bugger (`gdb`) can use UST to record GDB tracepoints. The kernel tracer captures activity for all processes and the system.

Traces are recorded in the *Common Trace Format* (CTF)[46]. The user-space tracer records events with zero-copy[47], resulting in a low overhead of 700 ns per event (according to the documentation). The overhead for the kernel-tracer is even lower.

LLTV is the *LTTng Viewer*, a tool providing statistical, graphical and text-based *views* of the recorded traces. Internally, the individual views are realized as modules. It has been shown capable of handling traces with a size of 10 GiB. Another viewer is incorporated into the *eclipse*[48] IDE as a plugin. With *LTTng*, several traces can be visualized concurrently; this technique has been used to view traces of a virtual machine together with a trace of the XEN host system [DD08].

Exemplary screenshots of LLTV for the desktop system running `partdiff-seq` are discussed next. In Figure 2.12, the *event view* and *control flow view* are presented. The *event view* – shown in the upper half of the screenshots – lists individual events in a textual representation while the *control flow view* – visible in the lower half of the screen – shows the activity in a timeline for each individual process. Graphical representation of the trace encodes the activity of each process in colors; green, for instance, encodes that the process on the left executes in user space (this is *partdiff-seq*).

The *statistical view* and *resource view* are shown in Figure 2.13. In the statistical view, aggregated information of all kinds of activity is provided. In the resource view, the activity on each CPU and interrupt can be observed, white color encodes user-space activity dispatched on a CPU. For easy analysis, the screenshot has been modified to show the likely[49] execution of *partdiff-seq* on the CPUs with green color. The vertical line marks the start of the program, then the single threaded program is migrated between all 4 cores on the quad-core system.

Consequently, with the tool, interactions between processes and system activity at a given time can be analyzed extensively. Note that while the overhead of the tracer is very low compared to other tools tracing 50 seconds of system activity created a set of files with the aggregated size of 57 MiB.

The tool provides capabilities to filter events, which is unfortunately not available for the graphical trace representation.

2.4.4. Available Tools for Analysis of Parallel Programs

Compared to the tools for sequential programs, tools for parallel applications are more involved. In the following, a few tools are presented which are aware of the programming and execution models of parallel applications, hence, they explicitly support the analysis of parallel applications. *VampirTrace* records execution behavior of HPC applications in the *Open Trace Format*[50]. Popular post-mortem performance analysis tools that analyze these trace files are *TAU*, *Vampir* and *Scalasca*. Those tools provide several ways to assist a developer in assessing application behavior.

An experimental tool environment which should be named is PIOviz [LKK+06a]. This environment is capable of tracing not only parallel applications, but also triggered activity on the *Parallel Virtual File System* version 2 (PVFS). In the following, brief introductions and screenshots are provided for all the mentioned tools.

[46] `http://www.efficios.com/ctf`

[47] Zero-copy is a technique in which data is communicated without copying it between memory buffers – in this case no copy between kernel-space and user-space is necessary.

[48] `http://www.eclipse.org/`

[49] It is not possible with the tool to actually relate applications with the resource view, therefore, the execution is guessed by the author based on the information provided by the control flow view.

[50] Further information is provided on Section 2.4.5.

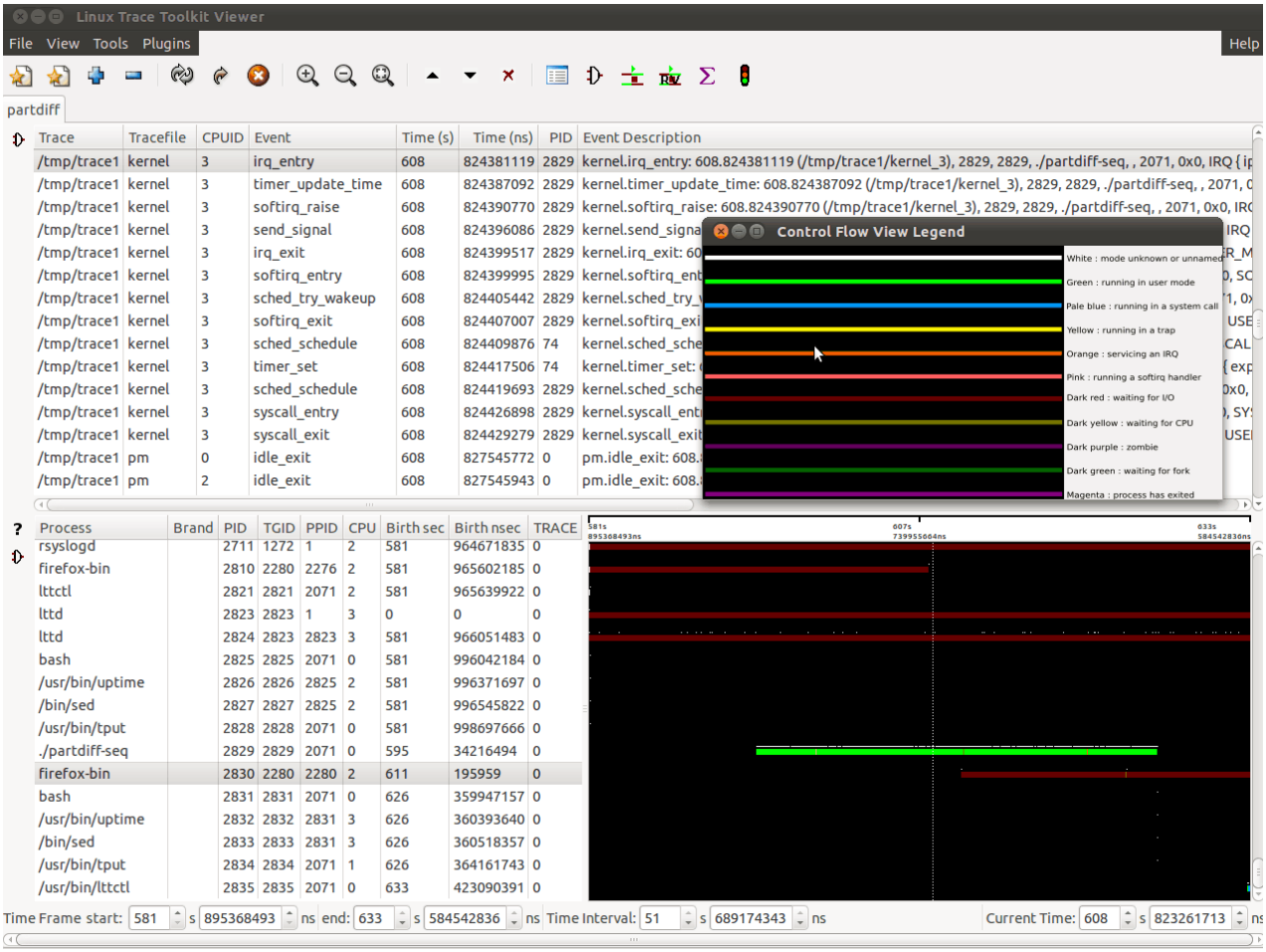


Figure 2.12.: Screenshot of the LTTV viewer for a trace of the system – event view and control flow view.

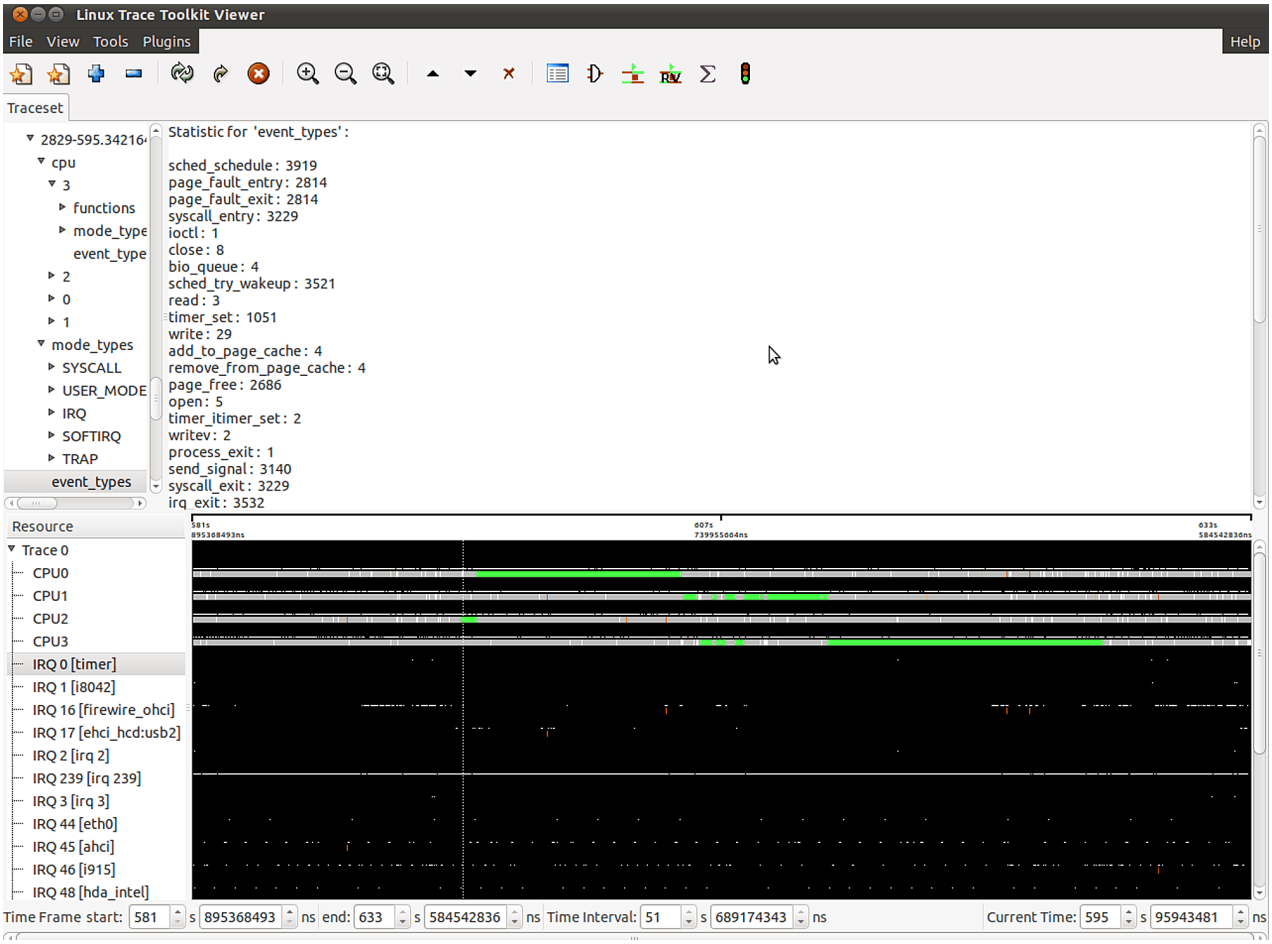


Figure 2.13.: Screenshot of the LTTV viewer for a trace of the system – statistic view and resource view. The likely scheduled execution of the program on the CPUs is marked in green.

VampirTrace

VampirTrace [MKJ+07, KBD+08] is a library and tool set which generates traces or profiles for MPI and OpenMP applications. It instruments applications to write files in the *Open Trace Format* (OTF) at runtime. Depending on the type of instrumentation, function calls of the application, MPI and OpenMP activity are recorded. Additionally, low-level POSIX I/O calls and POSIX Threads are intercepted, CPU counters are supported. Additional counters can be supplied by plugins [STHI10].

VampirTrace utilizes *MPI Profiling Interface* (PMPI), a profiling interface provided by MPI[51]. VampirTrace stores a subset of parameters and context information from MPI calls. This enables identifying communication partners and provides information about the exchanged data, e.g., the message size. OpenMP provides an interface that eases instrumentation, the *OpenMP Pragma and Region Instrumentor* (OPARI) [MMSW02], which is used by VampirTrace.

Furthermore, it supports performance counters provided by PAPI. In case the tracing of performance counters is enabled by the user, selected counters are recorded for every generated event.

To perform source code instrumentation, the application must be recompiled using provided build scripts. These are wrappers for the compiler and include a source-to-source compiler which modifies functions in order to generate events at every function entry and exit. *Dyninst*[52] [BH00] can be used to instrument binaries, too. However, the binary instrumentation has the drawback that only limited information about application internals are accessible.

TAU

The *Tuning and Analysis Utilities* [SM06], are the swiss army knife for performance analysis. They support not only many languages (C, C++, Java, Python, CUDA, ...), but also a rich variety of capabilities like profiling and tracing, sampling, throttling of event generation, extended information for POSIX I/O and communication and hardware counter support via PAPI.

Several tools are shipped with TAU which include: converters to export data to other trace or profile formats, automatic instrumentation for source code, and a visualizer for trace profiles (*ParaProf*). *Jumpshot* [ZLGS99] or *Vampir* can be used to visualize traces. The Jumpshot viewer[53], shipped with TAU, is originally part of the *MPI Parallel Environment* (MPE) and visualizes the *SLOG2* trace format.

ParaProf is a Java GUI which displays profiling results. Compared to the tools for sequential analysis, like *gprof*, with ParaProf all processes and threads of a parallel application can be analyzed together. The tool focuses on analyzing behavior of the processes and threads for a single metric at a time. Values for the selected metric can be compared, additionally average and standard deviation is provided. The metric under investigation can be chosen – for example, time or floating point operations, and the data is provided either for the executed functions or a group of functions. Additionally, derived metrics like Flop/s can be created. To analyze time-dependent behavior, TAU provides an API that can be used in a program to divide execution into phases, which are profiled and analyzed independently.

The available views in ParaProf are histograms, a communication matrix showing the interaction between the processes, textual or a 3D visualization. The 3D visualization shows the value of a selected metric for each pair of thread and function.

Performance results of experiments can be archived, either directly in ParaProf or in a (remote) database. Metrics can be compared across experiments to monitor performance over the application development, or

[51] The MPI Profiling Interface defines that every MPI function is exported with two names, an MPI prefixed call and a PMPI prefixed version. Users use the MPI call; by providing an instrumented version of the MPI function, all user calls can be intercepted. Instrumented functions can perform appropriate logging and they usually call PMPI functions that are directly provided by the MPI library.

[52] `http://www.dyninst.org/`

[53] See Page 77 for further information on Jumpshot.

run-time behavior of various parameter sets can be assessed. *PerfExplorer* is another tool shipped with TAU, it eases statistical analysis of profiles such as clustering and correlation across experiments. It can be used to create diagrams, for example to illustrate achieved speedup. Furthermore, application performance can be assessed over multiple runs, and even a longer time period.

In order to demonstrate a few of the available views, a profile of our PDE was generated for 4 processes and visualized in ParaProf. In Figure 2.14, a few windows are shown. In the upper window, the experiment is chosen and information about run-time environment characterization is given which identifies the experimental setup and run-time settings. The legend windows (on the left) show the color-code and name of the recorded events and available groups. To improve user interaction, the events and function calls are grouped into sets – in the example, the MPI group can be selected to assess communication overhead quickly. In TAU, the call-graph is just encoded into the event names. Thus, the number of events in the function legend is rather large; in the example, calculate invokes several functions.

The main data window (in the center) with the unstacked bars in the middle of the figure displays a metric for all functions and processes. In the screenshot the *time* metric has been chosen and the mouse hovers over the calculate function. By clicking an interesting function, a window pops up that just shows this metric for all threads. In the figure, the window below shows the time for `MPI_Sendrecv()` which is called by the exchange function.

Interaction between the 4 processes can be analyzed with a communication matrix, in Figure 2.15 the number of calls is encoded in a heat-map. In the figure, a diagonal communication pattern of the PDE can be observed – a process communicates with its two neighbors. A few communications from the ranks to Rank 0 are seen, this is the finalization step in which the results are gathered.

Text statistics of Rank 0 for I/O and messages are provided in Figure 2.16, showing the number of I/O calls and amount of data.

Note that the standard automatic instrumentation, which intercepts every function entry and exit, causes serious overhead; by tracing, the wall-clock run-time of the PDE increases from 13 s to 576 s. In the same run, more than 1.8 GiB of trace data will be generated per process, if tracing is enabled. Consequently, this demonstrates the need to filter unimportant events during the instrumentation or at run-time. TAU provides tools to automatically filter events, and it throttles an event when it is fired too often.

For more information about visualization and usage refer to the *TAU User Guide* [Tau10].

Vampir

Vampir [MKJ+07, KBD+08] is a mature commercial trace analysis tool, which visualizes files that have been generated with VampirTrace. However, it has a proprietary code base and cannot be extended by third parties.

Large trace files may not fit into the memory of one machine and take a long time to visualize and preprocess, therefore, the software called *VampirServer* extends the performance of Vampir by outsourcing analysis capabilities and trace handling to a set of remote processes. The number of server processes can be scaled to match the size of the trace file and the required performance.

In Vampir, multiple displays, each performing a specific visualization, can be arranged to a *workspace*. Displays can be configured, e.g., to show a particular process, to filter information or to visualize another metric. Zooming on a timeline is propagated to all displays.

For a PDE run, a workspace containing all displays has been created and is shown in Figure 2.17. Here, only a fraction of time is actually traced as the overhead with the default instrumentation that traces invocations of all functions is overwhelming[54].

[54] The PDE for the trace runs 100 iterations instead of the 10.000 that have been executed with an instrumented program for the other tools.

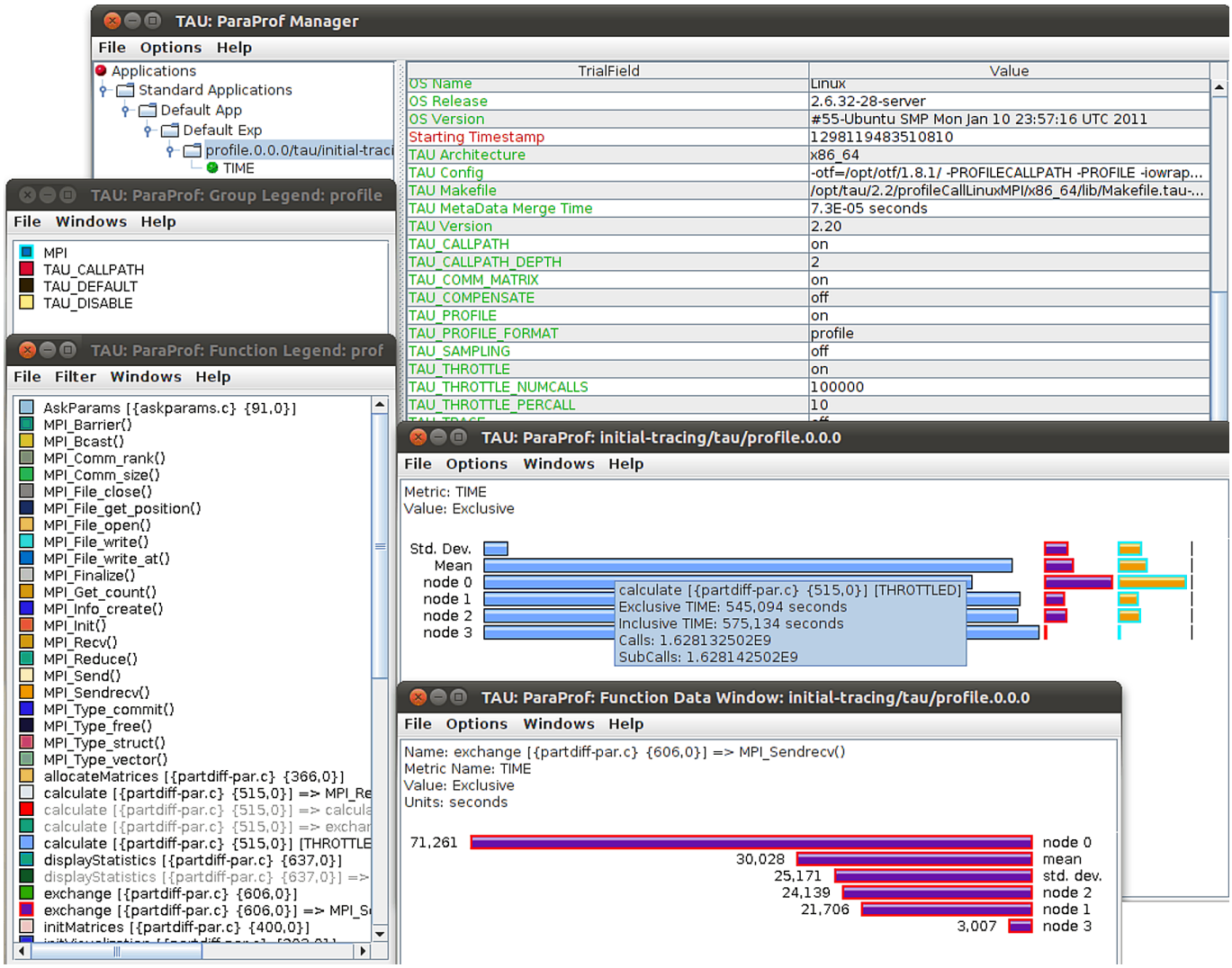


Figure 2.14.: Screenshot of ParaProf – PDE profile including experimental information.

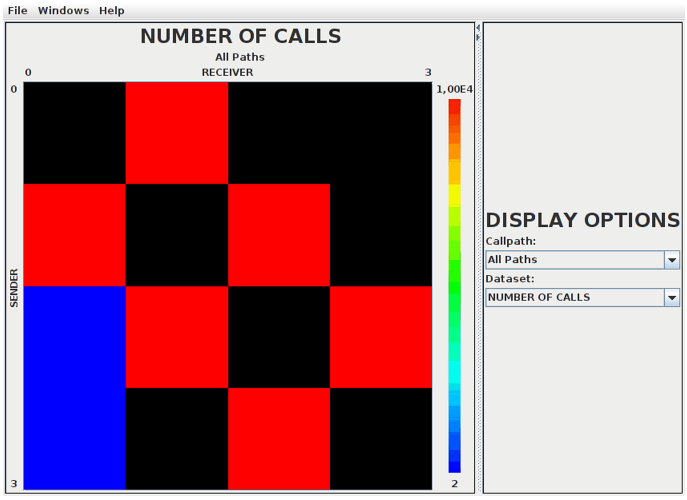


Figure 2.15.: Screenshot of ParaProf – PDE communication matrix.

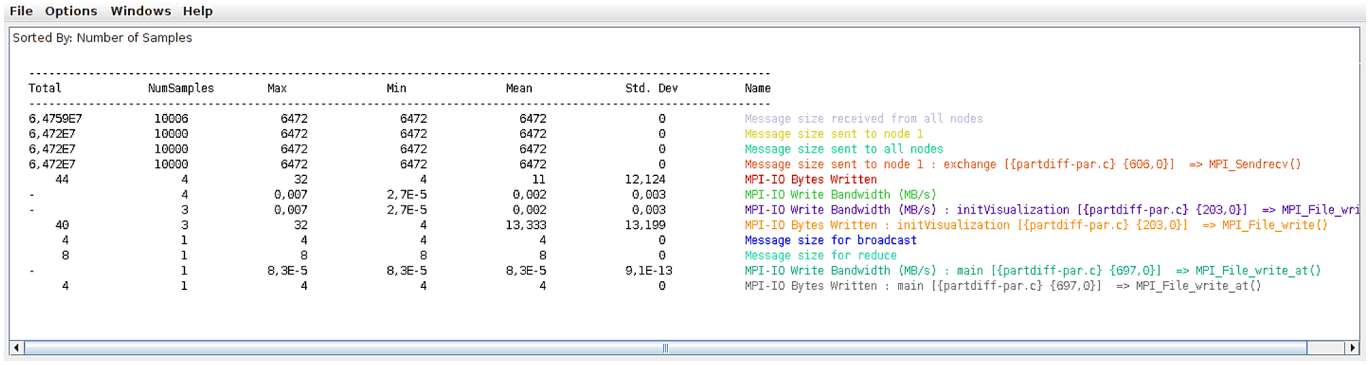


Figure 2.16.: Screenshot of ParaProf – *user event statistics* for Process 0 including MPI-IO statistics.

As Vampir has a huge number of displays, they are enumerated in the screenshot and described individually:

1. Next to the symbol toolbar is an overview timeline, which is visible all the time and does not change with zooms. This timeline shows the observed activity of all processes over time in a condensed form. The height of color in the bar encodes the number of processes that execute an operations of a given kind at a given time. Functions are grouped during the tracing and represented by different colors, example groups are I/O, communication or application. By default, MPI functions are colored in red whereas time spent in the application is colored in green. If all processes perform operations of the same group, then a column contains one color, for example at the beginning all processes perform MPI calls. To add a certain display, a user can click on the specific icon in the toolbar.

2. The *Master Timeline* shows the activity for each individual thread in colors according to the group. Inter-process messages are visualized by black lines. In Vampir there is no concept which associates the processes to the hardware topology. Therefore, the user must know the mapping of the processes to hardware.

3. In the *Process Timeline*, the call-stack of an individual process is given (here Process 0).

4. A *Counter Timeline* shows the values of one PAPI counter for one process. The minimum, maximum and average values can be plotted into one graph. In this display, the total number of instructions which were performed is plotted. The observed spike during startup might be an artifact caused by an overflow of the counter.

5. This is also a counter timeline that visualizes the Flop/s for Process 0. Note that there could be as many replicates of the displays as fit on the screen.

6. For one selectable metric, the *Performance Radar* encodes the counter values in a given color, similar to a heat map. In contrast to the counter timeline, all processes can be visualized together. Unfortunately, due to the counter overflow in the example, all processes are drawn in blue.

7. The *Call Tree* shows the call-graph together with a profile for the visualized time interval.

8. In the *Context View*, more information of the selected object is provided – most visible entities, such as an event, function or process, can be selected. In the example, the whole function group is clicked.

9. The *Function Summary* provides information on a set of processes. In the left window, the exclusive time of all processes is accumulated while the right window prints the number of invocations per group.

10. Available groups and their color scheme are listed in the *Function Legend*. By instrumenting the application manually with VampirTrace, more groups can be created.

11. An overview of the number of messages or message sizes is given in the *Message Summary*.

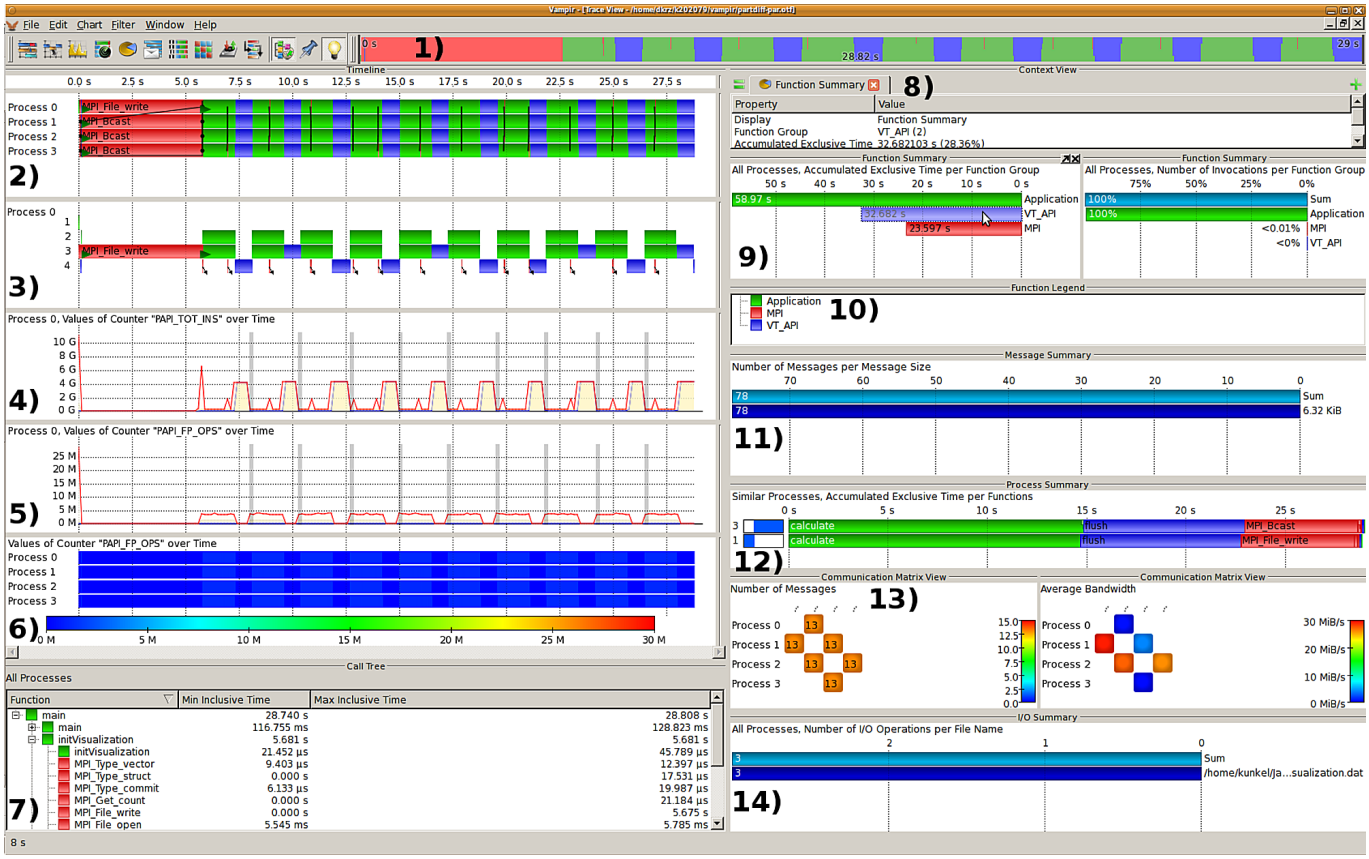


Figure 2.17.: Screenshot of a Vampir workspace.

12. In the *Process Summary*, a profile is generated and visualized for every process. If the space does not suffice, the display automatically clusters similar processes. In the example, it detected three processes with the upper function profile and one with the lower function profile – the master process which performs the actual I/O.

13. Similar to TAU, the inter-process behavior is visualized in the *Communication Matrix View*. Two displays have been created, the left window just shows the number of messages exchanged between two processes, while the right window indicates the average bandwidth.

14. The *I/O Summary* finishes our tour through the available displays: Several metrics are available and related to the file name: accessed data volume, number of operations or bandwidth.

Scalasca

Scalasca [GWW+10], is a performance analysis toolset which automatically analyzes large-scale parallel applications that use the MPI, OpenMP or a hybrid MPI/OpenMP programming model. It has been successfully applied to applications running with 200.000 processes on a BlueGene/P system [WBM+10].

Scalasca can be run in two modes, either a summary of the parallel program is created at run-time, or the application activity can be recorded in the *EPILOG* trace format and then analyzed post-mortem.

In the trace mode, Scalasca searches automatically for a common class of run-time communication bottlenecks, for example, for late senders. Scalasca ships with the sequential analyzer `expert` [WM03] and the parallel analyzer `scout` that identify wait-state patterns. The parallel analyzer runs with the same number of processes as the original application. While the analysis is performed, it replays the communication pattern of the original program and updates statistics accordingly. Therefore, `scout` scales similarly to the original application. However, the sequential tool `expert` detects more inefficient communication patterns.

Scalasca can instrument the application automatically, either by using compiler options, by transformation of the source code or by linking the program with an already instrumented library. Similarly to the other tools, an API for manual instrumentation is provided.

In contrast to previously referenced tracing tools, with Scalasca the application is typically started with an additional software monitoring system. After the application terminates the system can automatically run `scout` to perform the parallel trace analysis.

Process statistics and identified bottlenecks are displayed in the *Qt* application *Cube3*, which allows browsing through the analysis results.

To assess the features, the PDE configuration with 4 processes is instrumented with Scalasca and instructed to generate summaries.

A screenshot of the summary is provided in Figure 2.18. The view is split into three columns: the first column shows the available metrics, the second column displays all functions in the call-graph and the contribution to the metrics, the last column shows the contribution of every process of the supercomputer to the value of the function (and metrics). In the analysis session, the user selects a metric on the left, then localizes the relevant function and, at last, analyzes the distribution of the selected metrics among the processes. In the given example, the number of send operations is selected in the metrics tree; sends are invoked in the call-graph by `exchange()` which is called by `calculate()` which in turn is called by `main()`.

The view on the left aggregates the metrics among all functions and processes. Also, note that in the hierarchical view, a collapsed node of a column aggregates the values of all children. That means each node shows the inclusive metric, while an expanded node displays the *exclusive metric*, i.e., the contribution to the metrics which is not caused by any of the child nodes. Child nodes show their share by themselves (compare the *Execution* node and children in the left column). A color scheme between blue and red encodes the values similar to a heat map in all three columns. This assists in spotting bottlenecks in the metrics, code and unbalanced processes on the hardware.

For figure Figure 2.18, 60.000 `MPI_Sendrecv()` functions have been invoked from the `exchange()` function. As the inner processes transfer twice as much messages, the four processes call the function 10.000 times, 20.000 times, 20.000 and 10.000 times, respectively. The right view visualizes either the topology of the machine – here a flat topology – or the numerical values as shown in Figure 2.19.

By instrumenting all user functions, the wall-clock time increased from 12 s to 307 s, by instrumenting just MPI with the provided PMPI library, the overhead is not measurable. With Cube, the bottleneck can be identified by looking at the (run-)time metric of the functions and processes in Figure 2.19. A total of 1158 s for all 4 processes is divided into 68 s MPI activity and the remaining time is spent for user activity. In the call-graph, 558 s is spent in `getResiduum()` and another 600 s in the calculate function itself; the right column shows that the load is almost balanced across all processes. As `getResiduum()` has a tiny function body, the overhead of the measurement system dominates run-time. In a real scenario, this function should not be instrumented and therefore would be filtered; Scalasca provides tools to filter events.

For the existing metrics, a short description is provided in the online help that assists the user in understanding them. For example, a screenshot of the metrics for computational imbalance between processes and the corresponding help is given in Figure 2.20.

Periscope [GO10] and PerfExpert [BKD+10] are other automatic tools. They scan performance properties at run-time; appropriate metrics are measured and evaluated automatically. Ultimately, as in Scalasca, this assists in automatic localization of certain types of bottlenecks. However, while all those automatic tools provide some hints, the user might be forced to use a visualization tool such as Vampir to really understand the behavior of the application.

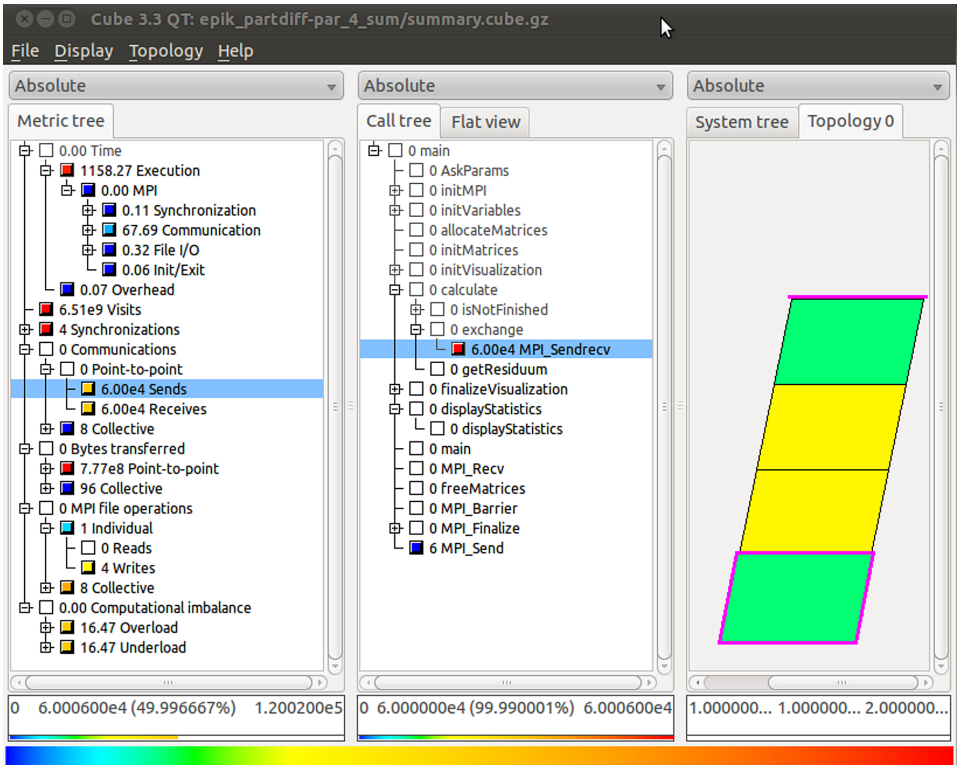


Figure 2.18.: Scalasca's Cube3 browser – the left column shows available metrics, the middle column assigns the metric's values to functions of the call-graph, the right column shows the contribution of every process to the function.

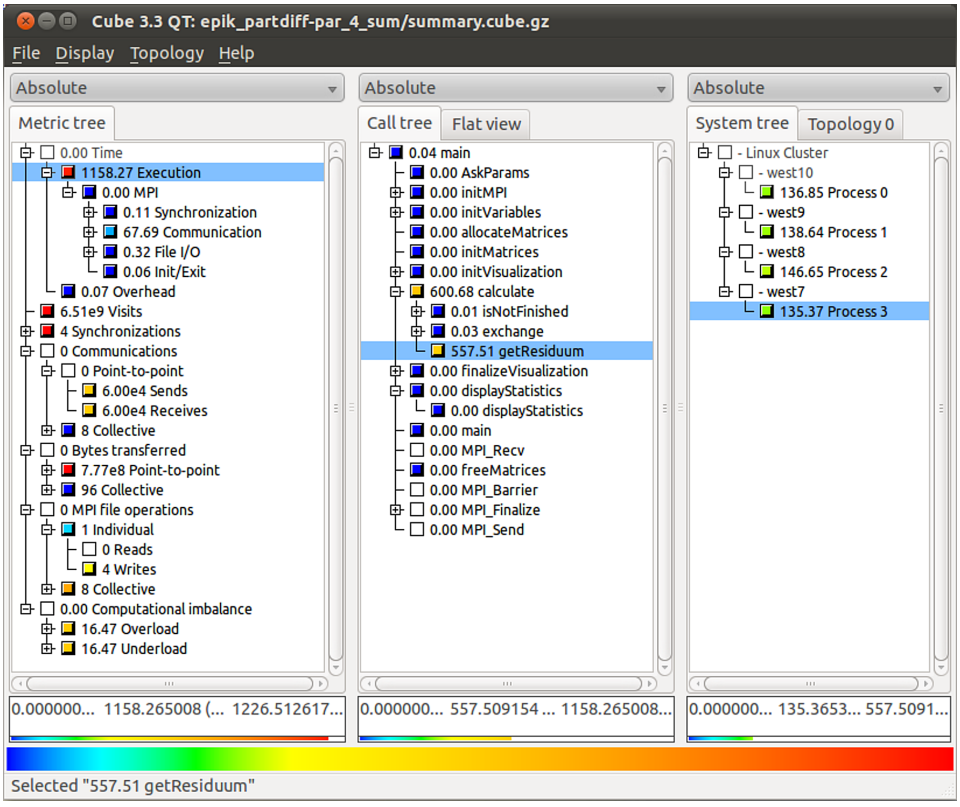


Figure 2.19.: Scalasca's Cube3 browser – identifying the computational overhead in `getResiduum()`.

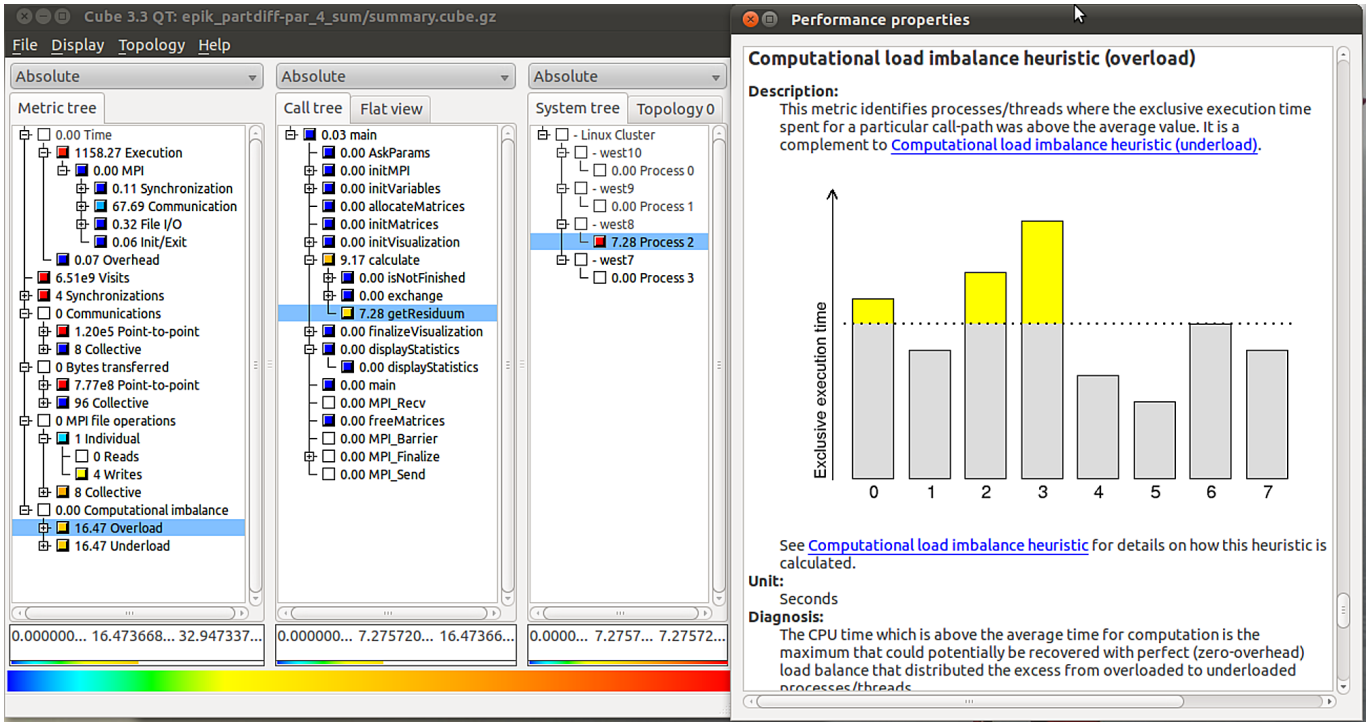


Figure 2.20.: Scalasca's Cube3 – assessing load imbalance and the online help for this metric.

MPE

The *MPI Parallel Environment* (MPE) is a loosely structured library of routines designed to support the parallel programmer in an MPI environment. It includes performance analysis tools for MPI programs, profiling libraries, graphical visualization tools and the trace visualization tool *Jumpshot* [ZLGS99]. MPE is shipped with MPICH-2 and contains different wrapper libraries, which use the PMPI profiling interface of MPI to replace the MPI calls with new functions.

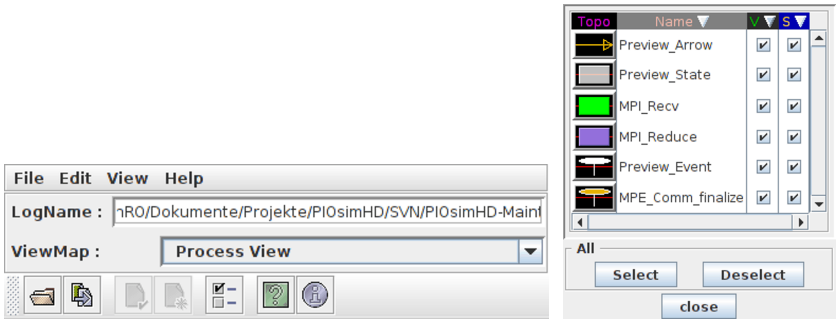
The trace visualizer of MPE is *Jumpshot*, which reads files in the SLOG2 format. SLOG2 tries to be scalable by storing aggregated information about intervals directly inside the format – further information is provided in [ZLGS99].

Upon startup of Jumpshot, the main window is shown in which a trace file can be loaded, a screenshot is given in Figure 2.21a. Jumpshot distinguishes between three types of entries: an *event*, a *state* that has a well defined start and end, and arrows which mark causal relations between states. A trace entry belongs to one named *category*. All available categories, assigned colors and whether they shall be visualized or searchable, are listed in the *legend window* (Figure 2.21b).

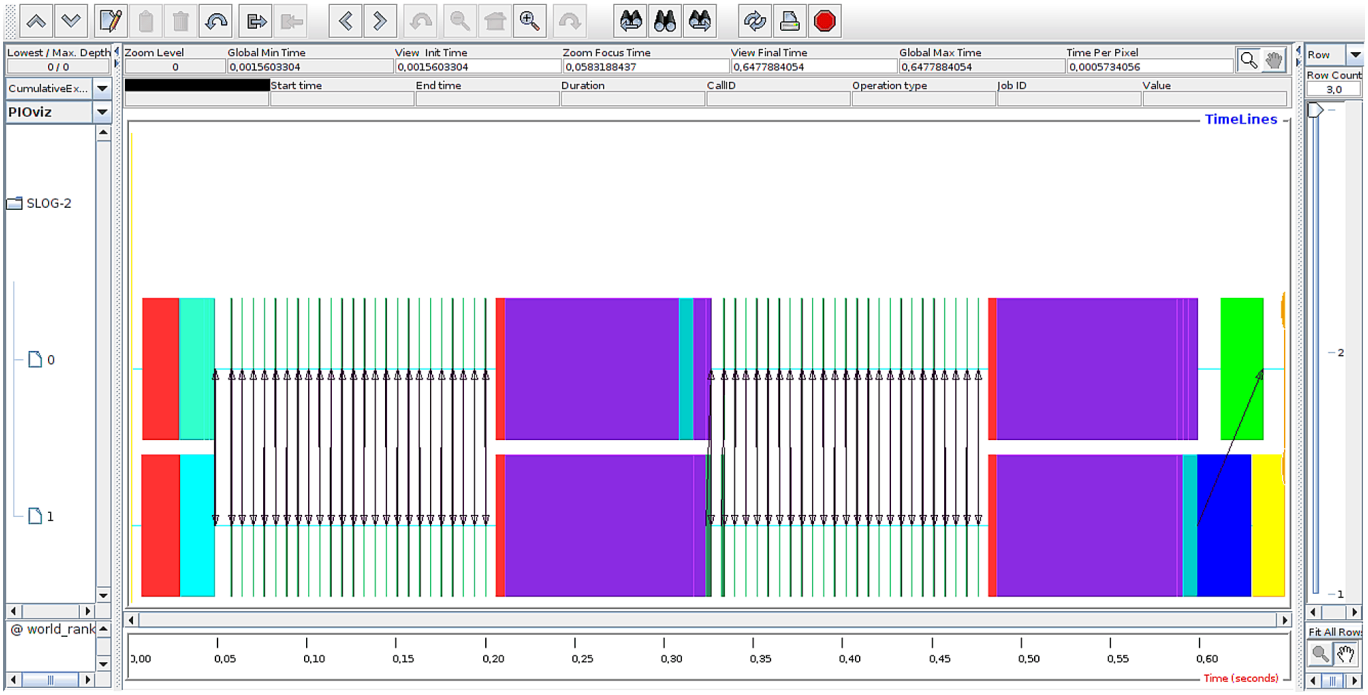
The *timeline window* shows the activity of each process over time. Processes are enumerated in a tree view and mapped to the timelines according to a *ViewMap*. A screenshot of the timeline window is given in Figure 2.21c. On the left side, the tree view shows two processes, the activity is drawn in the center. A horizontal timeline renders the activity of one processor in one row, the activity is encoded with colors as defined in the legend window. White areas in the activity correspond to computation by the client processes, the colors show the MPI function; violet, for example, represents `MPI_Reduce()`.

A user can select an interval and open a *profile window*, which aggregates the time of the states over each category and process. An example is provided in Figure 2.21d; the violet color indicates that most time is spent in `MPI_Reduce()`.

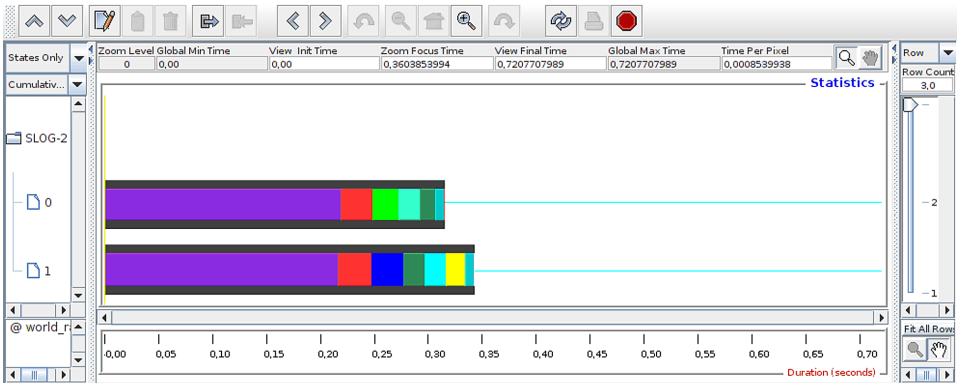
Both windows offer functionality to zoom and scroll in the window, this is provided by the icons in the toolbar. Timelines can be enlarged with the slider on the right of the windows. Additionally, individual timelines can be deleted or moved around; to move a timeline, it must be cut by the user and inserted after



(a) Main window (b) Legend window



(c) Timeline window



(d) Profile window

Figure 2.21.: Jumpshot windows.

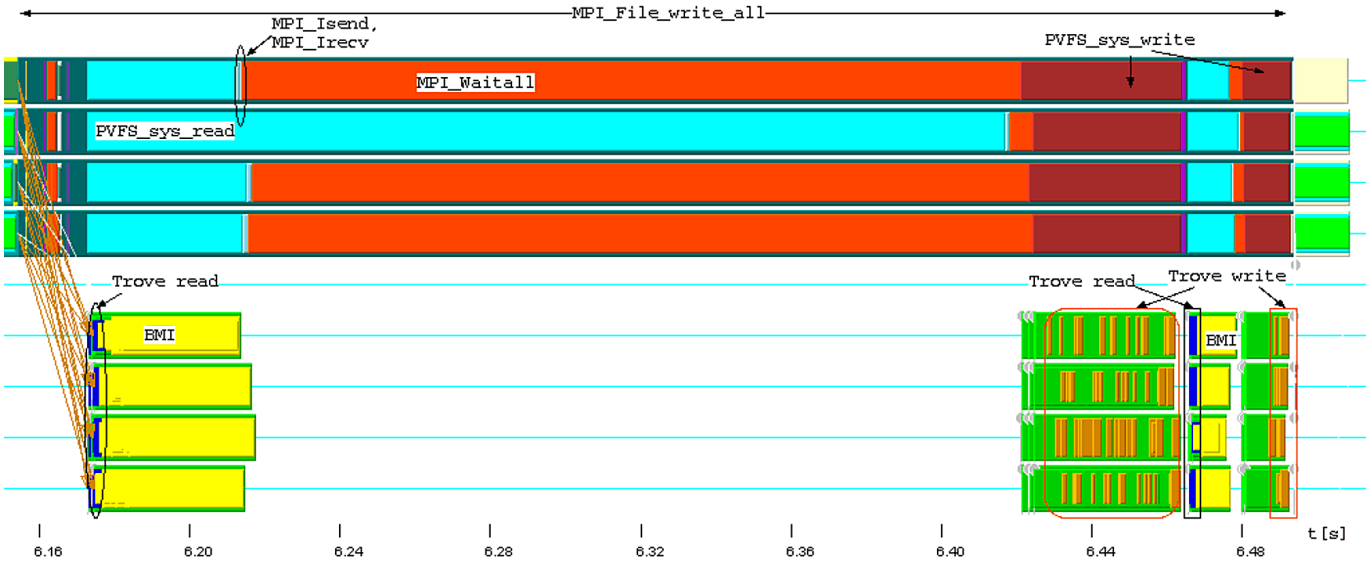


Figure 2.22.: Modified screenshot of PIOviz visualizing the interaction between 4 clients and 4 servers with explanations [KTML09].

another timeline.

PIOviz

The *Parallel Input/Output Visualization Environment* [LKK+06b, KTML09] (PIOviz) is able to trace and visualize activities on the servers of the parallel file system PVFS in conjunction with the client events triggering these activities. PIOviz correlates the behavior of the servers with program events. Developers can use these features to analyze and optimize MPI-IO applications together with PVFS. Additionally, PIOviz also collects device statistics, such as network and disk utilization, from the operating system, and computes a few PVFS-internal statistics [KL08].

The PMPI wrapper provided in MPE is used to trace the MPI function calls. However, with the original MPE only MPI activity is recorded and analyzed. PIOviz extends its capabilities by recording communication inside MPI calls, client-sided PVFS activity, and corresponding operations in PVFS servers. Compared to other tools, PIOviz is considered experimental, however, it provides novel capabilities that are not available elsewhere.

In brief, the environment consists of a set of user-space tools, modifications to the I/O part of MPICH2 and MPE, and logging enhancements to PVFS. To relate client and server activity, the following changes are made: PIOviz modifies the MPE logging wrapper to add a call-ID to each I/O request. Patches to MPI-IO and PVFS transfer this ID to the server and through the different layers, and record interesting information. The call-ID allows us to associate the MPI call with the PVFS operations triggered by this call. Also, the environment introduces additional user-space tools that transform SLOG2 files depending on this extra information. It contains modifications to Jumpshot providing additional information in the viewer.

To assess performance in the workflow, independent trace files are created on client-side and server-side once a user executes an MPI(-IO) program. Then a set of tools post-process these files and finally merges them into a single file containing all enriched information about client and server activities. PIOviz uses MPICH's SLOG2 format, therefore a user can analyze trace information with Jumpshot.

An example screenshot of PIOviz is given in Figure 2.22. Client process activity is given in the upper four timelines, and the lower five timelines show the activity for one PVFS metadata server and four data servers.

By looking at the screenshot it can be observed that the second process spends about 20 ms more time in the `PVFS_sys_read()` operation of the `MPI_File_write_all()` and thus the process waits for the read operation to complete. However, the server activity finished already, therefore, the servers are waiting for requests from clients and are not the cause of the inefficiency. Instead we claim the client library caused the observed behavior. Without knowledge of the server activity, a hypothesis for a potential bottleneck could not exclude the network, the server, or the client-server protocol.

Besides PIOviz, to our knowledge, there is no trace environment available which can gather information of client and server activity and correlates them – a recent funded project aims to extend TAU towards this goal [BCI+10], though.

2.4.5. Trace Formats

There are several trace formats available because many performance analysis tools rely on their own trace format. Usually, command line tools are provided to convert the tool-specific trace format to other well-known trace formats. Therefore, already existing tools can be applied to process the (converted) trace. Scalasca and TAU, for example, provide converters into the *Open Trace Format* (OTF).

A few general aspects in designing a trace format and its interface are discussed at first, the list is a lose collection of aspects and does not aim for completeness. Then, the concepts behind OTF are introduced.

Basically, all trace formats have been created with specific design goals in mind, however, several goals are common to most of them. The following abstract requirements and concepts represent the author's view; they are defined after looking at several trace formats and existing tools[55]:

- Recording of all *relevant information* for post-mortem analysis must be possible. This includes the possibility to record arbitrary data that is necessary to characterize an event in detail. A *context* provides more information about the recorded events or the utilized resources. For example, timestamps are required to understand the temporal causality; an identifier can specify the processor a thread is (currently) executed on. Timestamps are especially difficult: Since local clocks are not as accurate as a primary reference clock, timestamps of different components are slightly incorrect thus sorting events by the locally created timestamp can lead to a wrong order of events. Therefore, either all clocks must be synchronized accurately with one reference clock, or mechanisms are provided that are able to fix incorrect ordering. *Metadata* describes the invariant properties of the environment in which the trace is recorded. This kind of description of system configuration and experiment is important when traces are kept for a long time.
- In heterogeneous environments *portability* between machine architectures becomes relevant.
- Methods to *reduce the trace file size* and to handle large traces should be available. One simple approach that reduces the file size is permit selective activation and deactivation of the run-time tracing, that means either the application activates tracing for the relevant area or the system deactivates itself automatically after a threshold is reached. Another method is to support compression. If that does not help to reduce the file size, then tools should be capable to process large trace files.
- Required post-processing of the trace files should be of *low overhead*. For example, the PIOviz environment relies on several post-processing stages in which the trace data is read completely. For larger traces this is infeasible.
- Analysis of the trace files should require *limited resources*. A subset of the data should be loadable. Loaded information might be a subset of the processes or the record types, or just a restriction of the time interval.
- *Efficient* parallel access. Technically, this can be achieved by splitting the trace information into multiple files which can be accessed independently to prevent locking and synchronization between

[55] The inspected trace formats are RLOG2, SLOG2, TAU trace files and OTF. To encourage further reading in the design goals, two literature references are provided explicitly.: The attempt for CTF [Des10] and design considerations for OTF [KBB+06].