

PaScal Viewer: a Tool for the Visualization of Parallel Scalability Trends

1st Anderson B. N. da Silva

Pró-Reitoria de Pesquisa, Inovação e Pós-Graduação
Instituto Federal da Paraíba
João Pessoa, Brazil
anderson.silva@ifpb.edu.br

2nd Daniel A. M. Cunha

Dept. de Eng. de Comp. e Automação
Univ. Federal do Rio Grande do Norte
Natal, Brazil
damouracunha@gmail.com

3rd Vitor R. G. Silva

Dept. de Eng. de Comp. e Automação
Univ. Federal do Rio Grande do Norte
Natal, Brazil
ramos.vitor89@gmail.com

4th Alex F. de A. Furtunato

Diretoria de Tecnologia da Informação
Instituto Federal do Rio Grande do Norte
Natal, Brazil
alex.furtunato@ifrn.edu.br

5th Samuel Xavier de Souza

Dept. de Eng. de Comp. e Automação
Univ. Federal do Rio Grande do Norte
Natal, Brazil
samuel@dca.ufrn.br

Abstract—Taking advantage of the growing number of cores in supercomputers to increase the scalability of parallel programs is an increasing challenge. Many advanced profiling tools have been developed to assist programmers in the process of analyzing data related to the execution of their program. Programmers can act upon the information generated by these data and make their programs reach higher performance levels. However, the information provided by profiling tools is generally designed to optimize the program for a specific execution environment, with a target number of cores and a target problem size. A code optimization driven towards scalability rather than specific performance requires the analysis of many distinct execution environments instead of details about a single environment. With the goal of providing more useful information for the analysis and optimization of code for parallel scalability, this work introduces the PaScal Viewer tool. It presents an novel and productive way to visualize scalability trends of parallel programs. It consists of four diagrams that offers visual support to identify parallel efficiency trends of the whole program, or parts of it, when running on scaling parallel environments with scaling problem sizes.

Index Terms—parallel programming, efficiency, scalability, performance optimization, visualization tool

I. INTRODUCTION

The number of cores in supercomputers continues to grow. Taking advantage of this to increase the performance of parallel programs is a continuous challenge. Developers must understand how their program behaves when more cores are used to process data, when more data need to be processed, or both—when data need to be processed by more cores.

Many techniques and profiling tools have been developed to assist programmers in the process of collecting and analyzing data related to the execution of their program [1]. These tools provide a large amount of information, measurements, details, and characteristics of the program. All of this information is usually related to one single execution of the program in a

particular environment. The configuration of this environment includes the number of cores, their operating frequency, and the size of the input data or the problem size. Among the various collected information, the elapsed time in each function of the code, the number of function calls, and the memory consumption of the program can be cited to name just a few [2]. It is possible to measure the parallel efficiency of the program in the analyzed environment. However, this value would probably change if the same program is executed in different environments. For this reason, from the information collected from a single run alone, it is not possible to evaluate how the efficiency will evolve when the program is executed with a different number of cores or with a different problem size. To discover the efficiency trends, developers need to perform the same analysis in many different environments. Then, they can compare the data manually and say if and when the algorithm tends to be scalable.

The information provided by these profiling tools is very useful to optimize the program execution for a single environment, with a target number of cores and a target problem size. However, when the goal is to analyze and optimize the code for parallel scalability, developers need to focus their attention on the variation of efficiency values when the program runs in many distinct execution configurations. In this case, it is more relevant to know fewer details about many executions than many more details about a single execution. In addition, current profiling tools use different techniques for collecting and analyzing data and, in some cases, they are developed for specific architectures and/or different parallelization models. These tools present the information collected in large data tables or complex graphs, and because of that, demand a "good" knowledge of their visualization interfaces [3]. Some approaches, such as [4], present the efficiency values for different environments in a single line chart. From such chart, developers could infer the program scalability trends, but this task is not always simple for a large number of environment

configurations are depicted. Weak scalability trend are also difficult to infer from these line charts.

SPERF is a simple tool that can automatically instrument parallel code with the insertion of time measurement [5]. From these measurements, developers can assess the execution time of each parallel region or regions of interest. With these specific time measurements, developers can verify the efficiency of the whole program or part of it. They can check, for example, if the scalability of the program as a whole deteriorates because of a specific region. In this way, they can focus in code regions that breaks scalability, much in the same way they do for optimizing single-run performance bottlenecks using traditional tools. This work uses the output of SPERF to construct visualization diagrams that unveils scalability trends. However, since the output of SPERF is a formatted text file, it can also be generated by another tool or by a productivity script.

This paper presents a visualization tool for the scalability analysis of parallel programs. For this, the tool takes as input the execution time measurements of a program in different configuration environments as provided by SPERF, translates these values into the corresponding efficiency values, and presents, through simple diagrams, the efficiency trends of this program. The objective of the tool is to avoid a tedious manual comparison of efficiency values. The tool is independent of architecture, parallelization model or profiling tool. It displays four color diagrams related to each analyzed region. One diagram holds the efficiency values and the other three show the variation of these values: when the number of cores is fixed; when the problem size is fixed; and when the number of cores and the problem size change proportionally at given rates. This tool can assist developers during the scalability analysis of their parallel programs in a simple and productive way. It helps on the identification of hot spots that when refactored could optimize the program scalability.

The remainder of this work is organized as follows Section II describes the tool and its color diagrams. The results of a simple case study are presented in Section III. Section IV presents the related work. And finally the contribution is summarized in Section V with an outlook of future works.

II. THE PaScAL VIEWER

This work presents a tool that introduces a novel and productive way to view the scalability trends of a parallel program, named Parallel Scalability Viewer or simply PaScal Viewer. For this, the tool translates the efficiency values collected from program executions into four color diagrams. These diagrams offer support to identify efficiency variation when the program runs on parallel environments and when it processes different amounts of data. In this sense, the tool aids developers in the identification of parallel scalability, including the analysis of whether this scalability is weak or strong, for the whole program or parts of it.

The tool is presented as a web-based application implemented in the Python programming language and the Django

framework [6]. The color diagrams are drawn using Bokeh, an interactive visualization library [7].

A. The Color Diagrams

The proposed color diagrams simplify the understanding and the visualization of the scalability trends from a parallel program. Fig. 1 presents the four diagrams generated from execution data collected from a theoretical program with the following characteristics:

- The serial execution time is given by $T_{\text{Serial}} = n^2$;
- The parallel execution time is given by $T_{\text{Parallel}} = n^2 / (p + \log_2(p))$;
- p corresponds to the number of cores and n corresponds to the problem size;

Each diagram is presented as a graphic of two axes. The horizontal axis corresponds to the number of cores and the vertical axis corresponds to the problem size. Both are organized in same order presented in the input file. The numerical values of each diagram element can be visualized in a tooltip, as shown in Fig.1.

The diagram located on the upper left corner of Fig. 1 presents the parallel efficiency values. Each element of the diagram, represented by a color, corresponds to a particular execution scenario, with a specific number of cores and problem size. The numerical values depicted in this first diagram are showed at Table I and Fig. 2. These values serve as base for constructing the other three diagrams and provide a general view of the program behavior.

TABLE I
EFFICIENCY VALUES OF A THEORETICAL PROGRAM.

	1	2	4	8	16	32	64	128	256	512	1024	2048	4096
i1	1,000	0,956	0,838	0,626	0,381	0,195	0,091	0,041	0,018	0,008	0,004	0,002	0,001
i2	1,000	0,988	0,952	0,866	0,703	0,483	0,278	0,141	0,066	0,031	0,014	0,006	0,003
i3	1,000	0,997	0,987	0,962	0,903	0,785	0,601	0,390	0,218	0,110	0,052	0,024	0,011
i4	1,000	0,999	0,997	0,990	0,973	0,935	0,856	0,717	0,524	0,327	0,179	0,090	0,043
i5	1,000	1,000	0,999	0,997	0,993	0,983	0,959	0,910	0,814	0,659	0,464	0,282	0,152
i6	1,000	1,000	1,000	0,999	0,998	0,996	0,990	0,976	0,946	0,885	0,776	0,610	0,417
i7	1,000	1,000	1,000	1,000	1,000	0,999	0,997	0,994	0,986	0,969	0,933	0,862	0,741
i8	1,000	1,000	1,000	1,000	1,000	1,000	0,999	0,998	0,996	0,992	0,982	0,962	0,920
i9	1,000	1,000	1,000	1,000	1,000	1,000	1,000	1,000	0,999	0,998	0,995	0,990	0,979
i10	1,000	1,000	1,000	1,000	1,000	1,000	1,000	1,000	1,000	0,999	0,999	0,998	0,995
i11	1,000	1,000	1,000	1,000	1,000	1,000	1,000	1,000	1,000	1,000	1,000	0,999	0,999
i12	1,000	1,000	1,000	1,000	1,000	1,000	1,000	1,000	1,000	1,000	1,000	1,000	1,000
i13	1,000	1,000	1,000	1,000	1,000	1,000	1,000	1,000	1,000	1,000	1,000	1,000	1,000

The other three diagrams present the results of the difference between the efficiency values represented in the first diagram for each two bordering execution scenarios. The colors in these diagrams change according to two distinct ranges. One range for the positive values and another for the negative ones. In the case of Fig. 1, the color range for the positive values varies from white to green (from #FFFFFF to #004337, in RGB) and for the negative values varies from white to brown (from #FFFFFF to #5D3506, in RGB).

The diagram located on the bottom left corner allows the scalability analysis of the program when the number of cores is fixed and the problem size increases. From it, developers can observe the general scalability trends of a program with relation to the increase of problem size. In this diagram,

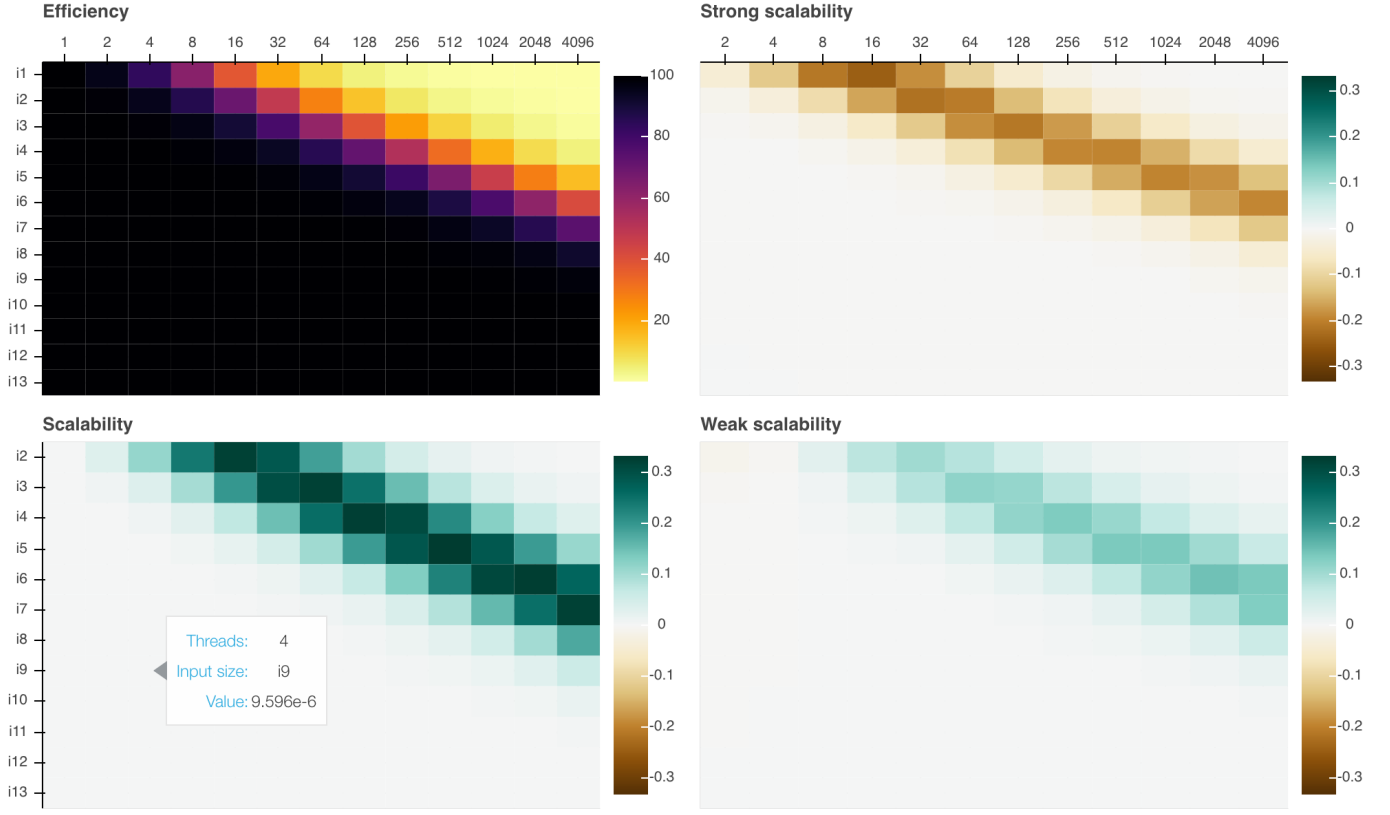


Fig. 1. Scalability diagrams of a theoretical program. The number of cores varies according to 1, 2, 4, 8, 16, 32, 64, 128, 256, 512, 1024, 2048 and 4096. The problem size, i1 to i13, varies according to 10, 20, 40, 80, 160, 320, 640, 1280, 2560, 5120, 10240, 20480 and 40960.

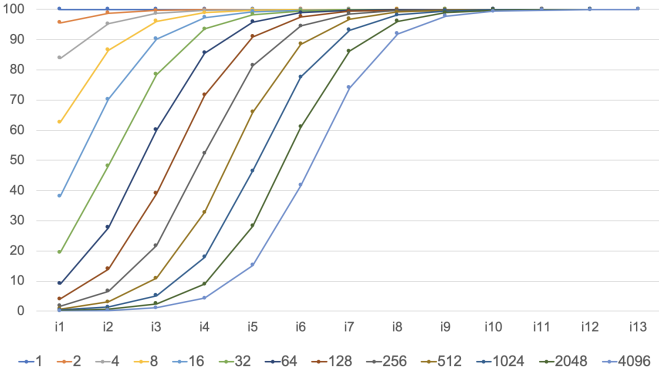


Fig. 2. Line chart with efficiency values in % of a theoretical program.

each element corresponds to the difference between efficiency values of two bordering configurations that use the same number of cores but with higher problem size. The values of each cell is given by

$$f(x, y + 1) - f(x, y), \quad (1)$$

where x represents the number of cores and y represents the problem size, both, presented in the efficiency diagram.

The diagram located on the upper right corner allows the scalability analysis of the program when the problem size is

fixed and the number of cores increases. From this diagram, developers can observe the strong scalability trends of a program. The elements of this diagram show the difference between the efficiency values of two bordering configurations that use the same problem size. In this case, the values are given by

$$f(x + 1, y) - f(x, y). \quad (2)$$

The diagram located on the bottom right corner allows the observation of weak scalability trends. The weak scalability trends can be identified in this diagram if the efficiency values increase or remain constant when the number of cores and the problem size used in executions scenarios increases with the same rate. In this case, the elements correspond the difference between the efficiency values of two bordering configurations in relation to the configuration that uses the higher number of cores and the higher problem size. These values are given by

$$f(x + 1, y + 1) - f(x, y). \quad (3)$$

These diagrams allow the visualization of the scalability trends of a program in a more dynamic and productive way because it is simpler and easier to focus the attention on a color variation than in the analysis of data presented in tables or line charts, as in Table I or Fig. 2. The PaScal Viewer presents the file name and the number of code lines that originated each diagrams set.

From the diagrams of Fig. 1, one can infer that: a) to any number of cores, when the problem size increases the efficiency values increase too, therefore, the program can be considered generally scalable; b) there is a limit to each problem size at which the increase of number of cores means efficiency improvement; increasing cores above this limit holds efficiency values constant; c) increasing the number of cores with any constant problem size decreases the efficiency values, therefore, the program is not strongly scalable; and d) there is no efficiency drop when the number of cores and the problem size increases at the same rate; therefore the program is weakly scalable.

B. Input File Format

The four diagrams of PaScaL Viewer are drawn from execution time measurements of a program. The tool does not measure the execution times. It reads them from an input file, translates them into efficiency values and then draw the diagrams.

The input file can be generated directly by a profiling tool, such as SPERF. SPERF is a simple profiling tool that allows the instrumentation of C/C++ programs and that exports the analysis data to .json, .xml and .csv file formats [5]. The input file can also be generated manually, or from another comfortable tool or script. It is a simple text file that contains data from all parts of the program, as shown in Fig 3 for the .json format. The data consists of the identification of each part analyzed, the number of cores, the problem size and the execution times of all execution scenarios. The data structure allows the inclusion of time measurements to various executions. The PaScaL Viewer uses the median of these values to compute the efficiency translated into the diagrams.

The data on the .json file are structured in arrays and objects according to the following understanding: a) the program can have various parallel regions; b) a parallel region can be executed one or many times with different input sizes; and c) for a specific input size, the program can be executed many times with different number of cores.

III. CASE STUDIES

In order to show the effectiveness of PaScaL Viewer, three specific applications from the PARSEC Benchmark Suite were used as a case studies. PARSEC is a suite of application for chip-multiprocessors that incorporates a selection of different combinations of parallel models and execution workloads. It is a package with applications that represent real computational challenges [8].

The applications chosen for these case studies were Blacksholes, Bodytrack and Freqmine. Two of these applications have inner parallel regions and therefore allow the analysis of the effect of their scalability on the efficiency trends of whole application. Freqmine is the one that has no inner regions. The applications were executed in a 32-core shared memory machine with 1, 2, 4, 8, 16 and 32 cores and with 10 distinct problem sizes: i1, i2, i3, i4, i5, i6, i7, i8, i9 and i10. The i2 problem size is twice as large as the i1 problem size; the i3

```

1 ▾ [
2 ▾   {
3     "region": "<initial_line>, <final_line>",
4     "filename": "<file_name>",
5     "executions": [
6     [
7     [
8         {
9         "argument": "<id_of_problem_size>",
10        "runs": [
11        { "threads": 1, "time": 100 },
12        { "threads": 2, "time": 51 },
13        { ... }
14        ]
15        },
16        {
17        "argument": "<id_of_problem_size>",
18        "runs": [
19        { "threads": 1, "time": 100 },
20        { "threads": 2, "time": 51 },
21        { ... }
22        ]
23        },
24        { ... }
25        ]
26    },
27    {
28        "region": "<initial_line>, <final_line>",
29        "filename": "<file_name>",
30        "executions": [ ... ]
31    }
32 ]

```

Fig. 3. The structure of the PaScaL Viewer input file.

problem size is twice the as large as i2 problem size, and so on. The SPERF tool was used to collect the execution time measurements and to generate the .json input file for the PaScaL Viewer.

A. The Blacksholes application

Blacksholes is an application from the financial domain analysis that solves a partial differential equation to calculate the prices of a given number of financial options. This application has just one inner parallel region.

The diagrams of Fig. 4 refer to the whole program and the diagrams of Fig. 5 describe the behavior of the inner parallel part. From the diagrams of Fig. 4, one can infer that: a) for any number of cores, the program presents better efficiency values for smaller problem sizes, and almost does not scale when the problem size increases; b) the program is not strong scalable in any scenario; and c) the program is not weakly scalable in any scenario.

The diagrams of Fig. 5 allow the following interpretation: a) the region presents better efficiency values for higher number of cores; b) the region is scalable because it does not present efficiency drop for increasing problem sizes; c) the increase in the number of cores does not improve the efficiency of the region, so it is not strongly scalable; and d) the region does not present weak scalability considering that increasing the number of cores and problem size proportionally does not improve the efficiency of the region.

B. The Bodytrack application

Bodytrack is an application of computer vision that tracks a human body from the analysis of an image sequence [8]. This application has three inner parallel regions.

The diagrams of Fig. 6 refers to the whole program. The diagrams of Fig. 7, Fig. 8 and Fig. 9 describe the behavior of three inner parallel regions of the program. From the diagrams of Fig. 6, one can realize a similar behavior to Blackscholes where: a) the program presents better efficiency values for smaller problem sizes; b) the program is not strong scalable; and c) the program is not weakly scalable. Although the two programs present resembling scalability trends, one can identify that the Bodytrack scale less than Blackscholes as the problem size increases.

The diagrams of Fig. 7 and Fig. 8 demonstrate similar scalability trends for the two analyzed regions. The analysis of these diagrams allow the following interpretation: a) for any number of cores, there is efficiency drop when the input size increases, with exception of i10 input size; b) the regions are scalable for just the i10 input size; c) the increases in the number of cores worsen the efficiency of the regions, so they are not strongly scalable; and d) the regions do not present weak scalability. From the number of cores and input sizes presented in this case study, one can not infer if for input sizes greater than i10 the scalability indexes will continue to increase.

The diagrams of Fig. 9 allows the following interpretation: a) for any number of cores, there is no clear improvement on the scalability trends; b) the region is not scalable because, in many cases, it presents efficiency drop when the problem sizes increases; c) it is not strongly scalable; and d) the region does not present weak scalability.

From the inner regions diagrams, one can infer that the efficiency drop of Bodytrack application to larger input sizes is related to the scalability of their inner parallel regions. In this case, the scalability of the whole program deteriorate influenced by its inner parts.

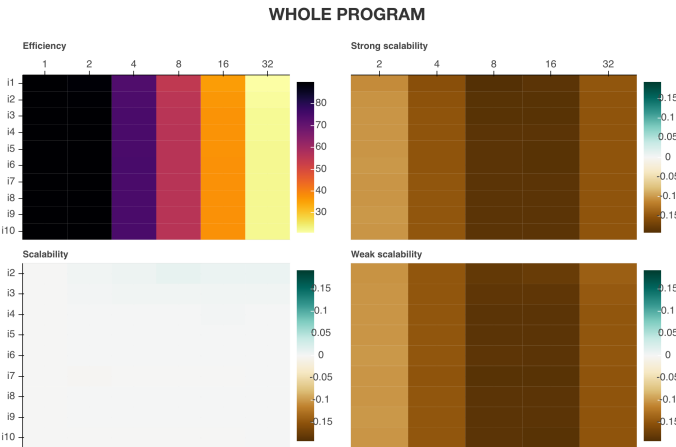


Fig. 4. Scalability diagrams for the whole Blackscholes application.

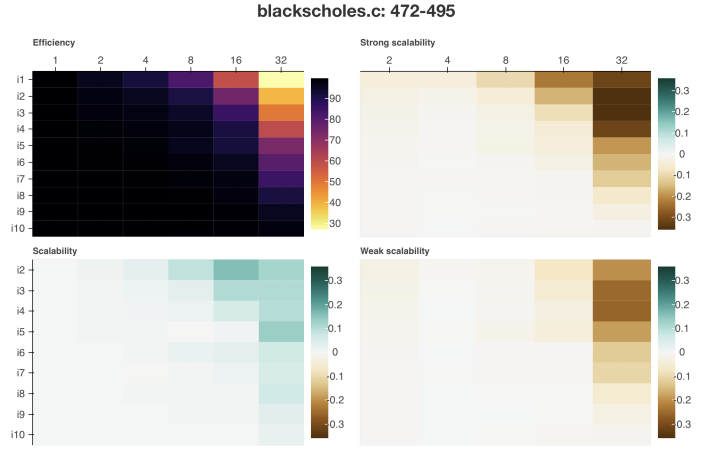


Fig. 5. Scalability diagrams for the inner parallel region of the Blackscholes application.

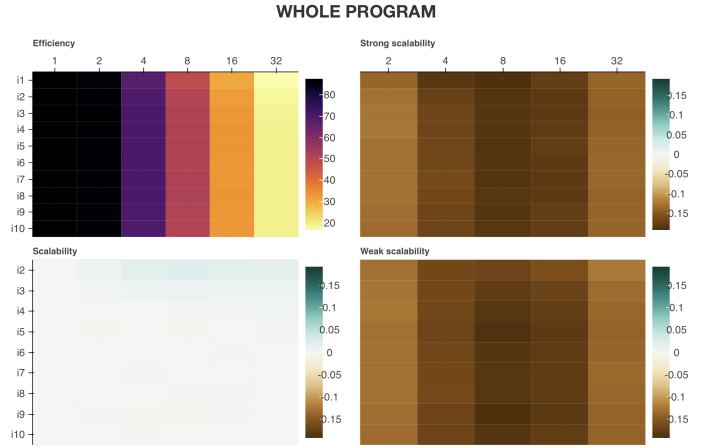


Fig. 6. Scalability diagrams for the whole Bodytrack application.

C. The Freqmine application

Freqmine is an application that identifies patterns in a transaction database through data mining techniques. It is an array-based version of the frequent pattern-growth method for frequent itemset mining.

The diagrams of Fig. 10 refer to the analysis of whole program and allow the following interpretation: a) the Freqmine program presents a continuously improving efficiency trend with better values for larger number of cores and input sizes; b) it is scalable because it does not present efficiency drop for any increasing problem size; c) the increase in the number of cores does not improve the efficiency of the program, so it is not strongly scalable, however, for smaller input sizes, as larger the number of cores the less is the loss of efficiency; and d) it tends to present weak scalability for larger problem numbers of cores and problem sizes although weak scalability could only be seen for input size around the i8.

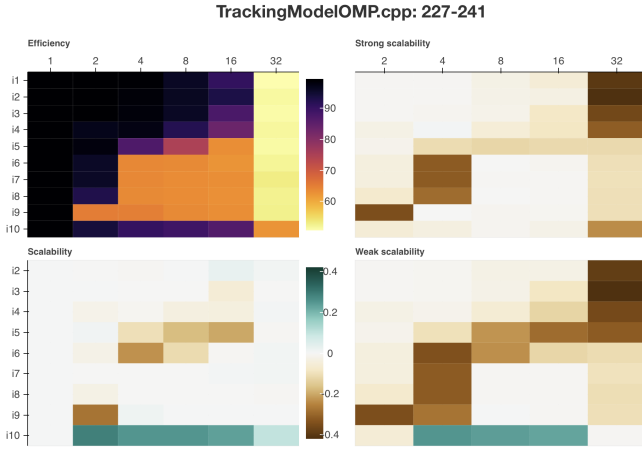


Fig. 7. Scalability diagrams for the first inner parallel region of the Bodytrack application.

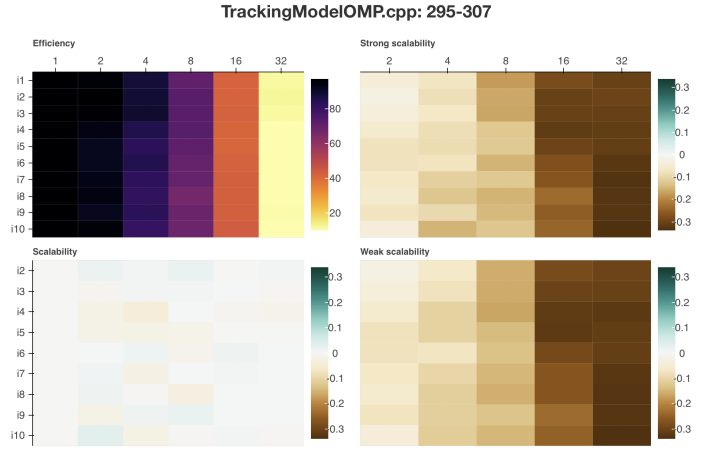


Fig. 9. Scalability diagrams for the third inner parallel region of the Bodytrack application.

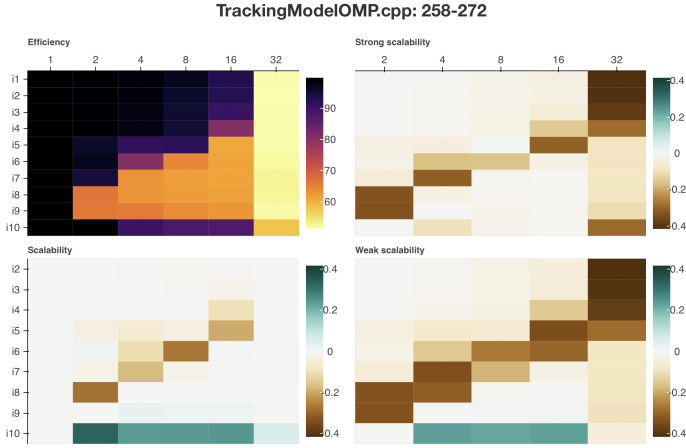


Fig. 8. Scalability diagrams for the second inner parallel region of the Bodytrack application.

IV. RELATED WORKS

Since Gprof [9], before the multi-core era, profilers work measuring the amount of routines calls and execution time of sequential programs. With the popularization of parallel architectures, the performance measurements became more sophisticated. Several tools emerged, initially with basic resources [10], [11], and then becoming more complex with advanced features and visualizations modes [12], [13], and, in general, focusing on performance metrics for large-scale parallel computer systems and applications [14], [15].

Instead of focusing on the profiling of a single run for the optimization of the program in a specific configuration environment, the PaScaL Viewer proposes a simple approach that mainly focus on the parallel efficiency of the applications. The objective is to present visual evidence of scalability trends. When targeting to specific regions of the code, this trend could also reveal scalability bottlenecks.

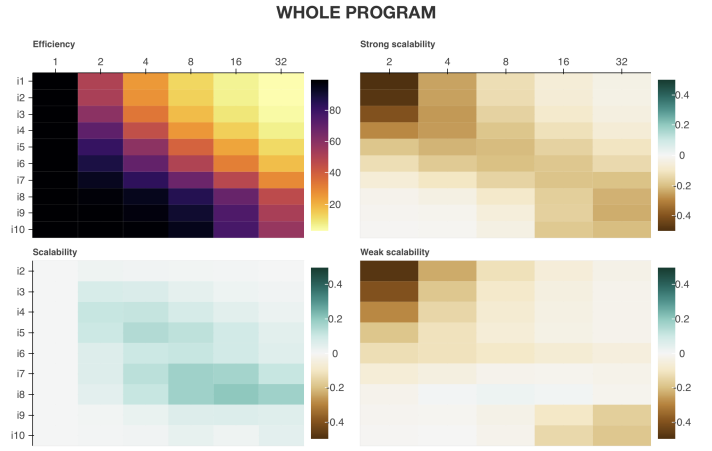


Fig. 10. Scalability diagrams for the whole Freqmine application.

V. CONCLUSION AND FUTURE WORKS

The PaScaL Viewer offers an efficient and productive way to analyze the scalability trends of a parallel program. From its four color diagrams, the tool simplifies the visualization of the program's parallel efficiency variation for multiple runs with various distinct configuration scenarios. It allows the identification of low scalability hot spots. That way, developers can focus their attention in these hot spots to optimize the program scalability.

As future work, the proposed tool will bring an interface that presents the diagrams of inner parts of a program hierarchically. This hierarchical view can help to identify more clearly how low scalability hot spots can impact the scalability of the whole program. Additionally, support to analyzing the scalability trends of finer parallel constructs like loops are also being investigated.

REFERENCES

- [1] H. T. Nguyen, L. Wei, A. Bhatle, T. Gamblin, D. Boehme, M. Schulz, K. L. Ma, and P. T. Bremer, "VIPACT: A visualization interface for analyzing calling context trees," *Proceedings of VPA 2016: 3rd*

Workshop on Visual Performance Analysis - Held in conjunction with SC 2016: The International Conference for High Performance Computing, Networking, Storage and Analysis, pp. 25–28, 2017.

- [2] A. Khamparia and J. S. Banu, “Program analysis with dynamic instrumentation Pin and performance tools,” *2013 IEEE International Conference on Emerging Trends in Computing, Communication and Nanotechnology, ICE-CCN 2013*, no. Iceccn, pp. 436–440, 2013.
- [3] J. Sairabanu, M. Rajasekhara Babu, A. Kar, and A. Basu, “A Survey of Performance Analysis Tools for OpenMP and MPI,” *Indian Journal of Science and Technology*, vol. 9, no. 43, 2016. [Online]. Available: <http://www.indjst.org/index.php/indjst/article/view/91712>
- [4] K. Huck and a.D. Malony, “PerfExplorer: A Performance Data Mining Framework For Large-Scale Parallel Computing,” *ACM/IEEE SC 2005 Conference (SC’05)*, 2005.
- [5] L. of Parallel Architectures for Signal Processing. Sperf 2.0. [Online]. Available: <https://gitlab.com/lappsufn/Sperf2.0>
- [6] D. S. Foundation. (2005) Django web framework. [Online]. Available: <https://www.djangoproject.com>
- [7] NumFOCUS. Bokeh. [Online]. Available: <https://bokehplots.com>
- [8] C. Bienia, S. Kumar, J. P. Singh, and K. Li, “The PARSEC benchmark suite: Characterization and architectural implications,” *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*, no. January, pp. 72–81, 2008.
- [9] S. L. Graham, P. B. Kessler, and M. K. McKusick, “Gprof: a Call Graph Execution Profiler,” in *SIGPLAN ’82 Proceedings of the 1982 SIGPLAN symposium on Compiler construction*. Boston, Massachusetts, USA: ACM, 1982, pp. 120–126.
- [10] W. E. Nagel, A. Arnold, M. Weber, H.-C. Hoppe, and K. Solchenbach, “VAMPIR: Visualization and Analysis of MPI Resources,” *Supercomputer*, vol. 63, no. 1, pp. 69–80, 1996.
- [11] V. Pillet, J. Labarta, T. Cortes, and S. Girona, “PARAVER: A Tool to Visualize and Analyze Parallel Code,” *Proceedings of WoTUG-18: Transputer and occam Developments*, no. February, pp. 17–31, 1995.
- [12] R. Bell, A. Malony, and S. Shende, “Paraprof: A portable, extensible, and scalable tool for parallel performance profile analysis,” *Euro-Par 2003 Parallel Processing*, pp. 17–26, 2003.
- [13] L. Adhianto, S. Banerjee, M. Fagan, M. Krentel, G. Marin, J. Mellor-Crummey, and N. R. Tallent, “HPCTOOLKIT: Tools for performance analysis of optimized parallel programs,” *Concurrency Computation Practice and Experience*, vol. 22, no. 6, pp. 685–701, 2010.
- [14] F. Wolf, E. Abrah, D. Becker, W. Frings, F. Karl, M. Geimer, S. Moore, M. Pfeifer, and B. J. N. Wylie, “Usage of the SCALASCA toolset for scalable performance analysis of large-scale parallel applications,” in *Proceedings of the 2nd International Workshop on Parallel Tools for High Performance Computing*, vol. 228. Stuttgart, Germany: Springer Berlin Heidelberg, 2008, pp. 157–167.
- [15] S. S. Shende and A. D. Malony, “The Tau Parallel Performance System,” *The International Journal of High Performance Computing Applications*, vol. 20, no. 2, pp. 287–311, 2006.