2021 IEEE/ACM International Workshop on Programming and Performance Visualization Tools (ProTools) | 978-1-6654-1110-3/21/$31.00 ©2021 IEEE | DOI: 10.1109/ProTools54808.2021.00006

TFProf: Profiling Large Taskflow Programs with Modern D3 and C++

Tsung-Wei Huang
tsung-wei.huang@utah.edu
Department of Electrical and Computer Engineering, University of Utah

***Abstract*—Task parallelism has emerged as an important tool to program heterogeneous resources that comprise manycore CPUs and GPUs. Among various tools that support task-parallel programming, *visualization* plays a key role in enabling developers to intuitively understand the execution profile of tasks and threads. However, as the complexity of parallel programs continues to increase, the need to efficiently visualize millions of tasks in an interactive environment has become the major bottleneck to developer's productivity. In this paper, we introduce *TFProf*, a web-based visualizer to assist developers to profile the execution of task-parallel programs in an easy-to-use browser interface. By leveraging modern D3 and C++ technology, TFProf can quickly visualize millions of tasks in a hierarchical level of detail. We have integrated TFProf into the popular task-parallel system, *Taskflow*, and demonstrated its practical use in large-scale parallel applications.**

I. Introduction

Recent years have seen a great deal amount of *task-based computing systems* (TCSs), such as oneTBB [1], StarPU [2], TPL [3], Legion [4], Kokkos [5], PaRSEC [6], HPX [7], and Fastflow [8], that aim to assist developers with the building of parallel and heterogeneous applications. Among various support tools, many TCSs provide *visualizers* for developers to intuitively inspect the execution of tasks using graphics libraries (e.g., OpenGL, Qt). However, visualization tools of existing TCSs are short of *dynamic* and *large-scale* rendering that allow developers to visualize multi-million tasks in an interactive environment. This shortage has largely limited developers' productivity in using existing TCSs [9].

In this paper, we introduce *TFProf*, a web-based visualizer to assist developers to intuitively understand the execution profile of task-parallel programs in an easy-to-use browser interface.[1] TFProf is built on top of our TCS, *Taskflow* [10], [11], that is being used by many academic and industrial projects.[2] By leveraging modern D3 [12] and C++ technology, TFProf efficiently visualizes millions of tasks in a hierarchical *level of detail* (LOD) and enables interactive analysis of task execution for improved user experience. Figure 1 shows an overview of TFProf for visualizing a Taskflow program. The top half shows the execution timeline of tasks in a selected window and the bottom half ranks these tasks in ascending order of their runtimes. We summarize our contributions as follows:

- We introduce a new browser-based visualization framework that combines the popular D3 JavaScript library and the powerful C++ programming language to render large task execution data at a multi-million scale.
- We introduce efficient clustering algorithms to explore task execution data in a hierarchical LOD that can be interactively rendered by existing browsers with silky-smooth animations.
- We have integrated TFProf into a real TCS, Taskflow, and demonstrated its capability in rendering millions of parallel tasks that are difficult to achieve using mainstream TCS visualization tools.

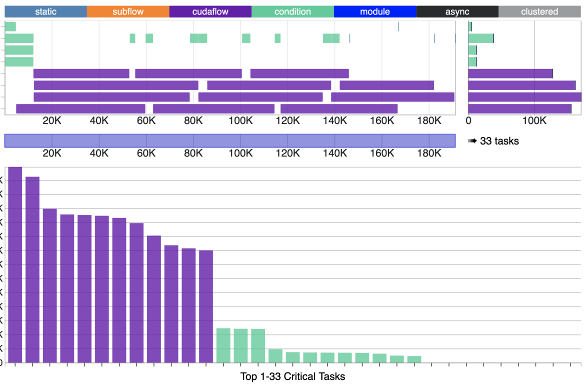


Fig. 1: An overview of TFProf for visualizing a machine-learning taskflow program [13]. TFProf leverages D3 and C++ to enable smooth visualization of large task-parallel programs in an easy-to-use browser interface.

We believe TFProf stands out as a unique visualization tool given the design trade-off and software decisions we have made. TFProf is open-source at [13] and [14] to facilitate intuitive support tools for debugging, performance profiling, and tuning of large-scale high-performance computing (HPC) applications.

II. Motivation

TFProf is motivated by our Taskflow project [14] to visualize large parallel programs that incorporate millions of tasks in a heterogeneous computing environment. Taskflow is a general-purpose parallel and heterogeneous C++ programming system using a *task graph*-based approach. In Taskflow, a

[1]TFProf website: https://taskflow.github.io/tfprof/
[2]Taskflow project website: https://taskflow.github.io/

task is the basic unit of computation executed by a *worker* thread on either a central processing unit (CPU) or a graphics processing unit (GPU). A worker is spawned by an *executor* to run dependent tasks in a *work-stealing* loop for dynamic load-balancing. Unlike existing TCSs that target small or medium task graphs, Taskflow is designed for large-scale parallel applications that spawn millions of CPU-GPU dependent tasks to compute irregular simulation and optimization workloads. Figure 2 shows a partial Taskflow graph that implements a VLSI placement optimization workload [15]. The complete graph has 1.7M tasks.

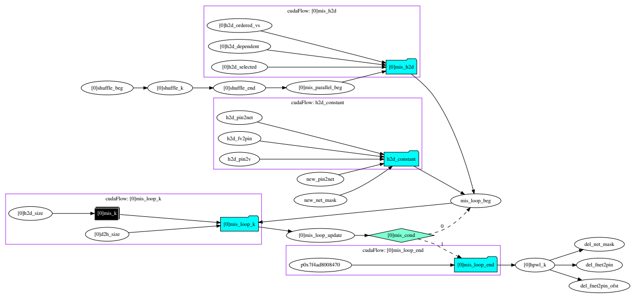


Fig. 2: A partial Taskflow graph for a VLSI placement optimization workload. The complete graph has 1.7M tasks.

While Taskflow can scale to millions of tasks, a key fundamental challenge has emerged: *How can we efficiently visualize a large number of tasks such that developers can intuitively understand the execution profile of tasks and threads?* To address this challenge, we have researched many visualization techniques in existing systems, such as the TBB FlowGraph Analyzer [1], Visual Trace Explorer (VTE) [16], Nvidia Visual Profiler (NVVP) [17], and so on. However, we found most of them fall short of our need due to three limitations: First, existing visualization tools are good at small- or medium-scale tasking rather than large numbers of tasks. For example, NVVP can take hours to visualize the execution timeline of 6K kernels. Second, most visualizers are static and do not support interactive analysis with smooth transitions. The latency between rendering different zoom sizes can be very high when the task count becomes large. Third, there exists a steep learning curve for users to visualize their parallel programs. There is no push-button solution that allows developers to quickly visualize their programs using an out-of-box interface in just a few minutes.

After years of research, we decided to leverage the popular web-based visualization library, *data-driven documents* (D3) [12], to tackle the limitations of existing visualizers in the HPC community. D3 is a JavaScript library for producing dynamic, interactive data visualizations in web browsers. It makes use of *scalable vector graphics* (SVG), HTML5, and cascading style sheets (CSS) standards. While D3 can improve user-side experience by making interactive visualization in an easy-to-use browser, it cannot process large data efficiently. Specifically, D3 maps each SVG element in a *document object model* (DOM) node and uses operators to manipulate them in a similar manner to jQuery. Most browsers can efficiently visualize 500–2000 DOM elements in the typical 60 frames per second (fps) budget. However, beyond 2000 elements, the rendering process significantly slows down and can easily cause the browser to hang forever. For instance, when drawing 1M elements, Firefox emits the error of allocation size overflow and Google Chrome hangs for several minutes. Figure 3 plots the D3 rendering time under different numbers of DOM elements we have measured in three major browsers, Google Chrome, Firefox, and Microsoft Internet Explorer (IE).

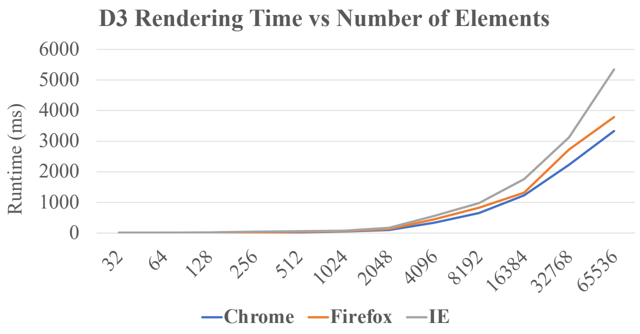


Fig. 3: Comparison of D3 rendering time between three browsers, Google Chrome, Firefox, and Internet Explorer (IE), at different numbers of DOM elements.

The most effective solution for rendering large data is presenting them hierarchically or with LOD. LOD is a concept that has been widely applied to various computer graphics applications, such as pyramid in image processing and mipmapping in 3D gaming. Regardless of the application, the concept is the same–*we squeeze large data into a smaller visual space by downsampling it to a lower resolution and display that instead.* If the number of elements is large, we display multiple levels of detail, each providing a higher level summary of the level below it. One familiar application of this technique is Google Maps [18], which has a database of hundreds of terabytes yet is capable of showing users us different sections of the globe in just a few milliseconds.

III. TFProf: Taskflow Profiler

This section introduces TFProf, a web-based visualizer to assist developers to profile the execution of a Taskflow program in an easy-to-use browser interface. TFProf has two modes, *client mode* and *server mode*, for visualizing small-scale and large-scales Taskflow programs, respectively.

A. Client Mode

Client mode of TFProf is completely written in JavaScript and takes input data in *JavaScript object notation* (JSON) format. Client mode is the easiest way to visualize a Taskflow program because it does not require a server to set up for processing task execution data. All Taskflow programs come with a lightweight profiling module to observer worker activities in every executor. To enable the profiler, users set the environment variable `TF_ENABLE_PROFILER` to a file name in which the profiling result will be stored. An example is shown in Listing 1.

```
~$ TF_ENABLE_PROFILER=result.json ./my_taskflow
~$ cat result.json
[{
  "executor": "0",
  "data": [{
      "worker": 12,
      "level": 0,
      "data": [
        {
          "span": [
            72,
            117
          ],
          "name": "12_0",
          "type": "static"
        },
        ... more task data
      ]
    ... more worker data
  }]
  ... more executor data
}]
```

Listing 1: Generate a JSON-based profile data from a Taskflow program. The JSON file defines three sections, *executor*, *worker*, and *task*, to describe the execution timeline of a Taskflow program.

The JSON output contains three sections of execution information for a Taskflow program, *executors*, *workers*, and *tasks*, in this order of hierarchy. The executor section contains the information of workers spawned from that executor and the worker section contains the time series data of task execution of that worker. Users can just past the output JSON data to the textbox of the client-mode TFProf in a browser and TFProf will render the data in pre-defined SVG regions using D3, as shown in Figure 1. In the main timeline chart, users can select a window (i.e., zoom-in) to visualize a particular range task execution timeline or double click the mouse to go back to the previously selected window (i.e., zoom-out), as shown in Figure 4 and Figure 5. The rendering between successive windows can be done with smooth transition. The bar chart in the bottom of the browser page will update the ranks of tasks based on a selected window. For example, the bar chart in Figure 4 sorts 33 tasks in ascending order of their execution times, and the bar chart in Figure 5 sorts 15 tasks in the selected window. Additionally, users can mouse over a task to open the tooltip and examine the statistics (e.g., type, name, runtime) of the task, as shown in Figure 6.

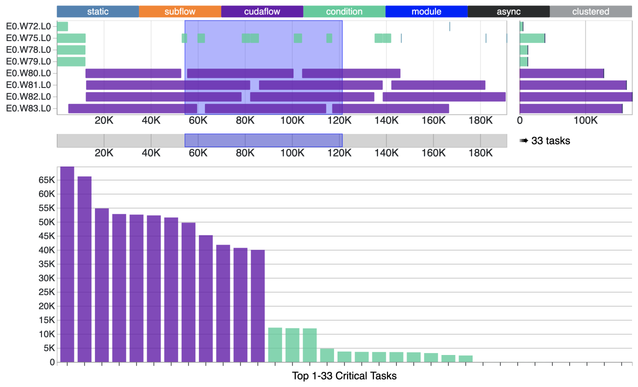


Fig. 4: Users can select a window of timeline (i.e., zoom-in) to visualize the task execution, as highlighted in the blue rectangle. Double clicking the mouse can go back to the previously selected window (i.e., zoom-out).

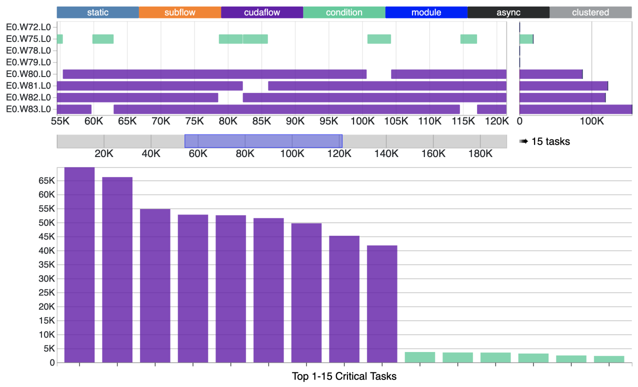


Fig. 5: Visualization of task execution in the selected window of Figure 4. The middle bar shows the overview of the selected window (marked in blue) in the overall execution length.

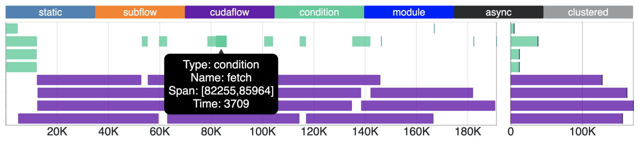


Fig. 6: Users can mouse over a task to see the statistics of the tasks, including task name, task runtime, and task type, in the opened tooltip.

B. Server Mode

Although the client-mode TFProf is very easy to use, it cannot present large data due to the inherent limitation of browsers and JavaScript. Table I compares the runtime of processing different JSON file sizes between mainstream browsers. Most browsers can process JSON files of 15–30 MB efficiently but they start slowing down significantly beyond 100 MB. In our case, large Taskflow programs can incorporate millions of tasks and add up to gigabytes of JSON files that cannot be handled by the client-mode TFProf. As an example, our Taskflow-enabled circuit timing analysis algorithm can spawn eight million tasks to propagate timing information through giant circuit networks, and the JSON file can be up to three gigabytes [19], [20], [21], [22].

To overcome this challenge, we have designed a *server-mode* TFProf that leverages the power of D3 and C++. On the front-end browser, we use D3 to visualize the selected dataset. On the back-end server, we use C++ to establish a lightweight database atop an HTTP server that communicates with the front-end D3 visualizer. When users apply a change to

TABLE I: Comparison of JSON processing time between mainstream browsers at different JSON file sizes

JSON Size	Chrome	Firefox	Safari	IE
150 MB	16.4 s	14.3 s	5.1 s	failed
115 MB	12.2 s	13.8 s	3.8 s	failed
76 MB	9.1 s	12.2 s	2.5 s	failed
38 MB	3.0 s	3.1 s	1.6 s	4.9s
15 MB	1.2 s	1.6 s	0.4 s	1.7s
7.8 MB	512 ms	1.1 s	243 ms	801 ms
3.9 MB	255 ms	646 ms	169 ms	408 ms

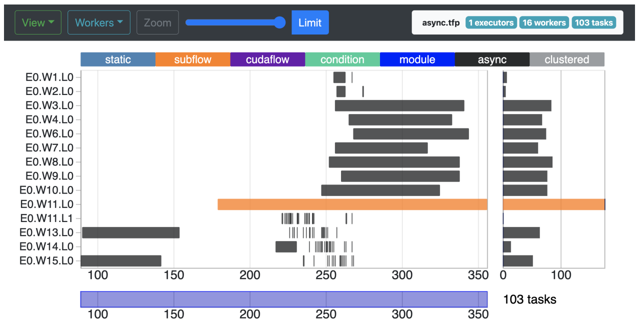


Fig. 7: Overview of server-mode TFProf, which is designed for visualizing large task execution data of millions of tasks.

the chart (e.g., select a new timeline range), the browser sends the request to the HTTP server and the server starts to process the data using powerful C++. When the C++ program finishes data processing, it sends back the result in JSON to the browser for visualization. Figure 7 gives an overview of the server-mode TFProf which contains more powerful toolboxes than the client mode for users to visualize large Taskflow programs. The most prominent difference is the view box, in which we introduce two LOD algorithms, *criticality view* and *cluster view*, to visualize millions of tasks hierarchically.

1) Criticality View: Criticality view lets users to select a range of tasks sorted by their runtimes. The maximum size of the range, k. is pre-defined to a value that can be fine-tuned for each browser to render with smooth transition. The default value is set to 1000 which allows D3 to visualize up to 1000 tasks with small latency on existing browsers. Criticality view visualize the top-k critical tasks in a selected timeline and allows users to understand which tasks take the most of the time. This is very useful and important for users to optimize most time-consuming tasks in a local range of their interest. Figure 8 shows a criticality view of a Fibonacci taskflow under a maximum rendering size of 512 tasks.

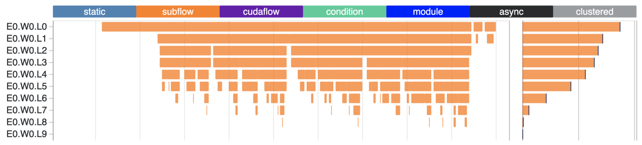


Fig. 8: Criticality view of a Fibonacci Taskflow program under a maximum rendering size of 512 tasks.

2) Cluster View: Cluster view lets users to select a range of tasks sorted by their runtimes. Since the selected range may contain many tasks that cannot be rendered efficiently by D3, we introduce a clustering algorithm to group adjacent tasks in ascending distance until the number of tasks and clustered tasks is below a maximum rendering threshold. The motivation of cluster view is to present high-level summary of task execution using suitable resolution, allowing users to see the scheduling maps in a global view. Then, users can select ranges of interest in a top-down fashion and gradually zoom in the selected ranges for more details. Figure 9 shows a cluster view of a Fibonacci taskflow under a maximum rendering size of 512 tasks. Similarly, Figure 9 shows a cluster view of the same program but under a limit of 200 tasks, and we can see more tasks are grouped together to reduce the number of rendered DOM nodes.

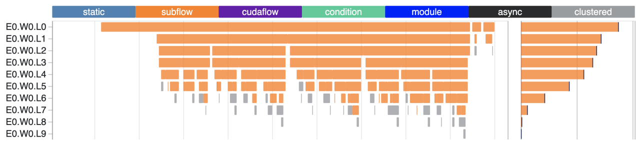


Fig. 9: Cluster view of a Fibonacci Taskflow program under a maximum rendering size of 512 tasks. Gray rectangles show clustered groups of tasks.

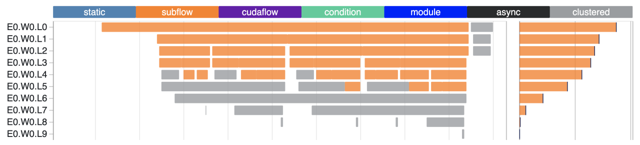


Fig. 10: Similar to Figure 9 but under a maximum rendering size of 200 tasks.

IV. Visualization Results

In this section, we present three large Taskflow programs and their visualization results using TFProf.

A. VLSI Placement

We applied Taskflow to solve a VLSI placement problem and use TFProf to visualize the task execution. The goal of VLSI placement is to determine the physical locations of cells (logic gates) in a fixed layout region using minimal interconnect wirelength. Modern placement typically incorporates hundreds of millions of cells and takes several hours to finish [15]. To reduce the long runtime, recent work started investigating new CPU-GPU algorithms. We consider a matching-based hybrid CPU-GPU placement refinement algorithm in ABCDPlace [15], that iterates the following (see Figure 11): (1) a GPU-based maximal independent set algorithm to identify cell candidates, (2) a CPU-based partition algorithm to cluster adjacent cells, and (3) a CPU-based bipartite matching algorithm to find the best permutation of cell locations. Each iteration contains overlapped CPU and

GPU tasks with nested conditions to decide the convergence. Figure 2 shows a partial taskflow graph of one iteration. A complete task graph can have up to 1.7 million CPU-GPU dependent tasks to place a million-gate design. Figure 12 shows the execution result in a cluster view. The rendering time for transitioning between successive windows is only about 10 ms.

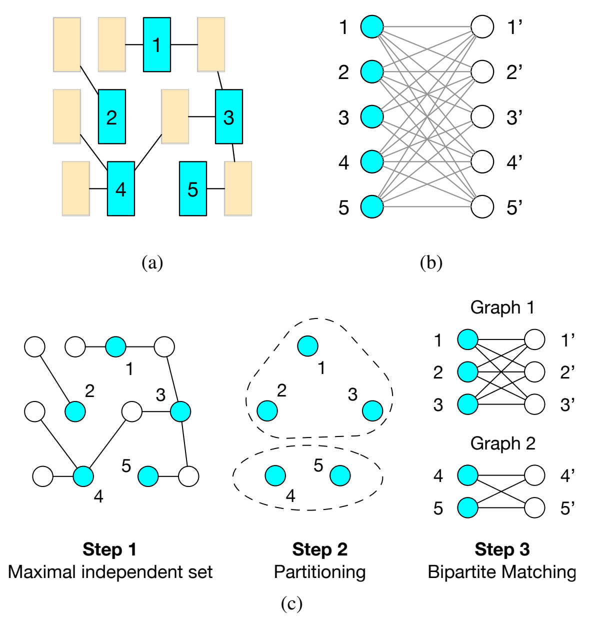


Fig. 11: A heterogeneous matching-based placement algorithm [15] that iterates the following three steps: Step 1 leverages a GPU-based maximum independent set algorithm to discover a set of independent nets. Step 2 leverages a CPU-based partition algorithm to cluster adjacent cells. Step 3 leverages a CPU-based bipartite matching algorithm to find the best permutation of cell locations with minimal wirelength.

B. Parallel Sort

We apply Taskflow to design a parallel-sort algorithm based on the quick sort algorithm and visualize the task execution. The taskflow graph spawns about 3M tasks to sort multiple large arrays of billions of elements. Figure 14 shows the task execution in cluster view, and Figure 15 shows the task execution in criticality view (top-512 critical tasks). Both views present the same program but in different views, allowing users to profile the task execution at different interest.

C. Parallel Iterations

We apply Taskflow to solve a matrix-multiplication workload using parallel-for tasks and visualize the task execution. The taskflow graph spawns about 5.1M tasks to multiply thousands of matrices in parallel. Figure 16 shows a cluster view of the execution result. The bar chart of the top-339 tasks in that selected window is shown in Figure 17. Figures 18–19 show two zoom-in views of Figure 16.

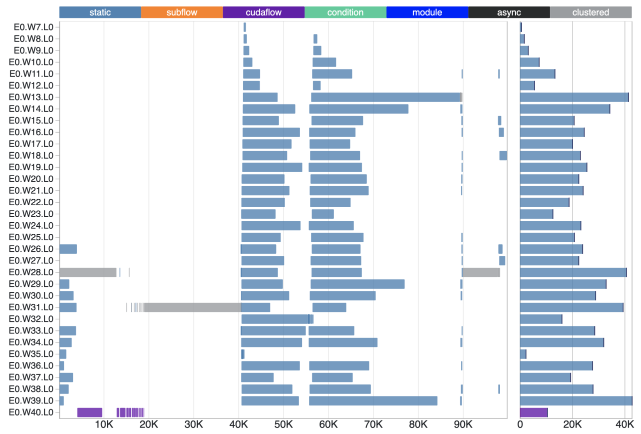


Fig. 12: Visualization result of a VLSI placement workload (partial taskflow graph shown in Figure 2) that incorporates 1.7M CPU-GPU dependent tasks [15]. The rendering time is about 981 ms for each selected range of tasks.

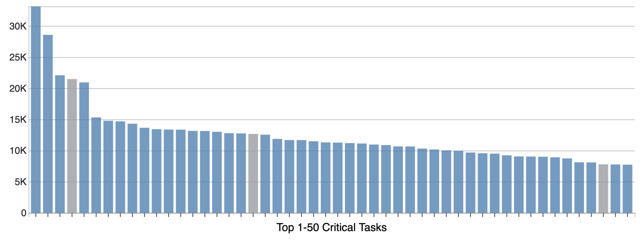
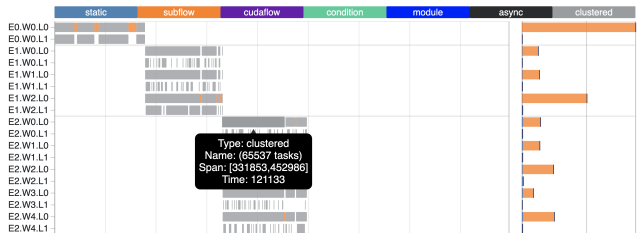


Fig. 13: Bar chart of the top-50 tasks in Figure 12.



Type: clustered
Name: (65537 tasks)
Span: [331853,452986]
Time: 121133

Fig. 14: Cluster view of a parallel sort taskflow graph that incorporates 2991763 tasks to sort multiple arrays of billions of elements. The rendering time is about 781 ms for each selected range of tasks.

V. Acknowledgements

The project is supported by the NSF Grant CCF-2126672 and the NumFOCUS Small Development Grant. We appreciate all contributors and users of Taskflow/TFProf for their valuable feedback to improve our system.

VI. Conclusion

In this paper, we have introduced TFProf, a scalable visualizer to assist developers to profile the execution of large Taskflow programs in an easy-to-use browser interface. By leveraging modern D3 and C++ technology, TFProf can

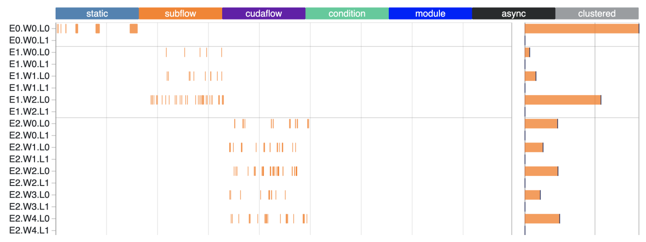


Fig. 15: Similar to Figure 14 but in the criticality view. The rendering time is about 431 ms for each selected range of tasks.

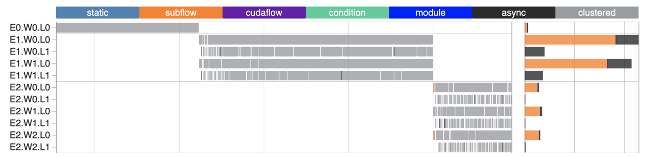


Fig. 16: Cluster view of the task execution result of a taskflow graph that incorporates 5.1M tasks to compute thousands of matrix multiplication instances in parallel. The rendering time is about 1.7 seconds for each selected range of tasks under a maximum rendering size of 512 tasks.

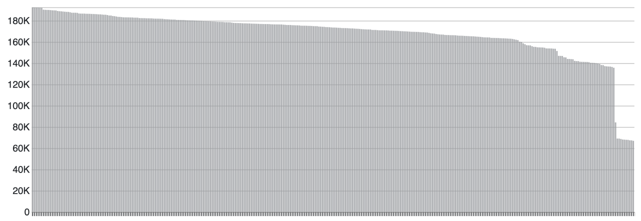


Fig. 17: Bar chart of the top-339 tasks in Figure 16.

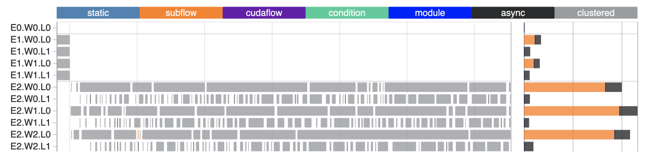


Fig. 18: Zoom-in of Figure 16. Most tasks are presented in clusters because the total number of tasks in the selected window remains large.

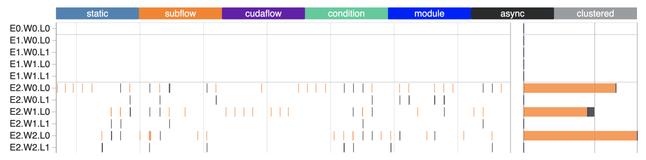


Fig. 19: Zoom-in of Figure 18. At this level of hierarchy, all tasks can be successfully rendered.

quickly visualize millions of tasks in a hierarchical level of detail. We have integrated TFProf into the tool chain of Taskflow and demonstrated its practical use in many large task-parallel programs that incorporate millions of tasks. Future work will enhance TFProf to visualize hierarhical task graph parallelism [23] and mapping the task graph to the scheudling result [24].

REFERENCES

[1] "Intel oneTBB," https://github.com/oneapi-src/oneTBB.

[2] C. Augonnet, S. Thibault, R. Namyst, and P.-A. Wacrenier, "StarPU: A Unified Platform for Task Scheduling on Heterogeneous Multicore Architectures," *Concurr. Comput. : Pract. Exper.*, vol. 23, no. 2, pp. 187–198, 2011.

[3] D. Leijen, W. Schulte, and S. Burckhardt, "The Design of a Task Parallel Library," in *ACM OOPSLA*, 2009, pp. 227–241.

[4] M. Bauer, S. Treichler, E. Slaughter, and A. Aiken, "Legion: Expressing locality and independence with logical regions," in *IEEE/ACM SC*, 2012, pp. 1–11.

[5] H. C. Edwards, C. R. Trott, and D. Sunderland, "Kokkos: Enabling manycore performance portability through polymorphic memory access patterns," *Journal of Parallel and Distributed Computing*, vol. 74, no. 12, pp. 3202 – 3216, 2014.

[6] G. Bosilca, A. Bouteiller, A. Danalis, M. Faverge, T. Herault, and J. J. Dongarra, "PaRSEC: Exploiting Heterogeneity to Enhance Scalability," *Computing in Science Engineering*, vol. 15, no. 6, pp. 36–45, 2013.

[7] H. Kaiser, T. Heller, B. Adelstein-Lelbach, A. Serio, and D. Fey, "HPX: A Task Based Programming Model in a Global Address Space," in *PGAS*, 2014, pp. 6:1–6:11.

[8] M. Aldinucci, M. Danelutto, P. Kilpatrick, and M. Torquati, *Fastflow: High-Level and Efficient Streaming on Multicore*. John Wiley and Sons, Ltd, 2017, ch. 13, pp. 261–280. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/9781119332015.ch13

[9] J. S. Vetter, R. Brightwell, M. Gokhale, P. McCormick, R. Ross, J. Shalf, K. Antypas, D. Donofrio, T. Humble, C. Schuman, B. Van Essen, S. Yoo, A. Aiken, D. Bernholdt, S. Byna, K. Cameron, F. Cappello, B. Chapman, A. Chien, M. Hall, R. Hartman-Baker, Z. Lan, M. Lang, J. Leidel, S. Li, R. Lucas, J. Mellor-Crummey, P. Peltz Jr., T. Peterka, M. Strout, and J. Wilke, "Extreme Heterogeneity 2018 - Productive Computational Science in the Era of Extreme Heterogeneity: Report for DOE ASCR Workshop on Extreme Heterogeneity," 2018.

[10] T.-W. Huang, C.-X. Lin, G. Guo, and M. Wong, "Cpp-Taskflow: Fast Task-based Parallel Programming using Modern C++," in *IEEE IPDPS*, 2019, pp. 974–983.

[11] T.-W. Huang, D.-L. Lin, C.-X. Lin, and Y. Lin, "Taskflow: A Lightweight Parallel and Heterogeneous Task Graph Computing System," in *IEEE TPDS*, 2021.

[12] "Data-Driven Documents (D3)," https://d3js.org/.

[13] "TFProf," https://taskflow.github.io/tfprof/.

[14] "Taskflow," https://taskflow.github.io/.

[15] Y. Lin, W. Li, J. Gu, H. Ren, B. Khailany, and D. Z. Pan, "ABCDPlace: Accelerated Batch-based Concurrent Detailed Placement on Multi-threaded CPUs and GPUs," *IEEE TCAD*, 2020.

[16] "Visual Trace Explorer," http://vite.gforge.inria.fr/.

[17] "Nvidia Visual Profiler," https://developer.nvidia.com/nvidia-visual-profiler.

[18] "GoogleMap," https://www.google.com/maps.

[19] T.-W. Huang and M. Wong, "OpenTimer: A high-performance timing analysis tool," in *IEEE/ACM ICCAD*, 2015, pp. 895–902.

[20] T.-W. Huang, G. Guo, C.-X. Lin, and M. Wong, "OpenTimer 2.0: A New Parallel Incremental Timing Analysis Engine," *IEEE TCAD*, vol. 40, no. 4, pp. 776–789, 2021.

[21] Z. Guo, T.-W. Huang, and Y. Lin, "GPU-accelerated Static Timing Analysis," in *IEEE/ACM ICCAD*, 2020, pp. 1–8.

[22] T.-W. Huang, C.-X. Lin, and M. D. F. Wong, "Opentimer v2: A parallel incremental timing analysis engine," *IEEE Design and Test*, vol. 38, no. 2, pp. 62–68, 2021.

[23] C.-X. Lin, T.-W. Huang, G. Guo, and M. Wong, "An Efficient and Composable Parallel Task Programming Library," in *IEEE HPEC*, 2019, pp. 1–7.

[24] C.-X. Lin, T.-W. Huang, and M. D. F. Wong, "An efficient work-stealing scheduler for task dependency graph," in *2020 IEEE International Conference on Parallel and Distributed Systems (ICPADS)*, 2020, pp. 64–71.