Survey of Performance Visualization

Wei Xu

Introduction

Optimizing parallel applications now rely on the performance of a number of hardware, software and application-specific aspects, such as multicore clusters, programmable graphics processing units (GPUs), multiple hierarchical memory access, network, and so on. The captured performance evaluation data is usually multivariate and heterogeneous, which makes the exploration and understanding extremely difficult. Visualization, as an indispensable domain for big data, has the capability to fusion the evaluation data, provide corresponding visual representations for exploration, and create effective user interaction and steering.

Performance visualization is a specific technique focusing on performance data of heavy computation applications. The data is acquired through instrumentation of a program, or monitoring system-wide performance information. In this survey, we focus on the techniques to visualize the instrumented performance data such as traces, profiles, counters and call paths, where the measurement records the parallel computation of a program in multi-core clusters.

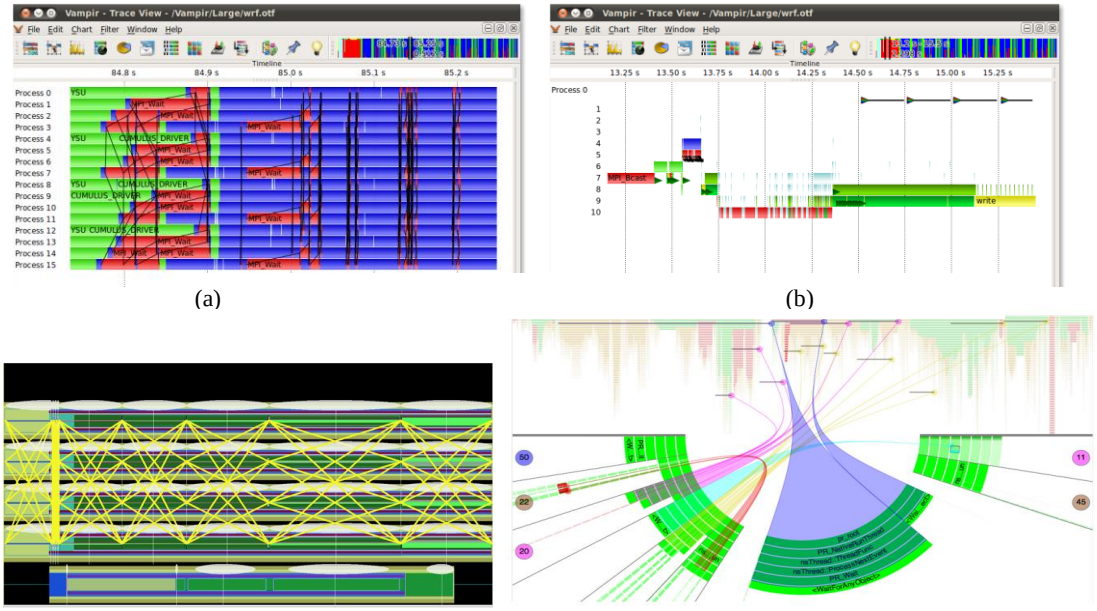

Fig. 1: trace timeline visualization examples: (a) Vampir timeline showing the execution on all processes, (b) Vampir timeline for one process with detailed function entry and exit, (c) the timeline of Jumpshot, and (d) the advanced visualization for focused thread comparison [3].

Existing approaches

The general purpose of performance evaluation includes: the global comprehension, problem detection and diagnosis [1]. Performance visualization is therefore designated to fulfill these goals. In specific, firstly, the design of the visualization must be able to show the big picture of the program execution. When interested area is targeted, users must narrow down the region and

mine more detailed information. Moreover, comparative study looking for correlation or dependency must be supported. For problem detection, abnormal behaviors can be highlighted in color that allows users to capture easily.

Current visualization works can be grouped by their applications in four contexts: hardware, software, tasks and application [1]. In our project, the acquired data are individual trace and profile files capturing the execution of independent workflow components. In specific, it includes a few types of data: 1) event table summarizing the start and end time of all function calls, 2) the message passing among threads, 3) profiling of certain metrics spent in each part of the code on each computing thread, and 4) the call path for each thread. Therefore, we only summarize the existing works that are commonly applied to our data types, while ignore other works such as the visualization for network, system memory usage, or system logs for multicore clusters.

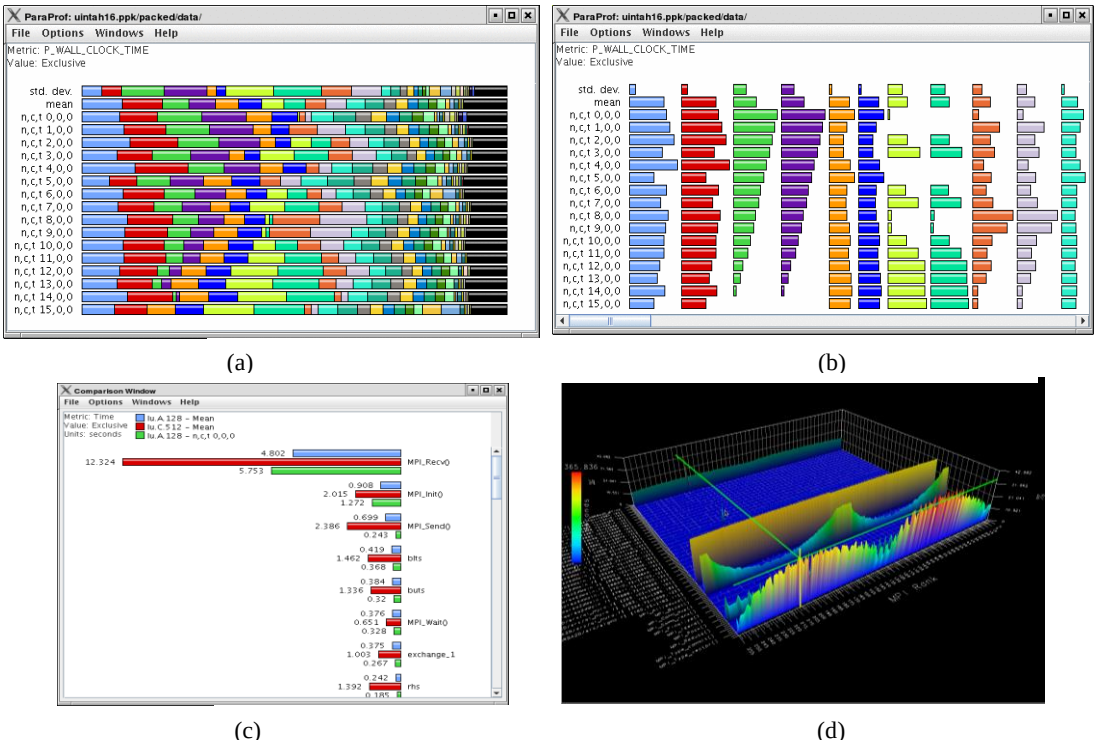

Fig. 2: Profile visualization examples: (a) ParaProf showing the profile of all functions in stacked view, (b) separated view, and the comparative view of different execution runs; (d) the 3D visualization comparing different metrics.

Trace timeline visualization

Trace records a sequence of timestamped events such as the entry and exit of function calls or a region of code, the message passing among threads, and job initiation of an entire run. A common practice is to assign the horizontal axis to the time variable, and the vertical axis to the computation processes or threads. Different approaches are usually variations of *Gantt charts*. Vampir[4] and Jumpshot [6] provide two examples of this kind of visualization, such as shown in Fig. 1. Generally, overview of the whole time period is first plotted. Then users can select interested area to reveal more detailed events happened during the selected period. Different functions or regions of code are colorized, and the black (yellow for Jumpshot) lines indicate message passing.

Besides, advanced visualization such as SyncTrace [3] provided a focus view showing multiple threads as sectors of a circle. The relationships between threads are shown with aggregated edges similar to chord diagram.

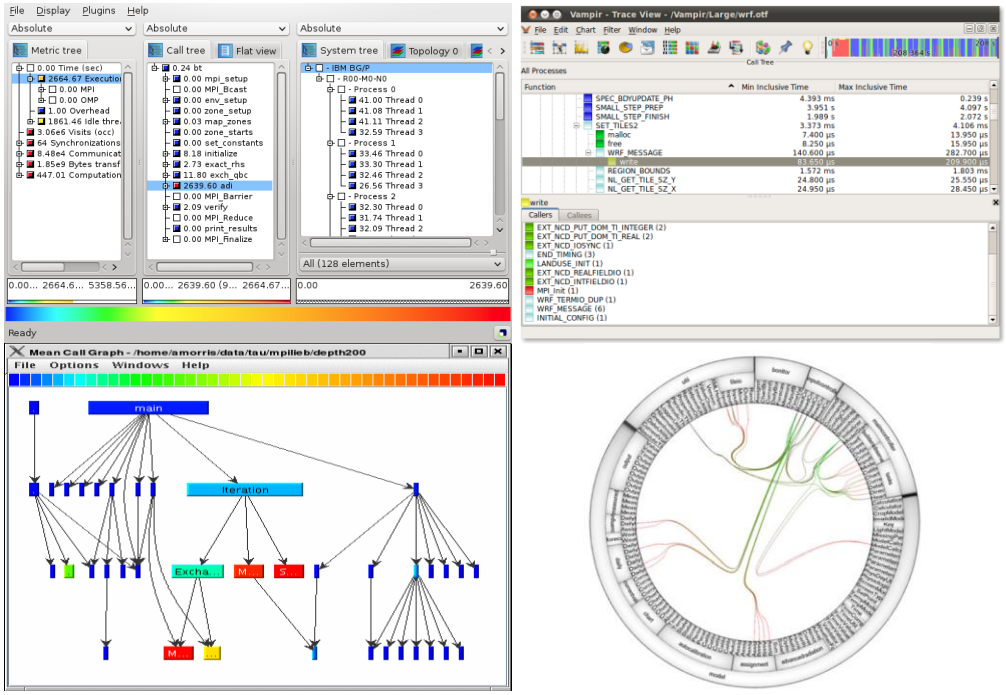

Fig. 3: Call path and call graph visualization examples: the call tree structure in CUBE (top left) and Vampir (top right), the call graph in ParaProf (bottom left), and the circular layout for caller-callee relationship (bottom right).

Profile visualization

Profile measures the percentage of time spent in each part of the code. Profile doesn't include temporal information, but can quickly identify key bottle necks in a program. Stacked bar charts, histogram, and advanced visualization in 3D are commonly used to give a comparative view of the percentage of time or other metric spent for different functions. ParaProf[ref] is one example of this kind of visualization as shown in Fig. 2. It also supports the comparison of certain function calls in different execution runs. The functions are color coded and plotted in different stacking modes. Other statistics can also be plotted for a selected function over all threads or for a selected metric correspondingly.

Call paths and Call graphs

Through profiling, the call paths information is also included. The percentage of time spent in each call path can be illustrated. The caller and callee relationship can also be identified. The common approach to visualize this kind of data is tree structure that utilizes node-link metaphor, or indented tree to preserve collapsible hierarchies. In Vampir, ParaProf and Cube [7], we can see the examples of such approach, shown in Fig. 3. Except that, a circular layout similar to sunburst is also useful to illustrate the caller-callee relationship [2] as shown in Fig. 3. The edge is bundled to avoid clutter. The structural dependency is thus easier to capture. The color is used to encode call direction as well as call time.

Message communication

As mentioned in the timeline visualization, message passing is also important. A straightforward approach is to draw a line between two functions for each message, as how Vampir and Jumpshot implement. On the other hand, the message communication between threads or processes can also be summarized in terms of a matrix, with proper colorization indicating additional information as shown in Fig. 4.

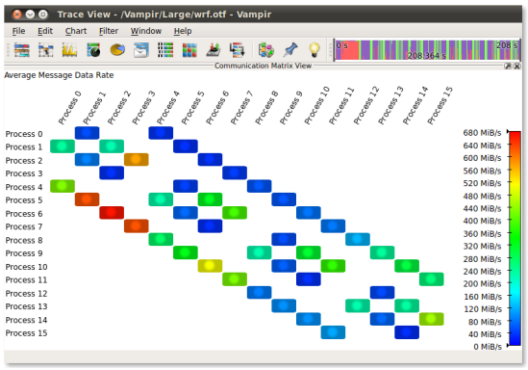

Fig. 4: Message communication: the message matrix in Vampir.

Gaps

For this CODAR project, we aim to visualize the measurements for workflows of various applications. However, existing tools are mostly for single application execution, which is not in the form of workflow composing a number of serial or parallel executions. Thus they lack the capability to illustrate the workflow structure and the I/O and other metadata between workflow components.

Besides, another major issue of existing works is the challenge for online performance evaluation. This not just requires the streaming visualization updates, but also relies on the online data reduction and sampling mechanism. Most existing works are designed for offline analysis. Although the visual representations are still effective for online evaluation, they must be adjusted to accommodate the streaming fashion so as to incrementally update and visualize data.

Our approach

The purpose of our tool is not to cover each detailed function of existing tools such as Vampire, ParaProf or Jumpshot. But instead, we aim to enable what is missing: 1) the capability to visualize and analyze the performance of the workflow execution for multiple applications, especially for serial workflows; and 2) starting from offline and finally reaching the goal for online evaluation.

In order to connect all required data together into the form of a workflow, we devise a new data structure – the data model for each workflow that includes the following information: 1) the overall structural description of the workflow, 2) the data flow and other metadata about the workflow, 3) the connected timeline of functional calls of the workflow, and 4) the associated profiling of the workflow.

Thus, to explore all the above information, we devise a level-of-detail multi-channel visualization framework with front-end plotting the data and back-end performing necessary data sampling, reduction, analysis and other light weighted computation. In specific, we utilize Gantt chart and stacked graph to represent events in terms of timeline as most existing works do. We also connect that to profile in the form of stacked bar graphs to reveal the corresponding statistics of the chosen metric. Then we implement Sankey diagram to show data I/O between components. Finally, we redesign the Chord diagram to support message communications between threads.

Additionally, other advanced techniques such as heatmaps and sunbursts are also considered to visualize other useful information in aggregated representations. A mockup view of our design is shown in Fig. 5.

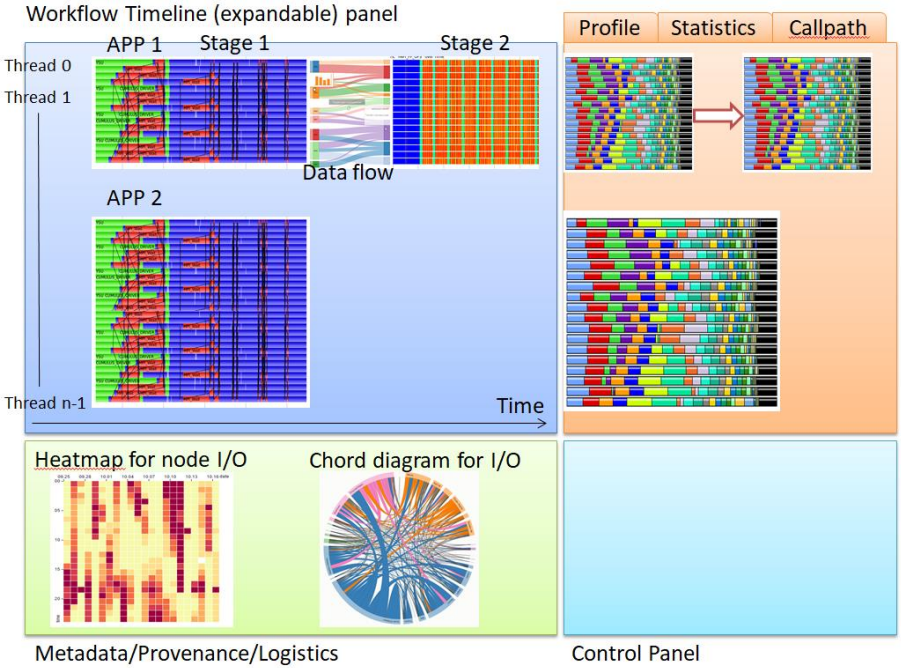

Fig. 5: The mockup view of our performance visualization interface.

In the offline mode, the data can be loaded directly from file system. In contrast, for the online mode, the performance data will come from intermediate interface such as ADIOS directly. Query based sampling and data reduction will be performed on the backend triggering by the interaction in the frontend.

Reference

[1] K. Isaacs, A. Gimenez, et al., “State of the Art of Performance Visualization,” Eurographics Conf. on Visualization 2014.

[2] CORNELISSEN B., HOLTEN D., ZAIDMAN A., MOONEN L., VAN WIJK J. J., VAN DEURSEN A.: Understanding execution traces using massive sequence and circular bundle views. In Proceedings of the 15th IEEE International Conference on Program Comprehension (Washington, DC, USA, 2007), ICPC ’07, IEEE Computer Society, pp. 49–58.

[3] KARRAN B., TRÃIJMPER J., DÃULLNER ″ J.: Synctrace: Visual thread-interplay analysis. In Proceedings (electronic) of the 1st Working Conference on Software Visualization (VISSOFT) (2013), IEEE Computer Society, p. 10.

[4] Vampir: https://www.vampir.eu/tutorial/manual

[5] ParaProf: https://www.cs.uoregon.edu/research/tau/docs/newguide/bk01pt02.html

[6] Jumpshot: https://www.cs.uoregon.edu/research/tau/docs/newguide/bk01ch04s03.html

[7] CUBE: http://apps.fz-juelich.de/scalasca/releases/cube/4.3/docs/manual/userguide.html