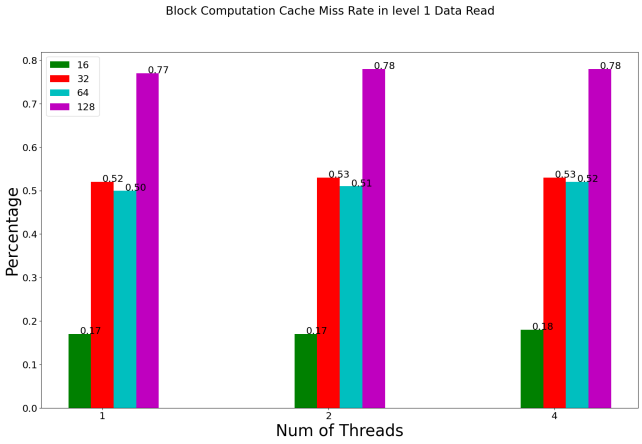

Figure 6: Block size and cache miss rate

Block size 16 gets the smallest cache miss rate, and the overall graph show the same ranks to the speedup. All of them have much better Level 1 data miss rate compared to the previous one (4.6 percent). We can conclude that by using the block algorithm, the program accesses the elements in a more cache-friendly manner, improving the cache locality and potentially reducing the cache miss rate.

So for a program running with Valgrind, we can tell it encounters a cache bottleneck when its Level 1 data cache miss rate is getting close to the threshold 4.6 percent. This threshold is the upper bound of the level 1 data cache for any read or write instruction to fetch.

To see more info about our other experiments about cache, and those experiments' Level 2 cache result and graphs, please check the repository.

5.2 Parallelism and scaling

We experiment several parallel programs to test its strong scalability and parallel region ratio. *sum_critical* and *sum_reduction* both sum up all the elements in a vector. *sum_critical* uses *#pragma omp critical* to ensure the correctness while *sum_reduction* uses *#pragma omp reduction(+:sum)*. *genprime_atomic* and *genprime_critical* generate primes with a naive algorithm same as lab2. The only difference is that *genprime_critical* uses #pragma omp critical to write primes while *genprime_atomic* uses #pragma omp atomic write to write prime. Theoretically, *sum_reduction* and *genprime_atomic* should scale better.

Figure 7 is the fit plot of running time with different thread numbers. Since it is a *log-log* plot, smaller fit line slope, better

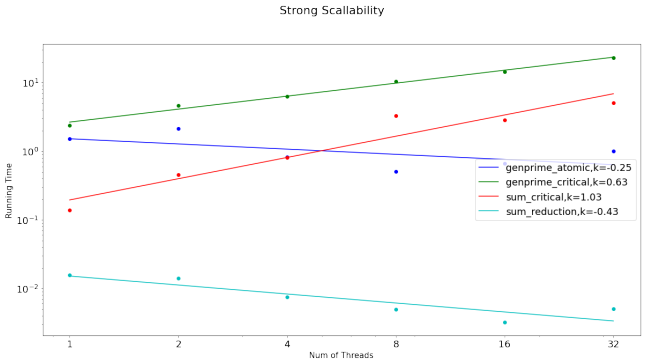

Figure 7: Actual vs Theoretical Speedup of different parallel programs.

Program	Parallel region ratio
genprime_atomic	0.9136
genprime_critical	0.9794
sum_reduction	0.1254
sum_critical	0.7679

Table 1: Ratio of parallel region in the whole program

scalability. As expectation, *sum_reduction* and *genprime_atomic* perform better as thread numbers increasing.

Table 1 shows the proportion of parallel region in the whole program. *genprime_critical* is the highest and *sum_reduction* is the smallest.

According to the figure 1, *sum_reduction* scales well, so there is no bottleneck in the program. *genprime_critical* and *genprime_atomic* have pool scalability and big proportion of parallel region, suggesting their lack of parallelism, which matches our expectation since generating prime number with such a naive algorithm is hard to parallel. *sum_critical* scales poorly and has a small parallel part. There is some synchronization problem requiring further checking.

5.3 Synchronization and load imbalance

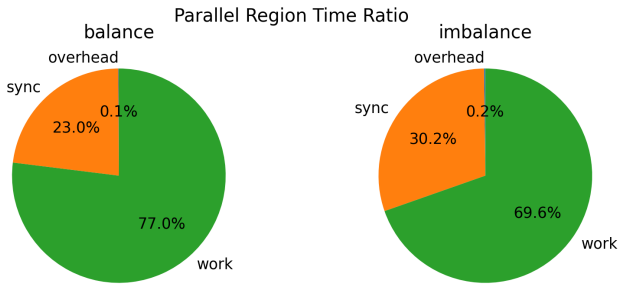

Figure 8: Different sync and work ratios of the parallel region.

We experiment our system's work region check and sync check on the benchmark. A showcase of the region ratios of two benchmark programs are shown in figure 8. As mentioned in section 4.1, the two programs runs a dummy function $f(x)$ in a for loop which creates an imbalanced load between the threads. First, we can see that both programs has a work region larger than 50% of the total parallel region and the system deems that this can pass the work region check. Then it comes to the sync check. The difference between the two programs is that one uses a default static scheduling, marked as "imbalance", while the other uses cyclic scheduling (static, 1), marked as "balance". And the system is able to measure and report the difference in time. We can see that the imbalance version spend more than 25% of the time for synchronization while the more balanced version spends 7% less. As a result, the system indicates the program has a load imbalance bottleneck.

6 CONCLUSION

In this project, we designed and implemented a system that utilizes popular profiling tools and can hierarchically measure parallel programs and report potential bottlenecks. We also implemented several benchmark programs using OpenMP to test and showcase the functionality and effectiveness of our program. During the course of developing this system, we found that the most challenging parts were identifying types of bottlenecks, clearly differentiating and organizing measuring and profiling, and setting standards for reporting potential bottlenecks.

We believe our system has rigorously addressed the first two challenges, but for the last one, our solution remains heuristic. We found that the performance evaluation of a parallel program can be closely dependent on factors such as user expectations, hardware and system specifications, application scenarios, etc. In light of these factors, one system with specific measurements and standards like ours is difficult to handle the various cases. Therefore, we think it may be a good idea for future bottleneck predicting systems to study and learn to adapt to these factors for more desirable and flexible predictions.

REFERENCES

[1] William Cohen. 2014. *Determining whether an application has poor cache performance.* Technical Report. RedHat Developer.
[2] Valgrind™ Developers. 2000-2022. *Cachegrind: a cache and branch-prediction profiler.* Technical Report.
[3] Jeffrey K Hollingsworth. 1994. *Finding Bottlenecks in Large Scale Parallel Programs.* Technical Report. University of Wisconsin-Madison Department of Computer Sciences.
[4] Charles E Leiserson, Neil C Thompson, Joel S Emer, Bradley C Kuszmaul, Butler W Lampson, Daniel Sanchez, and Tao B Schardl. 2020. There's plenty of room at the Top: What will drive computer performance after Moore's law? *Science* 368, 6495 (2020), eaam9744.
[5] Nicholas Nethercote and Julian Seward. 2007. Valgrind: a framework for heavy-weight dynamic binary instrumentation. *ACM Sigplan notices* 42, 6 (2007), 89–100.
[6] Vitoria Pinho, Hervé Yviquel, Marcio Machado Pereira, and Guido Araujo. 2020. OmpTracing: Easy Profiling of OpenMP Programs. (2020), 249–256. https://doi.org/10.1109/SBAC-PAD49847.2020.00042
[7] Vibha Rajput and Alok Katiyar. 2013. Proactive bottleneck performance analysis in parallel computing using openMP. *arXiv preprint arXiv:1311.1907* (2013).