Visualization and Analysis of Parallel Dataflow Execution with Smart Traces

Daniel K. Osmari, Huy T. Vo, Cláudio T. Silva
Polytechnic School of Engineering
New York University

João L. D. Comba
Instituto de Informática
UFRGS

Lauro Lins
AT&T Research

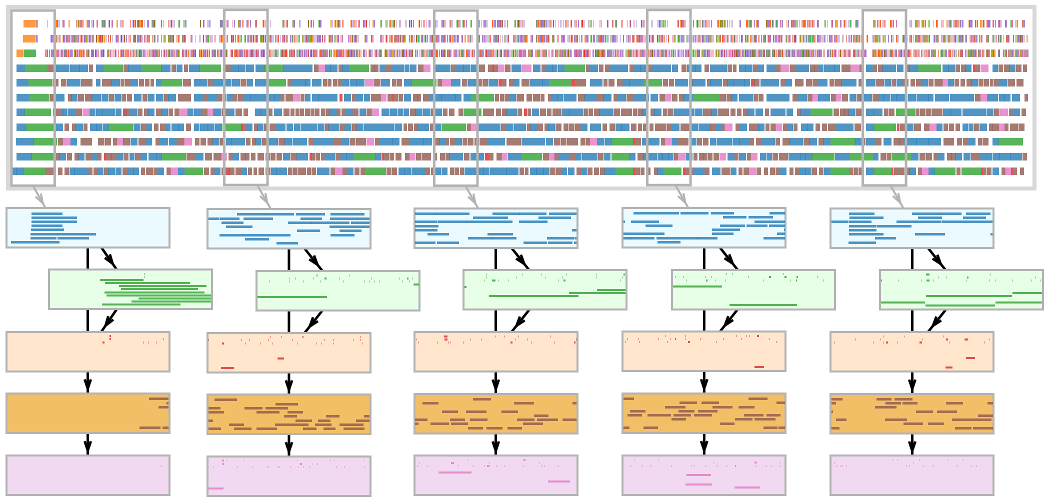

Fig. 1. Smart Trace visualization for an edge detection dataflow: (top) Gantt chart displaying a parallel trace collected with time-varying state data for each execution unit (top three rows are GPU devices, bottom 3 rows are CPU threads). The colors in the Gantt chart are associated to modules of the pipeline. (bottom) display of five intervals of the Gantt chart inside each module of the dataflow, thus allowing the performance of each module to be analysed separately.

***Abstract*—Most performance analysis tools focus on presenting an overload of details, with little application-dependent structure, and predefined statistical summaries. This makes the complex relations present in a parallel program not directly recognizable to the user, making the task of identifying performance issues more costly in both time and effort. In this work we investigate the requirements to create visualizations of execution traces of parallel programs modeled as dataflows. We propose the Smart Trace (ST) concept, to encode the structure of the data, and guide the construction of specialized visualizations. A visualization tool can then leverage the relationships in the data to automate a given analysis task. We show with examples the power and flexibility of visualizations we can create to address specific questions formulated about the analysis of the data, with emphasis in parallel dataflow traces.**

***Keywords*-Trace visualization; Parallel computing.**

I. Introduction

Parallel computation in its many different forms is essential to improve performance of computational tasks. Such computation power was the target of extensive research in the past, where seminal work in parallel computation was developed. However, its widespread use only become common recently, with the rise of multi-core CPUs, many-core GPUs, and computer clusters at affordable prices. Each of these configurations have aspects that impact performance, and leveraging the benefits of parallel computation is still a challenging task.

An important aspect is the post-analysis of computational traces generated upon execution of a parallel program. A parallel trace records a time-series of information about resource utilization and activation procedures during program execution. Trace analysis was discussed in [1]–[4], which identified the value of visualization to help understand the complex interplay of information in trace data. For this purpose a Gantt chart is often used, which corresponds to a 2D time-series graph that displays in which processing unit a task is executing in a given instance of time. Information visualization techniques developed along the years and the diversity of parallel computation power available today allows the visualization and analysis of parallel traces to be revisited.

In this work we introduce *Smart Traces* (ST), a new concept that aims at producing insightful visualizations of parallel trace data generated in dataflow systems. We used STs to evaluate *Hyperflow* [5], a parallel dataflow framework developed for leveraging the parallel power of heterogeneous systems. STs was designed to address several challenges raised by this analysis. Trace data encodes complex relationships among computational resources in time (e.g. processors, tasks, memory transfers, etc), which need to be grouped and inspected in different ways, sometimes in isolated visualizations, but most frequently as coupled visualizations. We designed a trace representation augmented with dataflow semantics to allows trace visualization and analysis. Furthermore, while it is easy to visually accommodate the individual of trace data for dozens of processing units, the growing number of parallel execution units available increases the amount of information to be displayed. Therefore, strategies to collapse time-series data into meaningful representations are required (e.g. Theme Rivers [6], Stacked Graphs [7], Edge Bundles [8] or Word Lines [9]). An example of ST is given in Figure 1.

The main contributions of this work are outlined below:

- A trace format augmented with dataflow information suitable for mapping information visualization techniques;
- The concept of STs, which correspond to coupled views of dataflow trace data;
- A predefined set of STs that allow immediate visualizations to be constructed directly from trace data.

II. Related Work

We summarize in different sections the literature that discuss several aspects related to parallel trace visualization.

Visualization of Parallel Systems

The *Paragraph* system [1] was one of the earliest work to identify the importance of visualization techniques to understand trace information in parallel systems. It raised the importance of creating an animation that reflects the dynamic behavior of trace, communication, task, and application-specific displays. To our knowledge, it is the first work that uses Gantt charts [10] to visualize time-series data of resource against task usage. In [2], [3] the ideas of Paragraph are expanded, with emphasis in user interface aspects that allow for multiple views of the data. In [4] it is discussed design goals for trace visualization. Message-passing protocols led to specifically-designed visualizations to understand the performance of such protocols. The *Viper* [11] system uses a standard set of views to inspect a given aspect of the parallel program state: animation, space-time or variables view. A system with focus more into data-parallel visualization than performance is described in [12], with emphasis on a system that could handle traces interactively. MPI trace visualization is first discussed in [13] and further elaborated in [14]–[16]. Common to these work is the use of Gantt charts to evaluate thread and CPU activities. *Tracevis* [17] designed search criteria to inspect trace data in certain regions, as well the ability to annotate traces with persistent information. The *Vampir NG* system [18] offers a client-server solution coupled with a compressed data structure to handle large trace data. Augmenting trace information with meta-data is also described in [19] with the purpose of obtaining analytical performance models of MPI programs. *Data Flow Tomography* [20] is a system that uses virtual machines with special instructions to track data movement and usage. The *DIMVisual* model [21] allows the integration of trace data in a distributed system, with subsequent treemap [22] and 3D visualization [23] approaches for grid monitoring and its respective network topology. In [24] Gantt charts are augmented with links between tasks to show interconnections in parallel applications. In [25] function calls are presented on Gantt charts with discretization and non-linear scaling of time, to facilitated identifying functions through visual patterns.

Visualization Techniques for Time-series Data

Summary graphs, such as histograms of function call counts or pie chart distribution of time spent in each module, are used in the visualization of trace data. Gantt charts [10] offer a display of a time-series of information with individual processor utilization, but becomes too dense when the number of resources is too large There are several techniques that address more compact or more expressive visualization of time series data. Approaches like Theme Rivers [6] or Stacked Graphs [7] compact information using the semantic data of the time-series to adjust the thickness of the stack with respect to a given baseline. Hierarchical Edge Bundles [26] apply the idea of edge bundles [8] to compact trace information, such as activation call relations. The graphical perception of different time-series visualization is described in [27], with pros and cons of each method with respect to certain tasks.

Coordinated and Multiple Views

Filtering, grouping and summarizing allow for compacting information into a smaller display area while still showing relevant information. However, to capture the different aspects of data, in particular the complex interplay of trace information, it becomes necessary to combine multiple visualizations into a common linked (or coupled) view [28]–[30]. Recent work in the area include Matchmaker [31], which groups multi-dimensional datasets, Word Lines [9], which control multiple simulation runs, and behaviorism [32], which create visualizations for dynamic data and its interconnections.

The ST implementation relies on a collection of interactive information visualization Python scripts implementing coordinated views, allowing the user to directly interact with the data to refine the visualizations, either by zooming in and inspecting values, or applying different visualization techniques. More details are given in the supplementary material.

III. Parallel Dataflow Instrumentation

We define below the characteristics of a parallel dataflow system, and how the execution trace data is collected.

A. Parallel Dataflow Model

A dataflow is a directed graph where each node corresponds to a module whose execution depends only on its inputs, and interaction between modules happens by data exchange. Each input parameter and output result in a module is denominated a “port”. Output ports are connected to input ports, defining the topology of the pipeline of the dataflow as edges of the graph. An input triggers a series of computations and data communication down the pipeline.

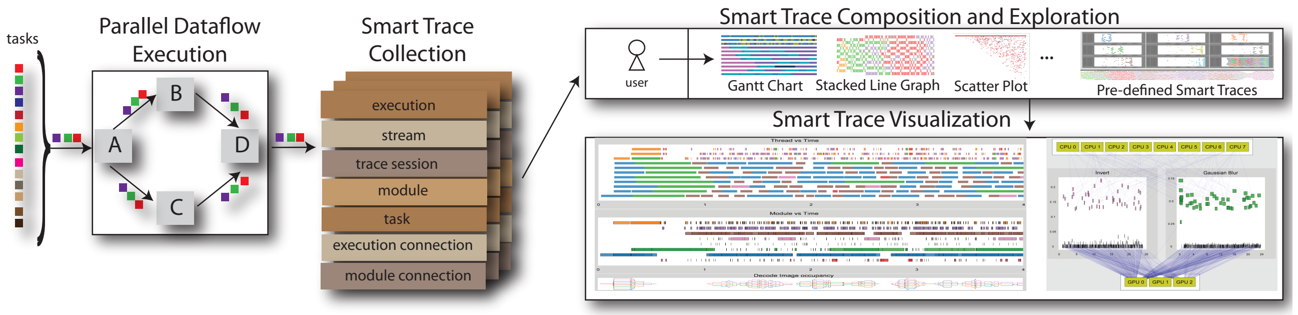

Fig. 2. ST Architecture. A parallel dataflow executes multiple tasks, and the trace execution is collected into a ST collection, which consists of trace data augmented with dataflow information. The ST exploration relies on visualization tools or predefined constructions to define a view of the trace collection.

In a sequential execution environment the dataflow is topologically sorted, guaranteeing that a module is executed only after its dependencies have executed. In parallel systems it is desirable to have multiple threads or processes to execute modules in parallel. There are many approaches for implementing a parallel dataflow architecture, but they require non-trivial coordination among threads and processes to make optimal use of resources. This coordination is usually performed by a scheduler, responsible for assigning tasks to modules, and for starting their execution. The scheduler can offer some guarantees, such as order of execution, real-time constraints, maximum number of resources to be allocated, etc, at the cost of a more complex and time-consuming implementation.

B. Code Instrumentation

The execution trace consists of time-stamp events collected by code instrumentation at specific points of interest. Code instrumentation can be performed at different levels-of-detail; in our analysis we concentrate on events related to dataflow execution. Automated tools for code instrumentation (e.g. DTrace [33], Vampir [34], TAU [35]) offer a more fine-grained and low-level collection than desired, which requires a filtering step to ignore uninteresting events, while being limited in the information that can be collected. Therefore, we decided for creating a customized code instrumentation that can specifically address important events during dataflow execution. Our analysis focuses on three types of events:

• **Execution**: represents the time a thread spent executing a unit of work, which can be either a function in the code, or a pipeline module in a dataflow. The execution is delimited by two events, marking the beginning and end of execution;

• **Variable update**: represents any variable with value updated during execution. It's recorded by a single event containing the variable identifier and its new value;

• **Communication**: represents data dependencies or synchronization operations, represented by send/receive event pairs. It is often difficult to register the entities involved in the message exchange without actually changing the data being transmitted.

The instrumentation can be implemented entirely on the dataflow framework, hidden from the application code. We collect meta-data describing the dataflow network. Data transfers between modules can be encoded as send/receive events, since every event references the task that produced it.

IV. Smart Traces

In this section we introduce the *ST* abstraction and describe how it helps the user in creating customized visualizations of trace data. The concept is implemented in interactive Python scripts, making full use of its dynamic environment, where code and data can be created and updated at any time to create linked visualizations. The process of creating a ST visualization proceeds as illustrated in Figure 2. As the execution trace of a parallel dataflow is recorded using code instrumentation, trace data is augmented with dataflow information, resulting in what we call a ST collection. This data is imported into an interactive application where the user can analyze the data by creating views of the data, called STs.

A. Smart Trace Collection

A ST collection consists of the trace data enriched with derived entities that interpret the meta-data. Each entity is instanced as a Python object that presents the derived relationships and aggregations explicitly, making the data exploration more convenient. The entities are:

• **State change:** actual raw data explicitly recorded in the execution trace. In addition to start and finish times for each module execution, it references the computation unit where it was executed (stream), predecessor and successor events according to both the pipeline and the stream, task being processed, and also application-specific information;

• **Stream:** associated to a system thread, process, or CPU. Contains start and finish times, references to events executed underneath it, and the trace session;

• **Trace Session:** global execution of the program, containing per-application metadata;

• **Task:** execution of one input across the whole pipeline. It contains attributes such as start and finish times, as well as any application-specific attributes describing the initial input and final output of the pipeline;

• **Module:** one pipeline stage, with references to its predecessors and successors in the pipeline. It references all state changes inside the module, with statistic summaries;

• **State change connection:** data transfer between two modules during the execution of a specific task. It contains links information about the difference in time between the end of execution of previous module and the start of the next one;
• **Module connection:** connectivity between two modules and corresponding state change connections (one for each task that passed through that module connection). This is used to collect statistics over the entire execution.

B. Smart Trace Composition and Exploration

The ST collection is presented in an interactive application as a list of objects that can be drag-and-dropped into a canvas; at this point a default visualization technique is applied, depending on the nature of the object (e.g. execution events are presented as Gantt charts, dataflow modules are shown as a directed graph, tasks are organized in scatter plots.)

Each visual object can be inspected, to have its associated values printed; subsets of objects can be selected (interactively or by scripting), and duplicated. Any subset of objects can be used as the input of a new visualization, chosen from a menu. When objects are selected, their attributes and relationships are shown in the list of objects. Therefore, it is possible to navigate from one entity to another, by dropping new objects into the canvas. Graphical elements can be labeled and colored to show different values associated with the data object.

This workflow allows the user to filter out areas of interest, and try out various parameters without necessarily losing the previous visualization, as long as the user makes duplicates of the objects being manipulated. The original data objects are shared, so at any point all visual objects corresponding to a given data point can be highlighted, linked by arrows, etc.

Coordinated multiple views are implemented by letting visualizations share parameters and trigger updates as a result of certain interactions. A long area chart for example can offer multiple zoom points that can be dragged in the time axis to let other visualizations display the relevant information.

V. Results

We designed experiments to validate the expressive power of STs. In this section we detail the implementation, starting with an enumeration of the goals of the trace analysis, followed by results obtained for each dataflow construction.

A. Dataflow System

Many of the existing dataflow systems are tightly coupled with larger specific environments (e.g. VisTrails, SCIRun) or have too rigid implementations (e.g. VTK, where the coordination of execution is not centralized into a single entity.) For that reason we chose to use HyperFlow [5], a light-weight parallel dataflow system, written in C++, that supports execution on both CPUs and GPUs. In HyperFlow, each module is implemented by deriving a base class. Besides describing input and output ports, the module can also specify both CPU and GPU implementations (the latter using CUDA). The scheduler (which runs in its own thread) uses a priority queue for the execution requests. The priority is based on the order the corresponding tasks were submitted to the system (the first task submitted has the highest priority). Whenever an execution thread is available, the scheduler looks for an execution request in the execution queue, and assigns it to the thread. In addition, the system tries to follow the streaming paradigm where data needs to be pushed downstream as soon as its ready. This constitutes a best-effort priority-based scheduling, as the order of completion of the tasks is not rigidly enforced. The centralized scheduler is the single point requiring instrumentation to collect the traces of the execution.

B. Trace Analysis Goals

We designed experiments to answer the following questions:
Q1: Identify the most time consuming modules in a dataflow and inspect its effect in the pipeline performance;
Q2: Identify which input tasks cause certain modules to take more time to compute;
Q3: Recognize which modules have their execution stalled due to waiting input from other modules;
Q4: Visually inspect performance aspects of a dataflow running in a heterogeneous system with CPUs and GPUs;
Q5: Recognise runtime characteristics that are emergent from various implementation details.

Different dataflow constructions are used to evaluate the questions above. The first two dataflow constructions were synthetically generated to simulate abnormal behaviors that can be spotted upon analysis. Each synthetic dataflow is specified by its topology, the estimated run time costs of each module, and the number of tasks to be processed during execution. The first example consists of a complex pipeline with a single module bottleneck that runs slower than the remaining modules. This construction addresses Q1. The second example uses an irregular workload, with multiple execution paths converging to a single join module at different speeds. In this case, the join module has to wait for all its inputs to be available. Such construction addresses Q2. Finally, we created intermittent slowdowns, where multiple inputs are processed, but only a few make some modules run slower (e.g. an image processing module that is optimized for power-of-two sized images). This addresses Q3. To address Q4 we used an image processing pipeline with modules that can be executed both on CPU and GPU. Finally, Q5 was addressed by analyzing inputs with larger degrees of parallelism.

C. Pipeline Results

1) Complex Pipeline: The first dataflow, shown in Figure 3 (left), is called *Complex* and contains multiple paths with different runtime costs joining at three points. The module *Join1* runs much slower than the others. It is not only a bottleneck, but almost always guarantees that the input for the following module (*Join2*) arrives late. Figure 3 (right) shows the result of an execution with 8192 inputs running with 256 worker threads on a 264-core SMP machine, with a time-compressed stacked line graph. The time scale was compressed non-linearly so each event in the graph has unit length, no matter how short or how long that event lasted.

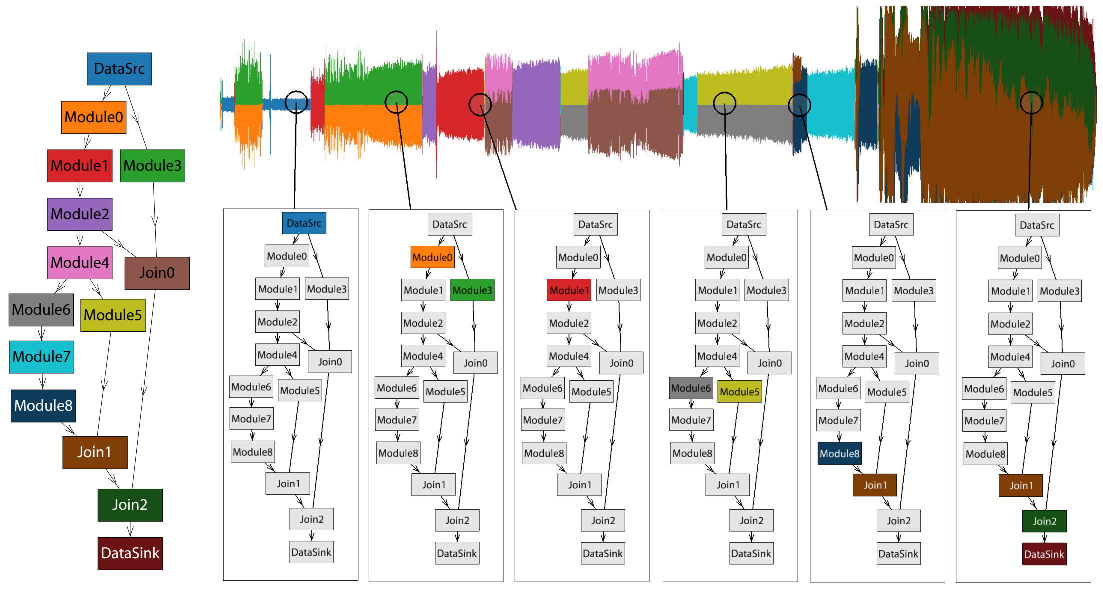

Fig. 3. Visualizing the execution of the *Complex* pipeline (left) with 8192 input tasks running with 256 threads. The time-compressed stacked line graph (right) represents the concurrency of execution (number of active threads, on the *y*-axis) over time (*x*-axis). The lines are ordered and colored by the module the corresponding thread is executing, forming large areas of the same color instead of alternated colored lines. The compressed stacked line-graph can identify interesting patterns, such as the brief interruption of execution of the first module *DataSrc* (blue) to execute modules 0 and 3 (green and orange). Snapshots of the pipeline states are given below, showing the modules executed at certain timesteps (e.g. the fourth snapshot has modules 5 (yellow) and 6 (gray).

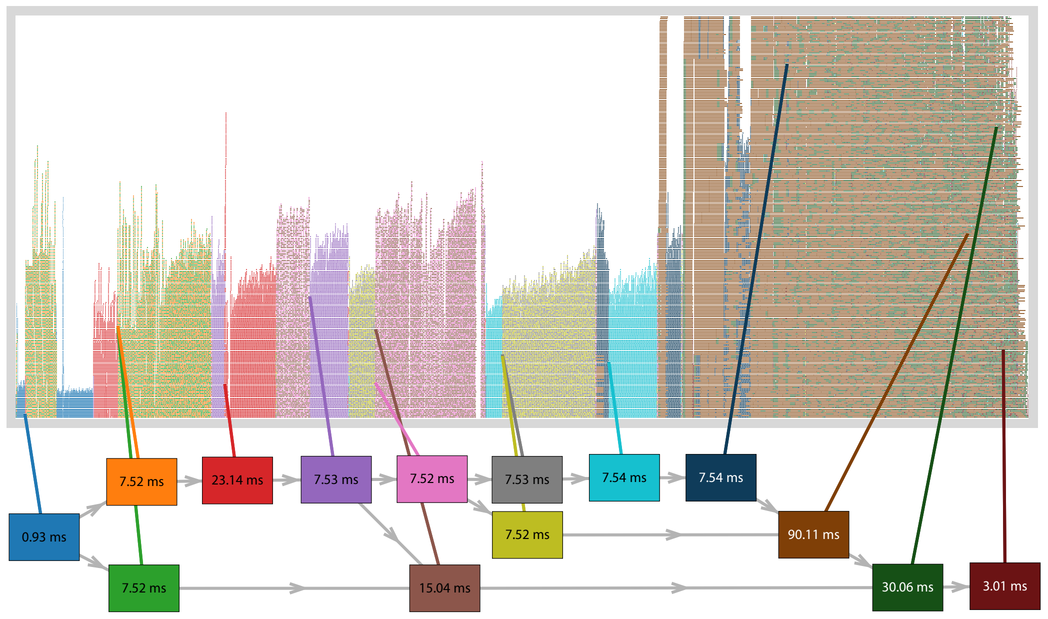

Fig. 4. Execution of one task in a system composed of 256 CPUs. (top) Gantt chart of thread utilization (bottom) time annotated diagram with time spent to execute each module for this specific task. Gray arrows show data movement through the pipeline, and the colors show the corresponding execution inside the Gantt chart. We can see that not only *Join0* (light brown) took longer to execute, it was scheduled much later after the termination of *Module8* (dark blue).

This shows more clearly short-lived behaviors without risking other events overlapping the same area of the chart. We can observe that the scheduler favors a grouping of executions by module, with a small level of concurrency for most of the execution. When data is available from the *DataSrc* module, execution of *Module0* and *Module3* starts, with little overlap of execution of these last two modules and *DataSrc*. Snapshots of active states at six timesteps during the execution is displayed below, coloring only active modules. This suggests that the synchronization between the scheduler and the worker threads is comparable to the runtime of each execution, thus the scheduler is not able to keep the worker threads busy.

The same execution is visualized in Figure 4, with a full Gantt chart on top, and a single dataflow task (corresponding to the the 1811th input) displayed below. The network below is composed of the actual state change events for that task, copied from the Gantt chart visualization, and inter-connected through the dataflow topology. The duration property of the event objects was used as the label of each event, and connections were created between them and the Gantt chart to show where in time they occur. We verify that module *Join1* is taking much longer to execute, but for this particular input it was also scheduled to executed much later in time than one would expect (e.g. after *Module8*, colored in dark blue).

2) Irregular Pipeline: This second dataflow models a fork (Figure 5), shown as *time annotated diagrams*, where the user selected to display statistics instead of the default labels. In this case one path has an arbitrarily irregular runtime, which slows down once every 17 inputs. This characteristic can be hard to identify by just looking at average run times. This dataflow was executed with 1024 inputs, with 4 working threads.

The communication between modules corresponds to the time since the data was available from the output port of a module until it was consumed by the next module. Delays in data transfer are shown in Figure 6(a), with the start of the delay on the x-axis in seconds, and the duration of the delay in the y-axis, in nanoseconds, with a logarithmic scale;

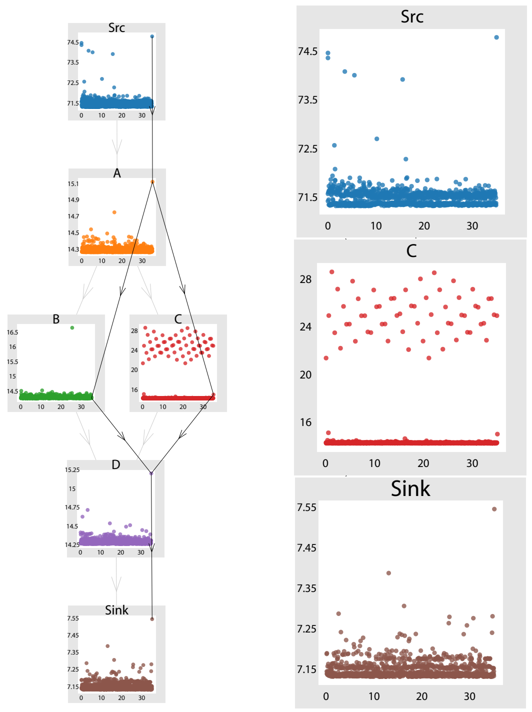

Fig. 5. Irregular pipeline: (left) dataflow with a per-module scatter plot of all executions, comparing the start time (x-axis, in seconds) with duration (y-axis, in milliseconds). Gray arrows show pipeline connections, while black arrows connect executions to one specific input. (right) zoom of *Src*, *C* and *Sink*.

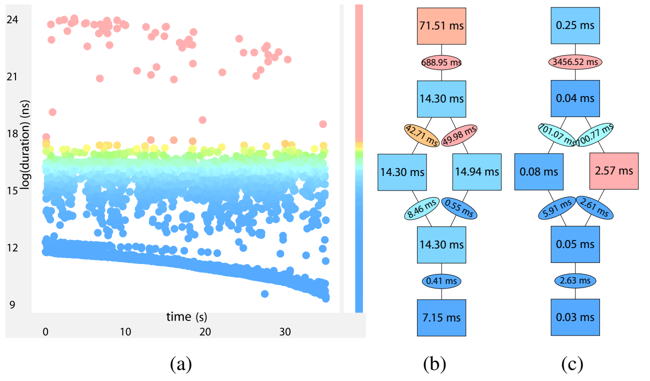

Fig. 6. Run times and delays in the *Irregular* pipeline: (a) scatter plot of data transfer delays comparing start time (x-axis, in seconds) and duration (y-axis, in nanoseconds, natural log scale), with the color map used on its right side, (b) mean values and (c) standard deviation for run times and transfer delays, which are annotated inside modules and on edges respectively.

the color map corresponds to the y axis[1]. We can clearly see three *groups* of delays in the plot. The shortest delays (on the bottom of the chart) are towards the end, most likely due to the scheduler queue getting smaller as the tasks complete; another group of delays vary more randomly, but on the same range, during the whole execution; and a few, but much longer delays (i.e. on the top), that also seem to decrease in duration towards the end. In Figure 5 (b) we can see the mean run times and transfer delays for modules and connections, while Figure 5 (c) shows the standard deviations of the same data. All values use the same color map as the scatterplot, normalized between the minimum and maximum values for each quantity, independently for modules and connections. Just after the data is produced by the first module (on top), and where the paths split, we see the data is idle for much longer than the rest of the connections. The scheduler seems to prioritize the left path, as data waits longer to go to the right path, and is the last to be consumed at the junction point (the larger delay on the left means the module was idle waiting for the right path). The standard deviation diagram shows the irregular runtime on the right path, as well as the variability in delays for pushing the data from the top two modules.

3) Edge Detection Pipeline: processing application that performs edge detection using the “Difference of Gaussians” algorithm, on a large image composed of individual tiles (each tile being an independent input for the pipeline), then composed into a single image output. The individual operations (inversion, blurring and difference) are performed in independent modules. The *Invert* and *Gaussian Blur* modules have two implementations, one for the CPU and another for the GPU (using CUDA). While many embarrassingly parallel tasks (such as many image processing operations) tend to perform better on the massively parallel hardware available on GPUs, it does not guarantee a performance boost; the overhead caused by data transfer and synchronization can easily offset the runtime trimmed from the execution if the implementation

[1] $e^9 \approx= 8.1\mu s,\ e^{12} \approx= 163\mu s,\ e^{15} \approx= 3.3ms,\ e^{18} \approx= 66ms, \ldots$

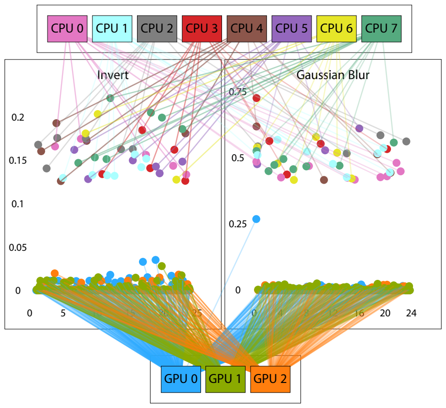

Fig. 7. Scatter plot for two modules (*Invert* and *Gaussian Blur*) showing duration of executions (y-axis) over time (x-axis) Lines connect the execution to the thread where it occurred, either on CPUs or GPUs. Observe that a single GPU execution in the *Gaussian Blur* module takes much longer to run.

is not properly tuned. This application received 512 input images with 3 MPixels each, running with 8 CPU threads and 3 GPU threads. Figure 7 shows an analysis focused on the events in modules *Invert* and *Gaussian Blur*. A scatterplot for the duration of each execution shows mainly two groups of executions. The *thread* objects were added to the visualization and connected by lines to events they executed. Colors are mapped to threads related to the object, showing how the fastest executions were performed by GPU implementations of those threads. There is one outlier in the *Gaussian Blur* module that took considerably more time to complete.

The first 4 seconds of execution are shown in Figure 8. The top Gantt chart shows one line per thread (first 3 lines are GPUs, remaining lines are CPUs). The first modules to execute on the GPU threads (data uploads) take a long time to execute, due to the CUDA runtime initialization. Soon after that, the GPU threads stay idle for most of the time, which is expected as the *Decode Image* (blue) is too slow in comparison. Reorganizing the chart by module (middle) we see an undesired behavior on the *Decode Image* module: it stops reading input images from the file system. By focusing on this module alone, we create another visualization (bottom) with a stacked line graph, showing one thread per line, and notice a second undesired behavior: with too many threads trying to perform I/O at the same time, the performance is likely to decrease, as I/O buffers (and possibly the physical media) have to deal with many non-sequential accesses. An ideal execution for this module would have at a time only a few threads active and no empty gaps (to minimize concurrent I/O overhead), and a scheduling policy could provide this behavior.

VI. Limitations

The design decision of using Python to represent and manipulate data implies an execution overhead inherent of the code being interpreted, with little to none optimization. This performance penalty is purely technological, and can be amortized if the chosen implementation uses a JIT interpreter. On the other hand, the dynamic aspect provides a convenient workflow without requiring the user to process data outside the environment to extract new information from the initial dataset. The user interface is still very experimental, and very tied to programming concepts. For most of the exploration we feel it should provide enough functionality without requiring programming skills; on the other hand, it is reasonable to expect a user analysing this data to be familiar with basic programming concepts. We also want to be able to generate animations of trace data from within the platform. Some visualizations require processing large amount of time-varying data, which are challenging to produce real-time animations.

VII. Conclusion and Future Work

In this work we presented different ways to look at parallel trace data, in particular to parallel dataflow traces. Current tools for trace analysis are limited to displaying resource utilization Gantt charts. As we demonstrated along the work, trace data is rich and full of information that can be seen in different perspectives. The notion of ST captures the idea of creating linked visualizations, which can be procedurally generated or simply generated from pre-recorded templates. The generality of the approach, enhanced with dataflow information, allowed us to observe this data in different ways, illustrate throughout the text with several examples.

In the future we would like to make the tool available to the public, as well as be able to perform code instrumentation for existing dataflow platforms, such as VTK. We believe it can be useful for the VTK user community, where we could test our system with diverse usage scenarios. This would require validating our technique with larger dataflow topologies.

Acknowledgment

The authors would like to thank CNPq (processes 476685/2012-5 and 309483/2011-5).

References

[1] M. Heath and J. Etheridge, "Visualizing the Performance of Parallel Programs," *Software, IEEE*, vol. 8, no. 5, pp. 29 –39, 1991.
[2] E. Kraemer and J. T. Stasko, "The Visualization of Parallel Systems: an Overview," *J. Parallel Distrib. Comput.*, vol. 18, pp. 105–117, 1993.
[3] S. Hackstadt, A. Malony, and B. Mohr, "Scalable Performance Visualization for Data-parallel Programs," in *Scalable High-Performance Computing Conference, Proceedings of the*, 1994, pp. 342 –349.
[4] G. Tomas and C. W. Ueberhuber, *Visualization of Scientific Parallel Programs*. Secaucus, NJ, USA: Springer-Verlag New York, Inc., 1994.
[5] H. Vo, D. Osmari, J. Comba, P. Lindstrom, and C. Silva, "HyperFlow: A Heterogeneous Dataflow Architecture," in *Eurographics Symposium on Parallel Graphics and Visualization*, 2012, pp. 1–10.
[6] S. Havre, B. Hetzler, and L. Nowell, "ThemeRiver: visualizing Theme Changes over Time," in *Information Visualization, IEEE Symposium on*, 2000, pp. 115–123.
[7] L. Byron and M. Wattenberg, "Stacked Graphs – Geometry & Aesthetics," *IEEE Transactions onVisualization and Computer Graphics*, vol. 14, no. 6, pp. 1245–1252, 2008.
[8] D. Holten, "Hierarchical Edge Bundles: Visualization of Adjacency Relations in Hierarchical Data," *IEEE Transactions on Visualization and Computer Graphics*, vol. 12, pp. 741–748, 2006.
[9] J. Waser, R. Fuchs, H. Ribičič, B. Schindler, G. Blöschl, and E. Gröller, "World Lines," *IEEE Transactions on Visualization and Computer Graphics*, vol. 16, no. 6, pp. 1458–1467, 2010.
[10] H. L. Gantt, *Work, Wages, and Profits*. New York : The Engineering magazine co., 1913.

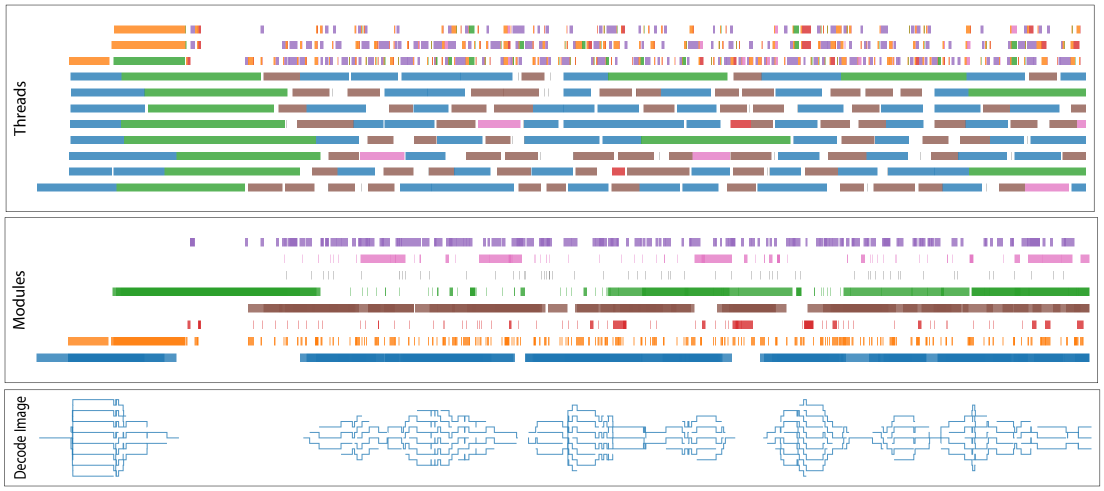

Fig. 8. Top and middle: two Gantt charts visualizations of two STs, one parametrized by thread and one by pipeline module; the time interval displayed is for the first 4 seconds of execution. The orange represents data upload to the GPU, and its first execution in each GPU thread (top three lines of the top chart) take longer to complete because of the CUDA runtime initialization. The bottom chart is a stacked line chart by module, where each line represents a thread executing that module, showing the concurrency profile of each module; the time is compressed non-linearly, where each event takes one unit of time.

[11] R. Schiefer and P. van der Stok, "VIPER: a Tool for the Visualisation of Parallel Programs," *Parallel and Distributed Processing, 1995. Proceedings. Euromicro Workshop on*, pp. 540–546, 1995.

[12] T. Wagner and R. Bergeron, "A Model and a System for Data-parallel Program Visualization," *Visualization, 1995. IEEE Conference on*, pp. 224–231, 1995.

[13] E. Karrels and E. Lusk, "Performance Analysis of MPI Programs," in *Proceedings of the Workshop on Environments and Tools for Parallel Scientific Computing*, 1994, pp. 195–200.

[14] C. Wu, A. Bolmarcich, M. Snir, D. Wootton, F. Parpia, A. Chan, E. Lusk, and W. Gropp, "From Trace Generation to Visualization: A Performance Framework for Distributed Parallel Systems," *Supercomputing, ACM/IEEE 2000 Conference*, p. 50, 2000.

[15] C. Wu and A. Bolmarcich, "Gantt Chart Visualization for MPI and Apache Multi-Dimensional Trace Files," in *ICPADS '02: Proceedings of the 9th International Conference on Parallel and Distributed Systems*. IEEE Computer Society, 2002, pp. 523–528.

[16] C. Sigovan, C. Muelder, and K.-L. Ma, "Visualizing Large-scale Parallel Communication Traces Using a Particle Animation Technique," *Comput. Graph. Forum*, vol. 32, no. 3, pp. 141–150, 2013.

[17] J. Roberts and C. Zilles, "TraceVis: an Execution Trace Visualization Tool," in *Proceedings of Workshop on Modeling, Benchmarking and Simulation (MoBS)*, 2005, pp. 5–12.

[18] A. Knüpfer, H. Brunst, and W. Nagel, "High Performance Event Trace Visualization," *Parallel, Distributed and Network-Based Processing, 2005. PDP 2005. 13th Euromicro Conference on*, pp. 258–263, 2005.

[19] D. Martinez, V. Blanco, M. Boullon, J. Cabaleiro, C. Rodriguez, and F. Rivera, "Software Tools for Performance Modeling of Parallel Programs," in *Parallel and Distributed Processing Symposium, IEEE International*, 2007, pp. 1 –8.

[20] S. Mysore, B. Mazloom, B. Agrawal, and T. Sherwood, "Understanding and Visualizing Full Systems with Data Flow Tomography," *SIGPLAN Not.*, vol. 43, pp. 211–221, 2008.

[21] M. Schnorr, O. Navaux, and B. de Oliveira Stein, "Dimvisual: Data integration Model for Visualization of Parallel Programs Behavior," *Cluster Computing and the Grid, 2006. CCGRID 06. Sixth IEEE International Symposium on*, vol. 1, pp. 473–480, 2006.

[22] L. Schnorr, G. Huard, and P. Navaux, "Towards Visualization Scalability through Time Intervals and Hierarchical Organization of Monitoring Data," *Cluster Computing and the Grid, 2009. CCGRID '09. 9th IEEE/ACM International Symposium on*, pp. 428–435, 2009.

[23] ——, "3D Approach to the Visualization of Parallel Applications and Grid Monitoring Information," *Grid Computing, 2008 9th IEEE/ACM International Conference on*, pp. 233–241, 2008.

[24] D. Hackenberg, G. Juckeland, and H. Brunst, "High Resolution Program Flow Visualization of Hardware Accelerated Hybrid Multi-core Applications," *IEEE/ACM International Conference on Cluster, Cloud and Grid Computing*, pp. 786–791, 2010.

[25] J. Trümper, J. Bohnet, and J. Döllner, "Understanding Complex Multi-threaded Software Systems by using Trace Visualization," in *Proceedings of the 5th international symposium on Software visualization*, ser. SOFTVIS '10. New York, NY, USA: ACM, 2010, pp. 133–142.

[26] D. Holten, B. Cornelissen, and J. van Wijk, "Trace Visualization using Hierarchical Edge Bundles and Massive Sequence Views," *Visualizing Software for Understanding and Analysis, 2007. VISSOFT 2007. 4th IEEE International Workshop on*, pp. 47–54, 2007.

[27] W. Javed, B. McDonnel, and N. Elmqvist, "Graphical Perception of Multiple Time Series," *IEEE Transactions on Visualization and Computer Graphics*, vol. 16, no. 6, pp. 927–934, 2010.

[28] N. Andrienko and G. Andrienko, *Exploratory Analysis of Spatial and Temporal Data: A Systematic Approach*. Springer-Verlag New York, Inc., 2005.

[29] J. C. Roberts, "State of the art: Coordinated & multiple views in exploratory visualization," in *Proceedings of the Fifth International Conference on Coordinated and Multiple Views in Exploratory Visualization*, 2007, pp. 61–71.

[30] G. Andrienko and N. Andrienko, "Coordinated Multiple Views: a Critical View," in *Coordinated and Multiple Views in Exploratory Visualization, 2007. CMV '07. Fifth International Conference on*, 2007, pp. 72 –74.

[31] A. Lex, M. Streit, C. Partl, K. Kashofer, and D. Schmalstieg, "Comparative Analysis of Multidimensional, Quantitative Data," *IEEE Transactions on Visualization and Computer Graphics*, vol. 16, no. 6, pp. 1027 –1035, 2010.

[32] A. Forbes, T. Hollerer, and G. Legrady, ""Behaviorism": a Framework for Dynamic Data Visualization," *IEEE Transactions on Visualization and Computer Graphics*, vol. 16, no. 6, pp. 1164–1171, 2010.

[33] B. M. Cantrill, M. W. Shapiro, and A. H. Leventhal, "Dynamic Instrumentation of Production Systems," in *Proceedings of the annual conference on USENIX Annual Technical Conference*, 2004, pp. 2–2.

[34] M. Müller, A. Knüpfer, M. Jurenz, M. Lieber, H. Brunst, H. Mix, and W. Nagel, "Developing Scalable Applications with Vampir, VampirServer and VampirTrace," *Parallel Computing: Architectures, Algorithms and Applications*, vol. 15, pp. 637–644, 2007.

[35] S. S. Shende and A. D. Malony, "The Tau Parallel Performance System," *Int. J. High Perform. Comput. Appl.*, vol. 20, pp. 287–311, 2006.