The API provides functions to the programmer to indicate when the computation starts or ends, or when the scientific application main loop occurs. On the one hand, this enables efficient communication to the servers without disturbing application communication. On the other hand, the pace in which data is created and written back is announced to the library. Concluding, ADIO provides a completely new API in which the programmer is forced to deal with I/O related aspects consciously – but due to the XML, system optimizations are possible without source-code modifications.

In our paper [KMKL11], the ADIOS interface is explained in detail. In this paper the interface is extended to offer visualization capabilities and improved energy efficiency.

Tuning library settings MPI libraries like IBM's *Parallel Environment* or Open MPI offer a rich set of environment specific parameters to tune the library internals towards a system or application. For example buffer sizes for the eager message protocol can be adjusted to the network characteristics. In Open MPI, the *Modular Component Architecture (MCA)* provides more than 250 parameters on a COST Beowulf cluster. The libraries provide empirically chosen defaults, which might be determined for a completely different system than the system the library is deployed on. Thus, the defaults might achieve only a fraction of theoretical performance. To provide a starting point for application specific tuning of those values, an administrator should provide appropriate values for the given system.

Chaarawi et.al. developed the *Open Tool for Parameter Optimization (OTPO)* for Open MPI which uses the automatic optimization algorithm from ADCL to determine the best settings of available MCA parameters for a given cluster system [CSGF08]. OTPO could be configured and run by administrators to set up efficient cluster defaults.

2.4. Performance Analysis and Tuning

In computer science, *performance analysis* refers to activity that fosters understanding in timing and resource utilization of applications. In terms of a single computer, the CPU, or to be more formal, each functional unit provided by the CPU, is considered to be a resource. Therefore, understanding resource utilization includes understanding run-time behavior and wall-clock time. For parallel applications, the concurrent computation, communication and parallel I/O increase the complexity of the analysis. Therefore, many components influence the resource utilization and run-time behavior; those have been discussed in Section 2.2.1.

Computational complexity theory is the field of computer science that provides methods to classify and estimate algorithm run-time depending on the problem size. Theoretical analysis of source code is usually infeasible as utilization of hardware at run-time can only be roughly estimated. Therefore, in practice, theoretical analysis is restricted to small code-pieces or clear application kernels, and typically software behavior is measured and assessed.

Programs can be classified according to their utilization characteristics and demand – important algorithms are categorized into 13 *motifs* [ABC+06]. Most applications could be thought of as a combination of the basic functionality required by those motifs. But even so, the characteristics of each real program must be analyzed individually.

In this section, it is first shown how application design and software engineering can assist in developing performance-demanding applications (Section 2.4.1). Those methods focus on integrated and automatic development to achieve efficient and performant applications.

As scientific programs usually require a huge amount of resources, one could expect them to be especially designed for performance. Unfortunately, that is not the case. One reason is that many scientific codes evolved over decades at a time when performance has been of low priority. Usually performance is analyzed after the correctness of the program has been evaluated. At this late stage, a functional version of the

code exists, which has been tested to some extent. Thus, a complete redesign is usually out of reach for the scientist because it is time-consuming.

Once the behavior of an application is understood, it can be modified to make the code more efficient with respect to resource consumption. *Tuning* refers to the iterative process in which the current status is systematically analyzed and optimized, it is described further in Section 2.4.2).

Plenty of tools exist which assist the developer and user in tuning an application and the underlying run-time system according to this *closed loop of performance tuning*. In Section 2.4.3, several tools to analyze sequential programs are introduced, followed by tools suitable for parallel applications in Section 2.4.4. Capabilities of each tool are illustrated on the same parallel code, i.e., all tools are tested with the same application and parameters. The experiment run uses our partial differential equation (PDE) solver `partdiff-par`[37], which implements the Jacobi and Gauß-Seidel methods. The sequential version called *partdiff-seq* excludes the MPI calls; apart from these modifications, the code base is identical.

Since behavior of a program is often analyzed after the program terminates, at last a common file format is introduced, which stores program activity for such a *post-mortem* analysis (Section 2.4.5). Some parts of this chapter have been published in the book [MMK+12].

The simulation environment that is presented later in this thesis will assist in post-mortem performance analysis but also in a performance oriented development of applications. Therefore, understanding these areas is important.

2.4.1. Developing Applications for Performance

In the industry the process of system and application tuning is often referred to as *performance engineering*. Software engineers designed special methods to embed performance engineering into the application development. With these approaches, performance is considered explicitly during the application design and its implementation.

Important existing approaches that embed performance engineering into the development cycle are presented next. Unfortunately, the research and processes in industry are not integrated in state-of-the-art HPC application development, although there are a few tools which assist in the development of parallel applications.

Computer-aided software engineering Tools and methodologies serve the developer during the lifecycle of software. In 1982, the term *Computer-aided software engineering* (CASE) was formed:

> *"Computer-aided software engineering is the scientific application of a set of tools and methods to a software system which is meant to result in high-quality, defect-free, and maintainable software products"* [Wik11], Wikibook: Introduction to Software Engineering/Tools/Modeling and Case Tools

CASE tools assist the programmer in the software development process, which basically consists of the following phases: requirement engineering, analysis, design, coding, documentation and testing. Information of one phase is linked to other phases by those tools. This way, information about early phases is accessible in later phases; for example, the requirements are referenced directly in the source code.

Automatic code generation from a model The *Model-Driven Architecture* (MDA) is a design methodology in which abstract models of the desired application are refined until a code skeleton representing the model is generated. Multiple existing technologies are bundled together with tools to assist in top-down development. For example, developers can create the models in appropriate description languages like the

[37] This PDE solver is frequently used in the lectures of our working group as an exercise to implement various parallel programming models. More information about the program is given in Section 7.9.1.

Unified Modeling Language (UML). UML class diagrams are then translated into Java skeletons; a reverse engineering of modified classes into the model is also possible.

CASE and MDA do not explicitly honor the non-functional requirement for performance. Certainly, due to the difficulty to meet this requirement in later stages of the development, as it is hard to estimate performance of complex applications beforehand.

Integrating performance aspects into code development In the industry, the direction towards design for performance – *Software Performance Engineering* [CUS01] (SPE) – is a newer development. Performance tuning guides or best practices on the other hand, are usually available for all programming languages.

Performance evaluation is anchored into the software development cycle; for example, see [CUS01, BGM03, BFG+04, PW05, SSM+09]. The book [CUS01] discusses performance-oriented design and provides best-practice and a methodology for predicting performance of object-oriented software quantitatively. Microsoft targets performance modeling of the .NET language explicitly in [Cor04] and discusses software development for performance.

Continuous performance management of Java or .NET applications is made available in the product *dynaTrace* [dyn10]. During the entire software life-cycle, the performance is monitored and compared with earlier results to detect performance degradation due to code modification.

The *Tuning and Performance Utilities* have been recently integrated into the *Eclipse Parallel Tools Platform* [SSM+09] to make them available to a wider audience (refer to section 2.4.4 for a description of TAU).

Modeling and estimating performance The *Object Management Group*, the standardization consortium which defined UML, extended the diagrams by defining annotations for performance requirements [Obj03]. Several systems exists which feed these annotations into performance models and simulators, this is discussed in [BGM03, BFG+04, WPP+05, PW05].

For example, in [PW05], interactions between actors and system components are evaluated by converting a use case map into a performance model. *PUMA* [WPP+05] is a tool set which actually translates arbitrary design models with performance annotations into a *Core Scenario Model* (CSM) containing the performance relevant aspects. This CSM can then be evaluated with various performance models. In their paper, an example is given in which behavior is modeled in UML and converted into *Petri Nets* and a *Queueing Network Model*.

Resource and network utilization of distributed systems is accounted for in the *Layered Queueing Network* (LQN) model: An entity performs activities which consume some resource, need time, impose wait delay or communicate with other entities via synchronous or asynchronous messages. As a third possibility, a request can be forwarded to another entity. The LQN model honors resource contention. It has been used to study many kinds of distributed systems [Fra00]. Simple models can be solved analytically to compute mean values of *service time* and *utilization* of it's components. To solve more sophisticated models, simulation is needed.

The Trace-Based Load Characterization (TLC) in [HWRI99] records trace information in a production environment and feeds it into an LQN model.

The integrated modeling and simulation environment CoFluent Studio [CoF09, CoF10] allows designing embedded multiprocessor systems and their software. Developers formulate requirements and specifications and finally specify their algorithm as an abstract sequence of functions. Similar to MDA, the model is synthesized into architecture specific code, but here the functions are considered as well. Then the performance characteristics of software under the given hardware are simulated and analyzed prior to implementation. The system model for embedded systems contains models for computing units, system buses, network routers, point-to-point links, memory and I/O interfaces including schedulers. Full system

simulation of the hardware/software system in a virtual prototype is also supported but does not target HPC.

Designing hardware and software together In hardware/software *co-design*, both the perspective of software and hardware are considered which is especially important for embedded systems [DM94]. This means that the hardware is designed with the software in mind which will run later on the hardware. Through this concurrent development, a synergy is achieved – the hardware supports the required functionality and the software is geared towards the hardware.

With this approach performance critical software functionality can be directly implemented in the hardware. To avoid the high costs involved in the design of an *Application-Specific Integrated Circuit* (ASIC), often *Field-Programmable Gate Array* (FPGA) technology is deployed. An FPGA can be reprogrammed online to implement a particular function.

Recently, hardware/software co-design became a hot topic for designing HPC environments [Ebc05, HMD+10, KWM+10] as it promises to increase energy efficiency of applications and paves the road for exascale:

> *"The traditional model of single-domain research activities where hardware and software techniques are explored in isolation will not address the current challenges."* [KBB+08]

The similarities and differences between HPC and embedded systems are discussed in [HMD+10], for example, in HPC the processor is not designed for a specific application as it could be for embedded systems.

On a side note, another, similar approach to co-design is to configure a system towards the requirements of the software. However, that means the system provides capabilities to be reconfigured according to the demand of the software. In earlier times this was called *configurable computing* [MSHA+97]. Slight reconfiguration and modification of hardware was recently deployed; for example, with the *Fermi* architecture of its GPUs, NVIDIA can partition available memory into shared memory and as local cache, depending on the user configuration [Pat09].

Performance-oriented development of parallel applications An early tool for semi-automatic development of parallel programs was *Hypertool* [WG90]. Hypertool takes a sequential data-partitioned program and generates code for parallel execution based on the user-supplied partitioning. Additionally, performance estimates and quality measures for the code are computed. In their paper, state-of-the-art software shows that the development issues were already a topic in the 90th.

Software Engineering for Parallel Processing [WK94] (SEPP) was an European project which tried to realize the design for performance during development of parallel applications by developing new CASE tools. Among these tools, a simulator should allow rapid prototyping and benchmarking. The contribution to the *EDPEPPS* environment [DZV+97] simulate parallel programs that use the *Parallel Virtual Machine* (PVM) message passing paradigm.

Computer Aided Parallel Software Engineering (CAPSE) was a similar approach, developed in parallel to SEPP, it is discussed in [GHKV96]. N-MAP is an environment developed within CAPSE that predicts application performance already during the development phase [FM95, FJ95]. In [FJ00], N-MAP is claimed to be superior to performance models, because a model abstracts from reality which they state *"is suspect to fail"*. Instead they propose that the developer prototypes the distributed algorithm directly in a N-MAP specification. The N-MAP specification is basically a language dialect derived from C in which inter-process communication is programmed explicitly. Functions can be implemented that represent the real algorithm, or the computation time can be specified by a particular function which might include random variables of some distributions. This implicit performance model is executed inside a simulator that multiplexes a single processor among all parallel processes and executes them sequentially. With this approach, the computation time will be accurately represented iff the real algorithm is encoded. Further, with

this approach, the simulation prototype can be transformed into a real PVM executable. With N-MAP, the workload of application processes can be optimized online: performance metrics are measured by sensors, future states and utilization are estimated, and then appropriate actions are initiated pro-actively.

Pllana et al. [PBXB08] propose performance-oriented program development instead of code-based performance tuning. With *Performance Prophet*, they describe a system which combines mathematical modeling and discrete-event simulation to predict application performance. In their workflow, a developer specifies the program behavior in an UML extension which is geared towards modeling parallel applications. Then, the program is translated into simulator code and executed to predict its performance. The execution time of code blocks is modeled mathematically by assigning a cost function to each code block. Cost functions could be designed by measuring behavior of the code for some inputs and extrapolating them to arbitrary inputs. In their approach, inter-process communication and the control flow are modeled with discrete-events.

All the mentioned tools try to change the way of software engineering to incorporate the performance relevant aspects early in the development cycle. In contrast, the closed loop of performance tuning optimizes programs after a running version exists.

2.4.2. Closed Loop of Performance Tuning

The localization of a performance issue on an existing system is a process in which a hypothesis is supported by measurement and theoretic considerations. Measurement is performed by executing the program while monitoring run-time behavior of the application and the system. In general, tuning is not limited to source code, it could be applied to any system.

The schematic view of the typical iterative optimization process, the *closed loop of performance tuning*, is shown in Figure 2.11:

1. To measure performance in an experiment, the environment consisting of hardware and software including their configuration is chosen, also the appropriate input, i.e., problem statement, is decided. While theoretic considerations allow projecting run-time behavior of arbitrary input sets, monitoring is limited to instances of program input. Thus, it might happen that optimizations made for a particular configuration degrade performance on a different setup or program input. Therefore, multiple experimental setups could be measured together. Often, the measurement itself influences the system by degrading performance, which must be kept in mind. Picking the appropriate measurement tools and granularity can reveal the relevant behavior of the system.

2. In the next step, obtained empirical performance data is analyzed to identify optimization potential in the source code and on the system. As execution of each instruction requires some resources, the code areas must be rated. First, hot-spots – code regions where execution requires significant portions of run-time (or system resources), are identified. Then, optimization potential of the hot spots is assessed based on potential performance gains and the estimated time required to modify the current solution. Knuth describes this issue excellently:

   > *"We should forget about small efficiencies, say about 97% of the time: premature optimization is the root of all evil. Yet we should not pass up our opportunities in that critical 3%. A good programmer will not be lulled into complacency by such reasoning, he will be wise to look carefully at the critical code; but only after that code has been identified."* [Knu79]

   Changing a few code lines to improve run-time by 5% is more efficient than to recode the whole input/output of a program, especially if I/O might account for only 1% of the total run-time. However, care must be taken when the potential is assessed, depending on the overall run-time, a small improvement might be valuable. From the view of the computing facility, decreasing run-time of a program which runs for millions of CPU hours by 1% yields a clear benefit by saving operational costs in form of 10.000 CPU hours (which is about 1.5 CPU years).

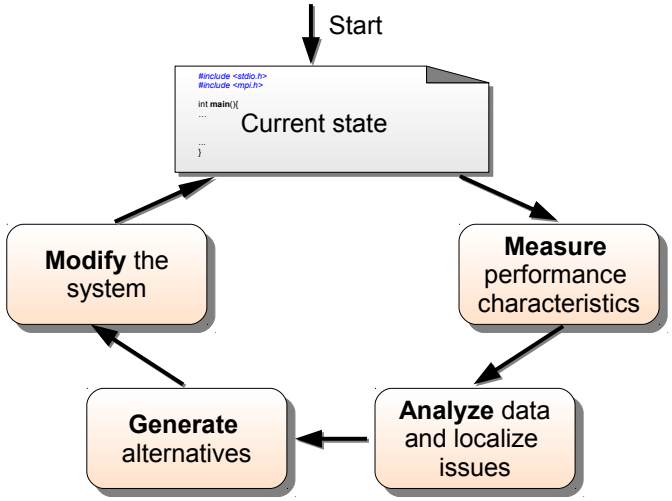


Figure 2.11.: Closed loop cycle of optimization and tuning.

> *"The conventional wisdom shared by many of today's software engineers calls for ignoring efficiency in the small; but I believe this is simply an overreaction to the abuses they see being practiced by penny-wise-and-pound-foolish programmers, who can't debug or maintain their "optimized" programs. In established engineering disciplines a 12% improvement, easily obtained, is never considered marginal and I believe the same viewpoint should prevail in software engineering."* [Knu79]

3. Based on the insight gained by the analysis, alternatives for the implementation are generated, or system modifications are considered, respectively, which apparently mitigate the observed performance issue. This is actually the hardest part of the tuning because it requires that the behavior of the new system can be predicted or estimated. However, often in practice multiple potential alternatives are evaluated, and based on the results the best one is chosen. Sometimes a modification requires modifications that affect other parts of the program and might even degrade overall performance. With increasing experience and knowledge of the person tuning the system, the number of alternatives is reduced as the future behavior can be better anticipated.

4. At the end of a loop the current system is modified, i.e., one of the performance relevant layers is adjusted, to realize the potential improvement of the new design. The system is reevaluated in the next cycle until the time required to change the system outweighs potential improvements, or potential gains are too small because the performance measured is already near-optimal. However, in practice, in most cases the efficiency of the current solution is not estimated; instead, the current run-time is considered to be potentially saved.

Large systems/applications are often complex; thus it is important to reduce the complexity. Therefore, application logic can be reduced to the core of the algorithm – the application *kernel*, which is then implemented in a *benchmark*. Such a benchmark can already provide a rough estimate for running the full application on a system.

In the following paragraphs, a few aspects of the loop are highlighted and discussed: the layers and components that are relevant, then the nature and diversity of performance data, concepts that enable collecting performance data, and the analysis of collected data.

Relevant layers and components In the context of this thesis all of the layers and components participating in an application execution[38] are subject to the optimization process. The information about state and activities of all of them could be important to assess the observations.

[38] See Section 2.2.1 on Page 27 for the layers involved in parallel programming.

Especially, if a component's characteristics contribute to the observed performance to a large extent, understanding its internals might enable tuning of future run-time behavior. To identify reasons for a bottleneck it is mandatory to look inside the particular component (or activity). Without this knowledge, one can deduct only that the process within a particular component causes the suboptimal behavior, but not the reason. It could happen, for instance, that dependencies of the spotted behavior are the cause, while a layer itself is very efficient. Also, *external* (background) activities influence the observable performance. Therefore, during a measurement observations must be accounted to the right activity, otherwise the resource consumption might be attributed to the wrong cause.

Consequently, striving for the best analysis capabilities requires us to obtain a transparent view of all concurrent activities to a very detailed level, however, in a complex system such as a parallel computer this is infeasible because every event must be accurately captured with a global time. Therefore, a promising compromise is to look at relevant activity on all layers and components and the application code, this way performance bottlenecks can be assigned to the particular layer. Furthermore, such a concepts permits identifying the particular application code that limits performance.

The implementation of a *stub*[39] for a layer which pretends to perform the operation allows an implicit performance evaluation of all layers above or below the replaced layer. An example of this technique can be found in [Kun06]; in this thesis the persistency layer of PVFS has been replaced with a stub to reveal the achievable metadata and data performance of PVFS.

Performance data In the closed loop, data is collected which characterizes the application run and system utilization. There are many types of data that can be collected. For example, the operating system provides a rich set of interesting characteristics such as memory, network and CPU usage. These characterize the activity of the whole system, and sometimes usage can even be assigned to individual applications.

The semantics of this data can be of various kinds, usually, a *metric* defines the measurement process and the way subsequent values are obtained. For example, *time* is a simple metric, which indicates the amount of time spent in a program, function or hardware command. The *resident set size* (RSS), i.e., the amount of occupied memory, is another metric.

Depending on the metrics, data can be obtained for a subset or all layers involved in application execution. For example, data can be collected from hardware devices or generated within software, either by the operating system, the application or from additional performance analysis tools.

The expressiveness of a metric depends on its measurement process: A metric can be fine-grained – just describing a single operation executed on a particular component at a given time; or it can be coarse-grained – aggregating all activity that is observed on a given component or even on the whole system.

One way of managing performance information is to store *statistics*, e.g., absolute values like number of function invocations, utilization of a component, average execution time of a function, or performed floating point operations. Statistics of the activity of a program is referred to as *profile*. A profile aggregates events by a given metric, for example by summing up the inclusive duration of function calls. In contrast to a *profile*, a *trace* records events of a program together with a timestamp and thus it provides the exact execution chronology and, therewith, allows analysis of temporal dependencies. External metrics like hardware performance can be integrated into traces as well. Tracing of behavior produces much more data, potentially degrading performance and distorting attempts of the user to analyze observation; therefore, in many cases only profiles are recorded. A combination of both approaches can be applied to reduce the overhead while still offering enough information for the analysis: Events that happen during a timespan can be recorded periodically as a profile for an interval – this allows analysis of temporal variability; By generating profiles for disjoint code regions, behavior of the different program phases can be assessed.

Once a way of aggregation is chosen, the performance data must be correlated to the interesting application's behavior and source code. Depending on the measurement process the assignment of information to

[39] In software engineering a stub refers to code that implements a method (or interface). It may simulate the behavior of existing code for some inputs to ease testing of other components that rely on the method (or interface).

the cause can be impossible, for example a statistic cannot reveal the contribution of concurrent activity.

Typically, it is not possible to gather detailed information for hardware activity because that would imply much overhead. Hardware sensors are available in some devices, which measure internal utilization or other metrics like energy consumption or error rate. For example, all decent consumer CPUs have built-in programmable performance counters. Depending on the CPU architecture, a wide range of counters are available which measure efficiency of cache and branch-prediction, the number of instructions run or the number of floating point operations performed. Modern processors provide a variety of over one hundred counters covering different aspects of a CPU. However, using performance counters of a CPU for analysis has a flaw – the CPUs provide only a limited number of registers to manage the counter values. Effectively only 4 to 6 counters can be active at any given time. This issue can be tackled by multiplexing the metrics from time to time and thus over long running processes a good estimate of the values can be provided. Some of the recorded counters actually capture the aggregate events for the full microprocessor, that means multiple physical CPUs share one event. Consequently, this statistical data cannot be associated with a particular process. Newer network interface cards accumulate the number of packets and the amount of data received and transmitted.

Collecting performance data There are several approaches of measuring the performance of a given application. A *monitor* is a system which collects data about the program execution. Approaches could be classified based on *where*, *when* and *how* run-time behavior is monitored.

A monitor might be capable of recording activities within an application (e.g., function calls), across used libraries, activities within the operating system such as interrupts, or it may track hardware activities; in principle, data can be collected from all components or layers described in Figure 2.7. Most *monitors* rely on software to measure the state of the system, data from available hardware sensors is usually queried from the software on-demand. Hardware monitors are too expensive, complicated and inflexible to capture program activity in detail.

Usually, changes are made to the program under inspection to increase analysis capabilities; the activity that alters a program is called *instrumentation*. Popular methods are to modify source code, to relink object files with patched functions or to modify machine code directly [SMAb01]. During execution, such a modified program invokes functions of the monitoring environment to provide additional information about the program execution and the system state. The described instrumentation functionality could also be supported directly by the (operating) system and, therewith, it could be possible to collect performance data without modifying the application.

As a software monitor requires certain resources to perform its duty (those can be considered as overhead), monitoring of an application perturbs the original execution. Observed data must be kept in memory and might be flushed to disk if memory space does not suffice. Additionally, computation is required to update the performance data. The overhead depends on the characteristics of the application and system – it might perturb behavior of the instrumented application so much that an assessment of the original behavior is impossible. Therefore, to reduce the overhead users enable only a subset of the potential features in a typical optimization setup.

Automatic instrumentation by tools usually tries to gather as much information as possible, therefore, the overhead is higher than with an approach in which the user modifies the source code manually. Normally, the user starts with an automatic instrumentation, then if the overhead is too high filters are applied until the trace file includes just enough information for the analysis. Some tools automatically filter activity if an event is fired to often, if the overhead of the measurement system itself grows too high. If filtering still incurs too much overhead, then interesting functions can be manually instrumented, i.e., by inserting calls to the monitoring interface by hand.

Additionally, a selective activation of the monitor can significantly reduce the amount of recorded data. Also, a monitor could sample events at a lower frequency, reducing the overhead and the trace detail level on the same extent.

Analyzing data Users analyze the data recorded by the monitoring system to localize optimization potential. Performance data is either recorded during program execution and assessed after the application finished, this approach of *post-mortem* analysis is also referred to as *offline* analysis. An advantage of this methodology is that data can be analyzed multiple times and compared with older results. Another approach is to gather and assess data *online* – while the program runs. This way feedback is provided immediately to the user, who could adjust settings to the monitoring environment depending on the results.

Due to the vast amount of data, sophisticated tools are required to localize performance issues of the system and correlate them with application behavior and finally identify source code causing them. Tools operate either manually, i.e., the user must inspect the data himself; a *semi-automatic tool* could give hints to the user where abnormalities or inefficiencies are found, or try to assess data automatically. Tool environments, which localize and tune code automatically, without user interaction, are on the wishlist of all programmers. However, due to the system and application complexity those are only applicable for a very small set of problems. Usually, tools offer analysis capability in several *views* or *displays*, each relevant to a particular type of analysis.

2.4.3. Available Tools for Analysis of Sequential Programs

There exist plenty of tools that assist in performance analysis and optimizations of sequential code, a handful of tools of different categories are briefly introduced: *GNU gprof* generates a profile of the application in user-space. *OProfile* can record and investigate application and system behavior including activity of the Linux kernel. CPU counters can be related to the individual operations. *PAPI* is a library which accesses CPU counters and provides additional hardware statistics. *Likwid* is a lightweight tool suite that reads CPU counters for an application. *LTTng* traces and visualizes activity of processes and within the kernel. However, compared to OProfile symbolic information of the application program is not supported.

All tools mentioned are licensed under an open and free license. The state of the latest stable versions available is discussed as of February 2011.

GNU gprof

GNU compilers can be instructed to include code into a program that will periodically collect samples of the program counter. During run-time profiles of function call timings and the *call graph*[40] are stored in a file. This profiling data can then be analyzed by the command line tool `gprof`.

To demonstrate the application of the tool, an excerpt of the `gprof` output for a run of the `partdiff-seq` PDE solver is given in Listing 2.1. In the *flat profile* (up to Line 12), the time is shown per function. While 6 functions have been invoked once (Lines 6 to 11), all run-time is spent in the function `calculate()` (Line 6). The textual representation of the call graph is also provided starting with Line 15 of the output. In Line 20 and Line 21, it is shown that the function `calculate()` is called from main once, additional invocations from different functions would generate further sections. The main function calls all 6 subroutines (Lines 24-30).

Most platforms provide tools alike to `gprof` to analyze performance of sequential programs. To analyze a parallel application, a profiler must be aware of the parallelism and provide an approach to handle it, for example, by generating one output for each of the spawned processes.

Listing 2.1: Excerpt of a `gprof` output for partdiff-seq

```
1  Flat profile:
2
3   Each sample counts as 0.01 seconds.
4    %   cumulative   self              self     total
5   time   seconds   seconds    calls   s/call   s/call  name
6  100.05     30.95    30.95        1    30.95    30.95  calculate
7    0.00     30.95     0.00        1     0.00     0.00  AskParams
```

[40] The call graph is a directed graph providing information about function invocation, the nodes of the graph represent functions and the edges function calls.

```
  0.00     30.95     0.00        1     0.00     0.00  DisplayMatrix
  0.00     30.95     0.00        1     0.00     0.00  allocateMatrices
  0.00     30.95     0.00        1     0.00     0.00  displayStatistics
  0.00     30.95     0.00        1     0.00     0.00  initMatrices

[...]

                     Call graph (explanation follows)

granularity: each sample hit covers 2 byte(s) for 0.03% of 30.95 seconds

index % time    self  children    called     name
                30.95    0.00       1/1           main [2]
[1]    100.0   30.95    0.00       1         calculate [1]
-----------------------------------------------
                                                  <spontaneous>
[2]    100.0    0.00   30.95                 main [2]
                30.95    0.00       1/1           calculate [1]
                 0.00    0.00       1/1           AskParams [3]
                 0.00    0.00       1/1           initMatrices [7]
                 0.00    0.00       1/1           allocateMatrices [5]
                 0.00    0.00       1/1           displayStatistics [6]
                 0.00    0.00       1/1           DisplayMatrix [4]
```

OProfile

OProfile[41] provides a sophisticated system-level profiling for the Linux operating system. Compared to `gprof`, OProfile gathers information from all running processes at the same time. Also, kernel internals are captured, and a configurable set of hardware performance counters. Profiling must be enabled and started by the super user, then all activities are recorded. Several tools are provided that analyze data recorded from user-space.

An example for system-level profiling of a desktop system running our PDE is given in listing 2.2. In this profile, the concurrent activities can be identified, also the time spent in kernel space[42] and in certain libraries becomes apparent. For each application, the user can create an individual profile, covering activities of the particular application as well as activities triggered by library and system calls.

A very handy feature of the OProfile system is that source code (and additionally assembler) can be annotated with the performance observations. This makes it easier to localize the time-consuming lines. In Listing 2.3, the source code of `calculate()` is shown with the sample count. The time spent in individual code lines becomes apparent, for example, 12% of the run-time is spent in Line 20, showing the optimization potential within this line.

Accessible hardware counters on the desktop system are shown in Listing 2.4[43].

Listing 2.2: Excerpt of a system-wide OProfile output while running partdiff-seq

```

CPU: Intel Architectural Perfmon, speed 1199 MHz (estimated)
Counted CPU_CLK_UNHALTED events (Clock cycles when not halted) with a unit mask of 0x00 (No unit mask) count 100000
samples  %        image name               app name                 symbol name
1068951  71.0457  partdiff-seq             partdiff-seq             calculate
185685   12.3412  no-vmlinux               no-vmlinux               /no-vmlinux
26927     1.7896  libgstflump3dec.so       libgstflump3dec.so       /usr/lib/gstreamer-0.10/libgstflump3dec.so
16738     1.1125  libspeexdsp.so.1.5.0     libspeexdsp.so.1.5.0     /usr/lib/libspeexdsp.so.1.5.0
13980     0.9292  Xorg                     Xorg                     /usr/bin/Xorg
12617     0.8386  libQtGui.so.4.7.0        libQtGui.so.4.7.0        /usr/lib/libQtGui.so.4.7.0
12110     0.8049  libQtCore.so.4.7.0       libQtCore.so.4.7.0       /usr/lib/libQtCore.so.4.7.0
10177     0.6764  libxul.so                libxul.so                /usr/lib/firefox-3.6.13/libxul.so
8375      0.5566  libmozjs.so              libmozjs.so              /usr/lib/firefox-3.6.13/libmozjs.so
8311      0.5524  libflashplayer.so        libflashplayer.so        /usr/lib/flashplugin-installer/libflashplayer.so
6670      0.4433  libdrm_intel.so.1.0.0    libdrm_intel.so.1.0.0    /lib/libdrm_intel.so.1.0.0
6097      0.4052  libpulsecommon-0.9.21.so libpulsecommon-0.9.21.so /usr/lib/libpulsecommon-0.9.21.so
4185      0.2781  oprofiled                oprofiled                /usr/bin/oprofiled
4111      0.2732  libpthread-2.12.1.so     libpthread-2.12.1.so     pthread_mutex_lock
3819      0.2538  libgstreamer-0.10.so.0.26.0 libgstreamer-0.10.so.0.26.0 /usr/lib/libgstreamer-0.10.so.0.26.0
3255      0.2163  libpthread-2.12.1.so     libpthread-2.12.1.so     pthread_mutex_unlock
```

Listing 2.3: Excerpt of the annotated partdiff-seq source code

```
             :/* ********************************************************************** */
```

[41] Visit `http://oprofile.sourceforge.net/` for further information.

[42] By providing the kernel symbol table, all activity inside the kernel is accounted in a fine-grained manner to the kernel-internal symbols; in the listing the `no-vmlinux` symbol aggregates all kernel activities.

[43] The output was created by using `opcontrol -list-events`.

```
2                :/* calculate: solves the equation                                    */
3                :/* ******************************************************************* */
4                :void calculate(void)
5                :{ /* calculate total: 1068951 99.9979 */
6                :  int i,j;                                    /* local variables for loops  */
7                :
8                :  while(term_iteration>0)
9                :  {
10      2 1.9e-04 :    maxresiduum=0;
11   5088  0.4760 :    for(i=1;i<N;i++)                                  /* over all rows     */
12               :    {                                                 /*                   */
13  74459  6.9655 :      for(j=1;j<N;j++)                                /* over all columns  */
14               :      {
15 484776 45.3497 :        star=                    -Matrix[m2][i-1][j]
16               :           -Matrix[m2][i][j-1]   -Matrix[m2][i][j+1]
17               :                                 -Matrix[m2][i+1][j]  +4.0*Matrix[m2][i][j];
18               :        residuum=getResiduum(i,j);
19     315  0.0295 :        korrektur=residuum;
20  134782 12.6086 :        residuum = (residuum<0) ? -residuum:  residuum; // if (residuum<0) residuum=residuum*(-1);
21  135906 12.7137 :        maxresiduum = (residuum < maxresiduum) ? maxresiduum : residuum;
22               :
23   66943  6.2624 :        Matrix[m1][i][j]=Matrix[m2][i][j]+korrektur;
24               :      }
25               :    }
26               :    stat_iteration=stat_iteration+1;
27      56  0.0052 :    stat_precision=maxresiduum;
28               :    checkQuit();
29               :  }
30               :}
31               :
```

Listing 2.4: Available OProfile events (accessible hardware counters) on an Intel Nehalem system

```
1
2  OProfile: available events for CPU type "Intel␣Architectural␣Perfmon"
3
4  See Intel 64 and IA-32 Architectures Software Developer's Manual
5  Volume 3B (Document 253669) Chapter 18 for architectural perfmon events
6  This is a limited set of fallback events because oprofile doesn't know your CPU
7  CPU_CLK_UNHALTED: (counter: all)
8          Clock cycles when not halted (min count: 6000)
9  INST_RETIRED: (counter: all)
10         number of instructions retired (min count: 6000)
11 LLC_MISSES: (counter: all)
12         Last level cache demand requests from this core that missed the LLC (min count: 6000)
13         Unit masks (default 0x41)
14         ----------
15         0x41: No unit mask
16 LLC_REFS: (counter: all)
17         Last level cache demand requests from this core (min count: 6000)
18         Unit masks (default 0x4f)
19         ----------
20         0x4f: No unit mask
21 BR_INST_RETIRED: (counter: all)
22         number of branch instructions retired (min count: 500)
23 BR_MISS_PRED_RETIRED: (counter: all)
24         number of mispredicted branches retired (precise) (min count: 500)
```

PAPI and Likwid

There are several approaches that access CPU counters. *PAPI*, the *Performance API* [MCW+05, TJYD09], is a portable library which enables programs to gather performance events of all modern x86 and Power processors.

PAPI evolved from an interface for CPU counters to the component-oriented PAPI-C [TJYD10], which extends the original PAPI to capture counters from a multitude of sources – ACPI, `lm_sensors` for temperature, or specific network interfaces (Myrinet). PAPI-C leverages the existing inhomogeneous vendor (and software) interfaces.

An alternative library and user space program to profile hardware counters for an application is Likwid [THW10]. Likwid is a user space tool that profiles hardware counters for the whole application execution. When the tool is started, it initializes the counters, then starts the application, and once the application terminates, the counters are stopped and a brief report is created. A small API is offered that allows a developer to restrict the measured code regions and, furthermore, it supports to split execution into phases that can be assessed individually.

An exemplary profile of the floating point group of `partdiff-seq` is given in Listing 2.5. In this example the Intel processor operated on an average clock of 3.427 GHz (Line 28), the number of *cycles per*

instruction(CPI) is 0.52, which means every other cycle one instruction is *retired*[44] on the core. The number of double precision floating point operations per second is about 1.7 GFlop/s. One can easily conduct from the average clock speed and CPI that if we execute one instruction every other cycle, then effectively 1.7 Gigainstructions are executed per second, which means most operations were floating point instructions. This little example already demonstrates the power of hardware counters and simple theoretic considerations.

Available groups for Likwid are shown in Listing 2.6. The group to measure is selected upon startup of Likwid. Note that the memory group is aggregated for all cores on a given chip.

Listing 2.5: Excerpt of the likwid output for partdiff-seq

```
1
2  -------------------------------------------------------------
3  -------------------------------------------------------------
4  CPU type:       Intel Core Westmere processor
5  CPU clock:      2.79 GHz
6  Measuring group FLOPS_DP
7  -------------------------------------------------------------
8  /opt/likwid/bin/likwid-pin -c 1 ./partdiff-seq 0 2 100 1 2 10000
9  [likwid-pin] Main PID -> core 1 - OK
10
11 [...]
12 < program output >
13 [...]
14
15 +--------------------------------------+-------------+
16 |                Event                 |   core 1    |
17 +--------------------------------------+-------------+
18 |           INSTR_RETIRED_ANY          | 2.02167e+11 |
19 |         CPU_CLK_UNHALTED_CORE        | 1.04973e+11 |
20 |         CPU_CLK_UNHALTED_REF         | 8.55369e+10 |
21 |    FP_COMP_OPS_EXE_SSE_FP_PACKED     | 1.16896e+06 |
22 |    FP_COMP_OPS_EXE_SSE_FP_SCALAR     | 5.22953e+10 |
23 | FP_COMP_OPS_EXE_SSE_SINGLE_PRECISION | 1.0281e+07  |
24 | FP_COMP_OPS_EXE_SSE_DOUBLE_PRECISION | 5.22862e+10 |
25 +--------------------------------------+-------------+
26 +-------------------------+-----------+
27 |          Metric         |  core 1   |
28 +-------------------------+-----------+
29 |       Runtime [s]       |  37.5892  |
30 |       Clock [MHz]       |  3427.22  |
31 |           CPI           |  0.51924  |
32 | DP MFlops/s (DP assumed) |  1696.62  |
33 |      Packed MUOPS/s     |  0.037923 |
34 |      Scalar MUOPS/s     |  1696.55  |
35 |        SP MUOPS/s       |  0.333533 |
36 |        DP MUOPS/s       |  1696.25  |
37 +-------------------------+-----------+
```

Listing 2.6: Available Likwid groups on an Intel Nehalem system

```
1  BRANCH: Branch prediction miss rate/ratio
2  CACHE: Data cache miss rate/ratio
3  CLOCK: Clock of cores
4  DATA: Load to store ratio
5  FLOPS_DP: Double Precision MFlops/s
6  FLOPS_SP: Single Precision MFlops/s
7  FLOPS_X87: X87 MFlops/s
8  L2: L2 cache bandwidth in MBytes/s
9  L2CACHE: L2 cache miss rate/ratio
10 L3: L3 cache bandwidth in MBytes/s
11 L3CACHE: L3 cache miss rate/ratio
12 MEM: Main memory bandwidth in MBytes/s
13 TLB: TLB miss rate/ratio
14 VIEW: Double Precision MFlops/s
```

LTTng

The *Linux Trace Toolkit*[45] provides a user-space and a kernel-space tracer [FDD09]. The user space tracer (UST) provides an API by which a developer can instrument the source code. Also, the *GNU Project De-*

[44] Many modern CPUs execute operations that might depend on operations that are still in the pipeline, for example, speculative execution of branches before the branch condition is actually evaluated; if the prediction is correct, the results of the executed instructions are stored, otherwise these results are invalid and must be discarded. When there are no conflicts, a processor retires the instruction allowing write back of the results. For the speed of the execution the speculatively executed instructions are irrelevant, therefore, they are not covered by the CPI metric.

[45] Documentation of the *Linux Trace Toolkit* including all tools is elaborate and available on `http://lttng.org/`. A quick-start tutorial can be found here: `http://omappedia.com/wiki/Using_LTTng`.

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.

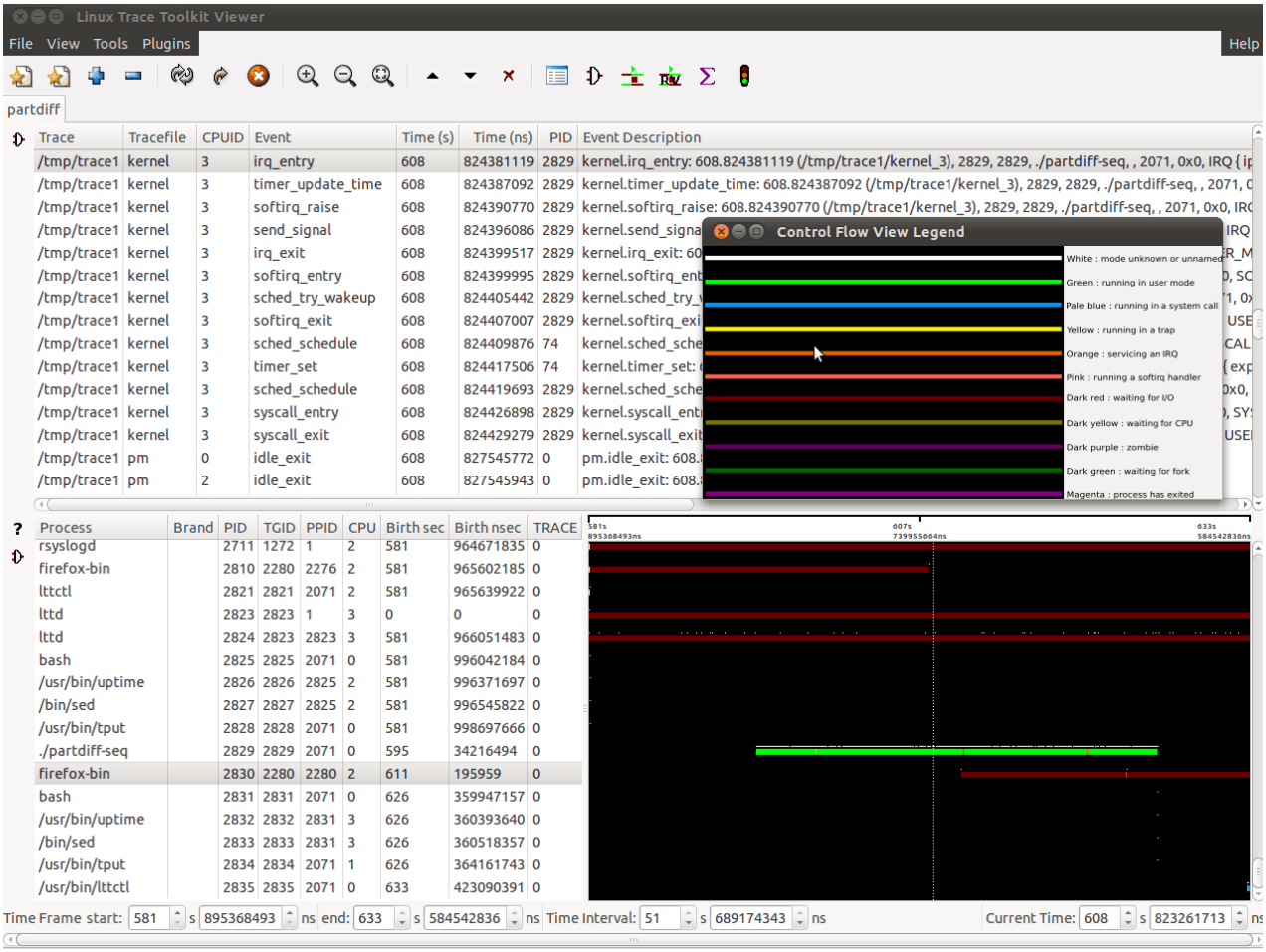


Figure 2.12.: Screenshot of the LTTV viewer for a trace of the system – event view and control flow view.

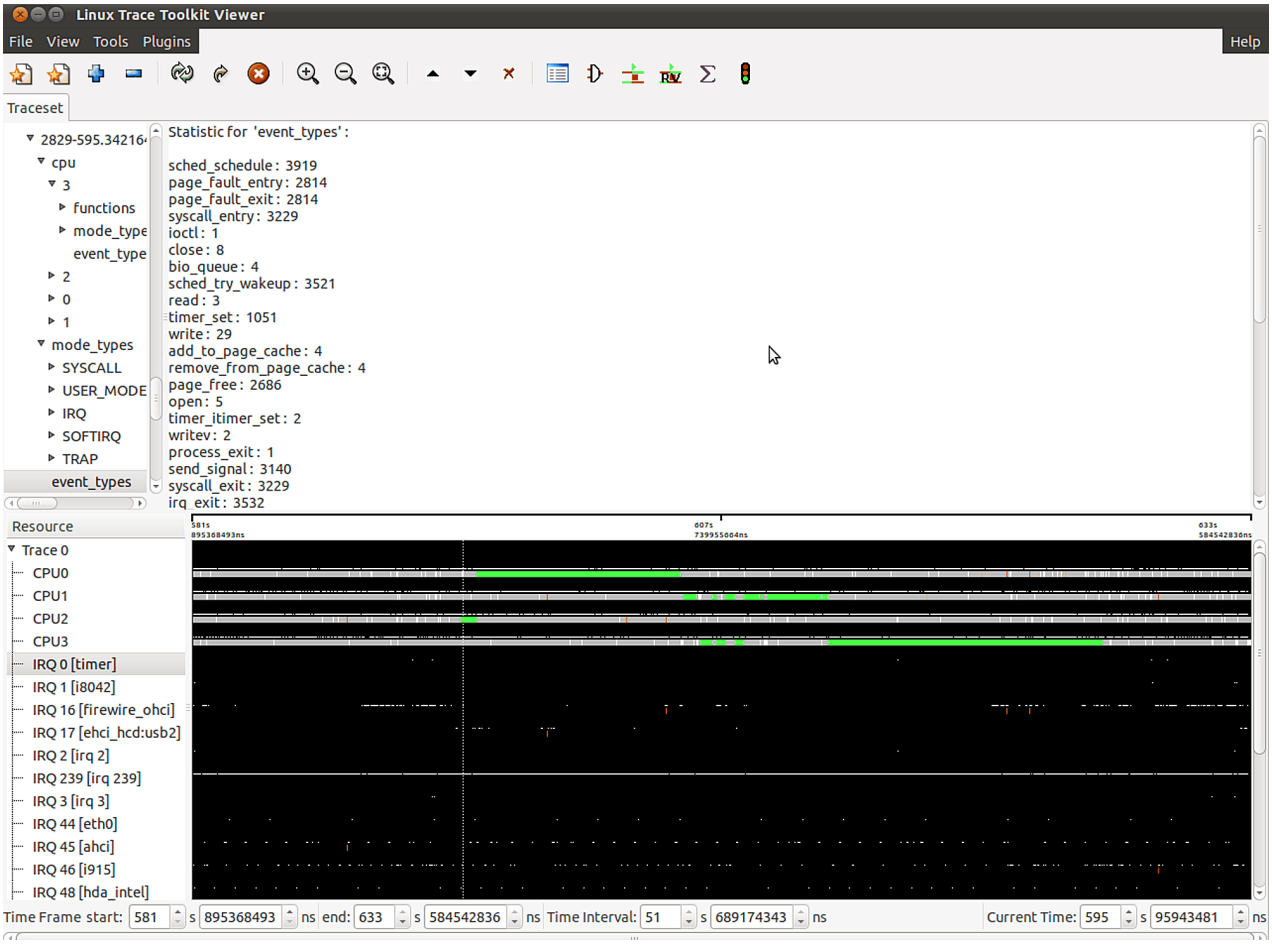


Figure 2.13.: Screenshot of the LTTV viewer for a trace of the system – statistic view and resource view. The likely scheduled execution of the program on the CPUs is marked in green.

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.

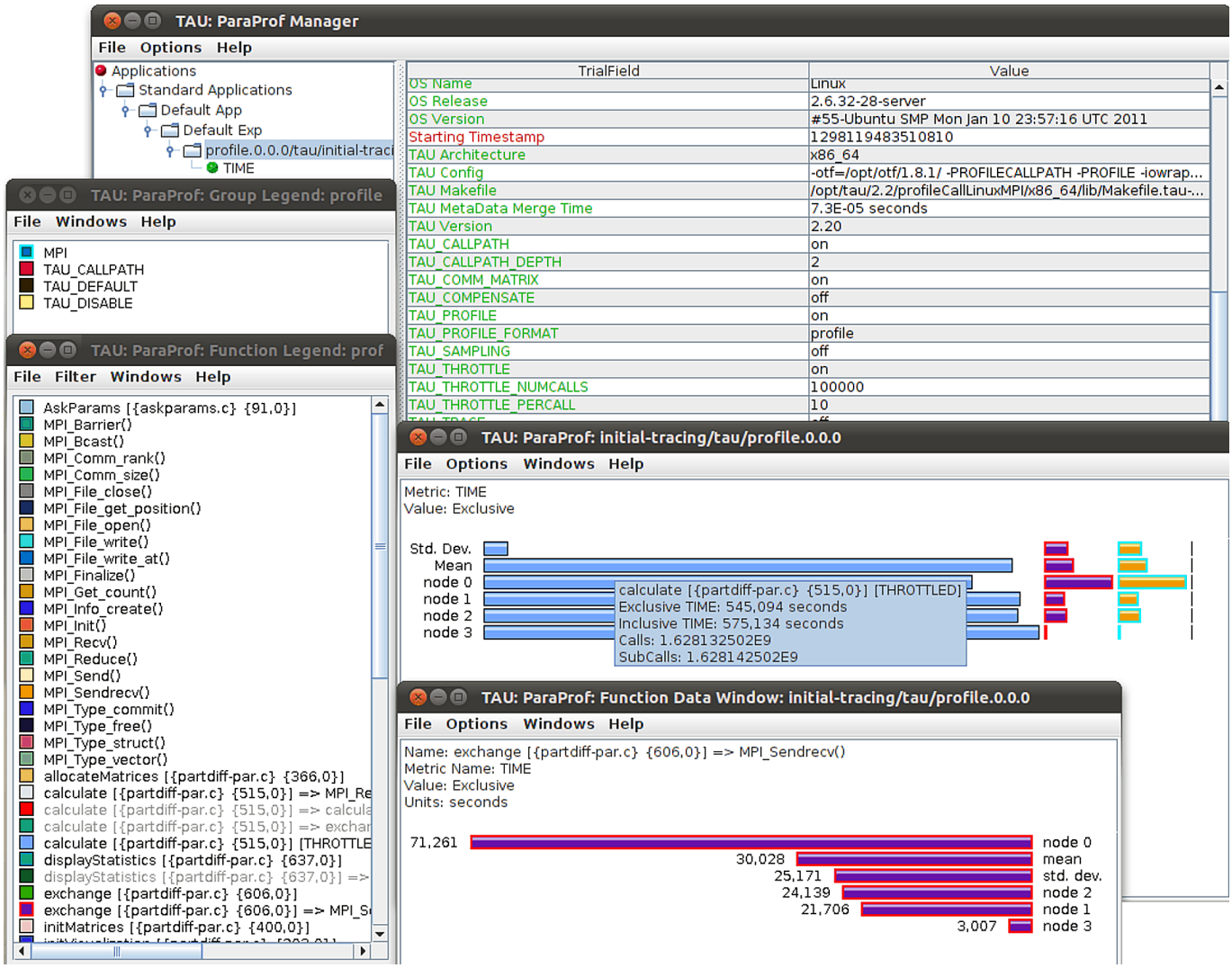


Figure 2.14.: Screenshot of ParaProf – PDE profile including experimental information.

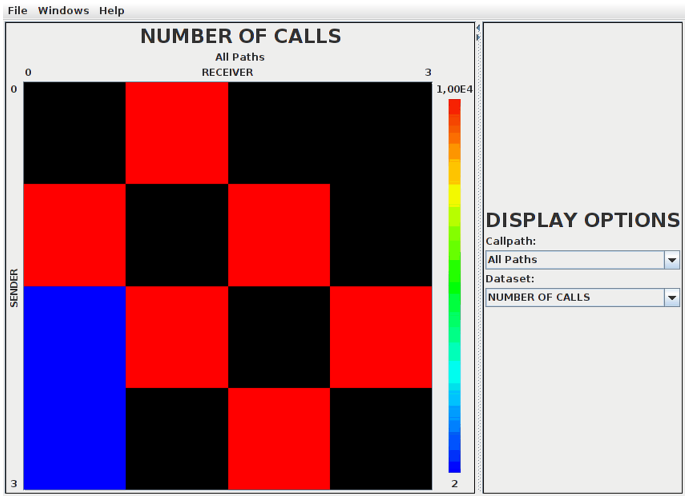


Figure 2.15.: Screenshot of ParaProf – PDE communication matrix.

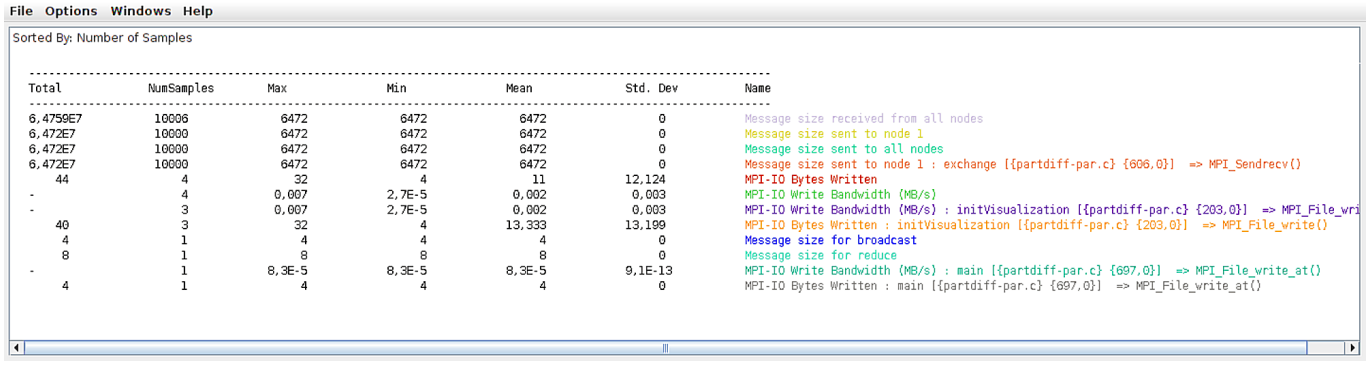


Figure 2.16.: Screenshot of ParaProf – *user event statistics* for Process 0 including MPI-IO statistics.

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*.

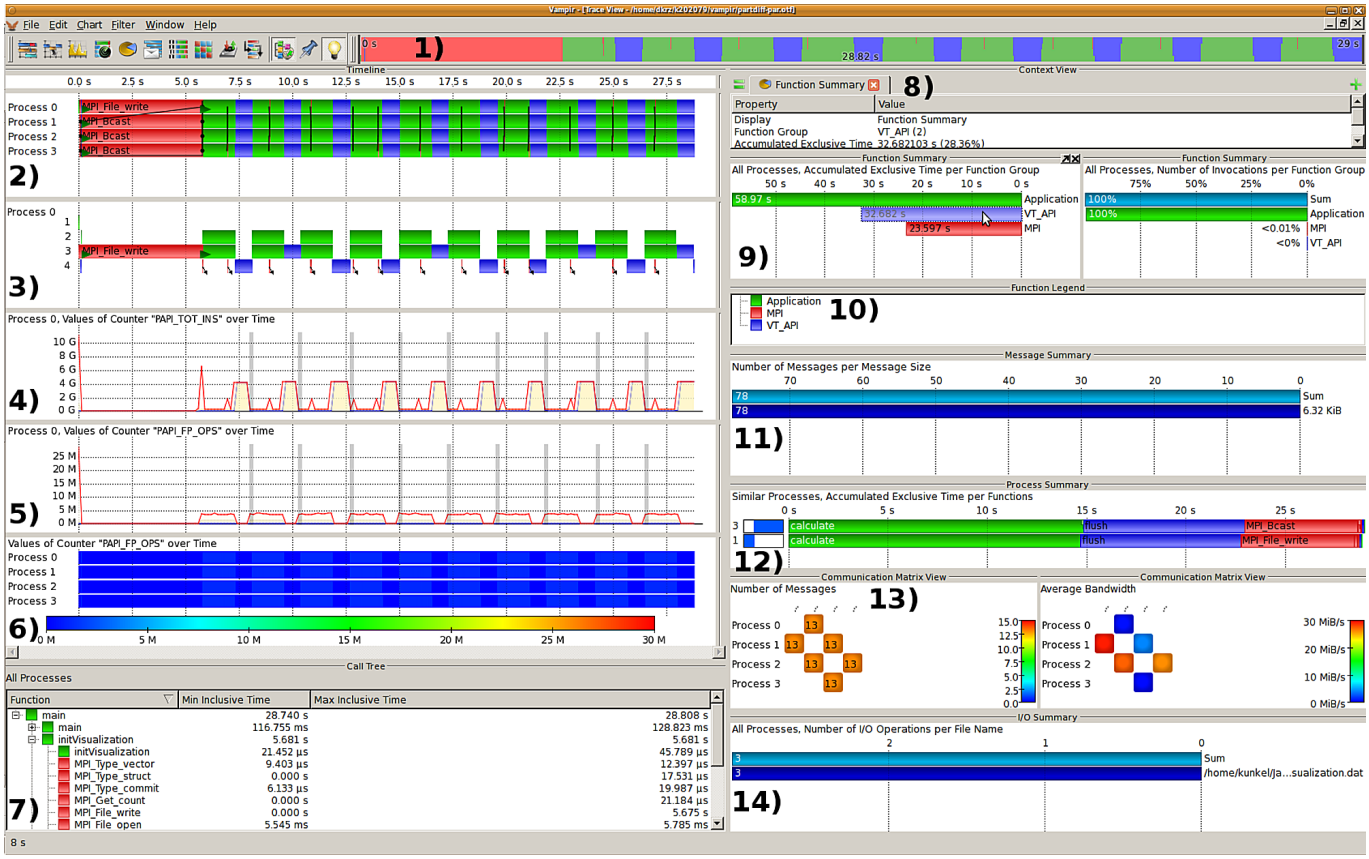


Figure 2.17.: Screenshot of a Vampir workspace.

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.

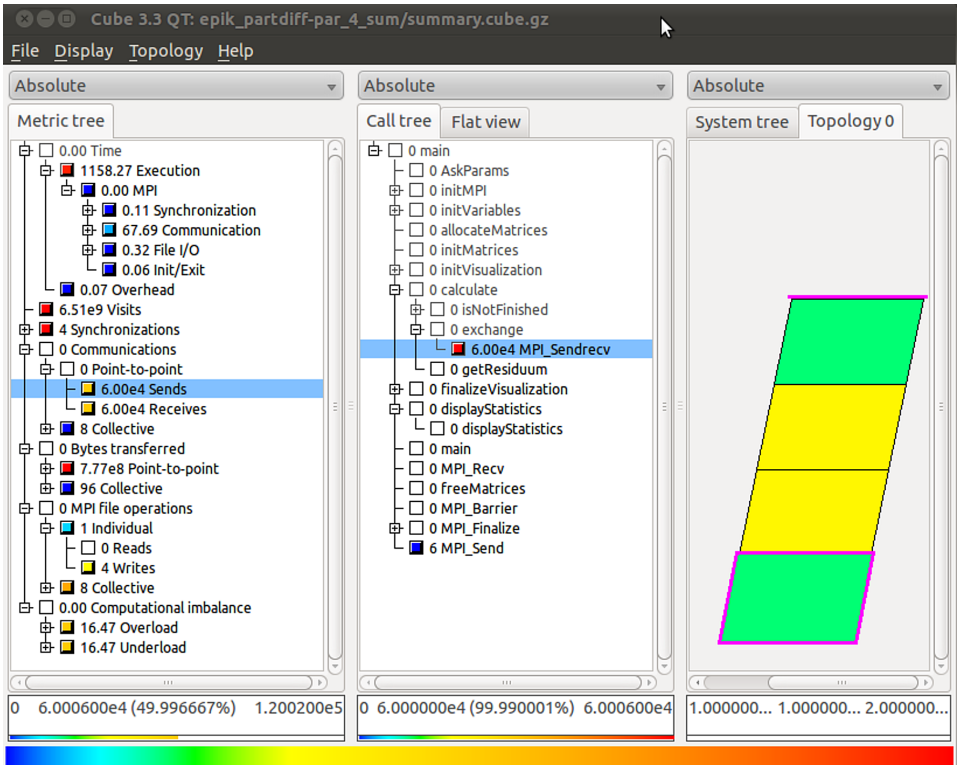


Figure 2.18.: Scalasca's Cube3 browser – the left column shows available metrics, the middle column assigns the metric's values to functions of the call-graph, the right column shows the contribution of every process to the function.

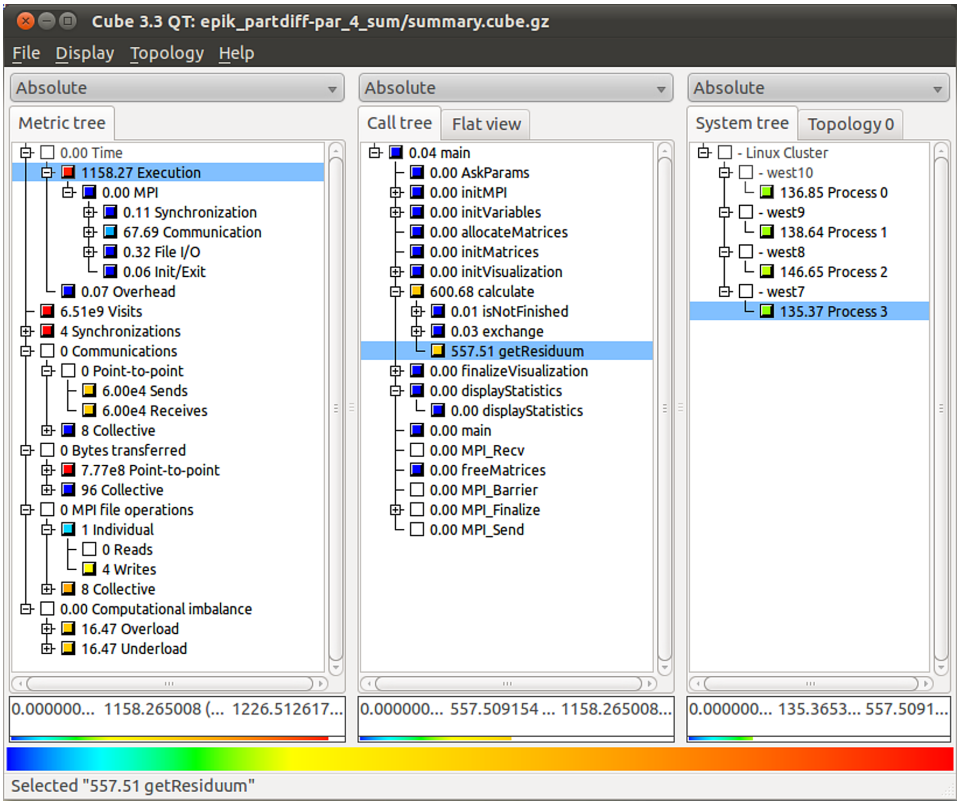


Figure 2.19.: Scalasca's Cube3 browser – identifying the computational overhead in `getResiduum()`.

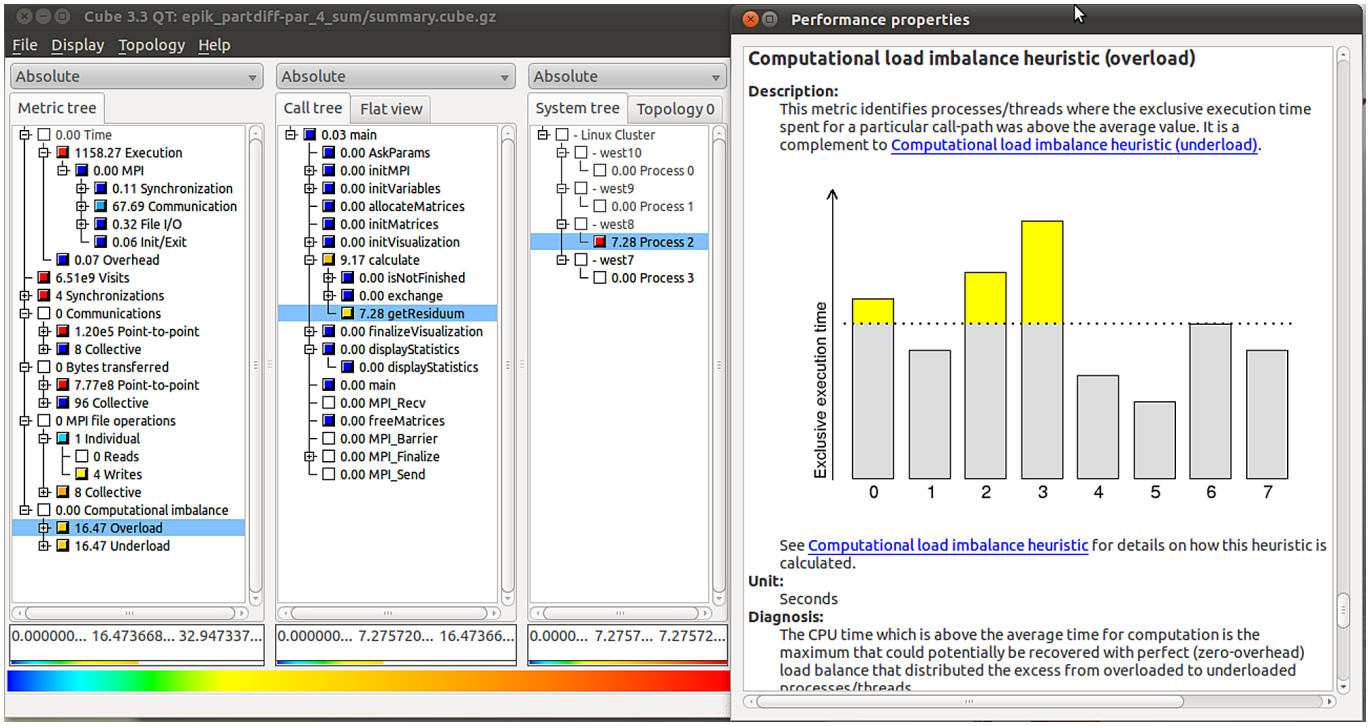


Figure 2.20.: Scalasca's Cube3 – assessing load imbalance and the online help for this metric.

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.

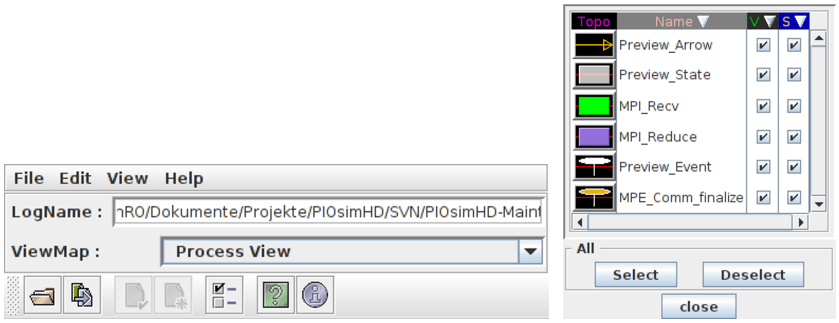
The trace visualizer of MPE is *Jumpshot*, which reads files in the SLOG2 format. SLOG2 tries to be scalable by storing aggregated information about intervals directly inside the format – further information is provided in [ZLGS99].

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).

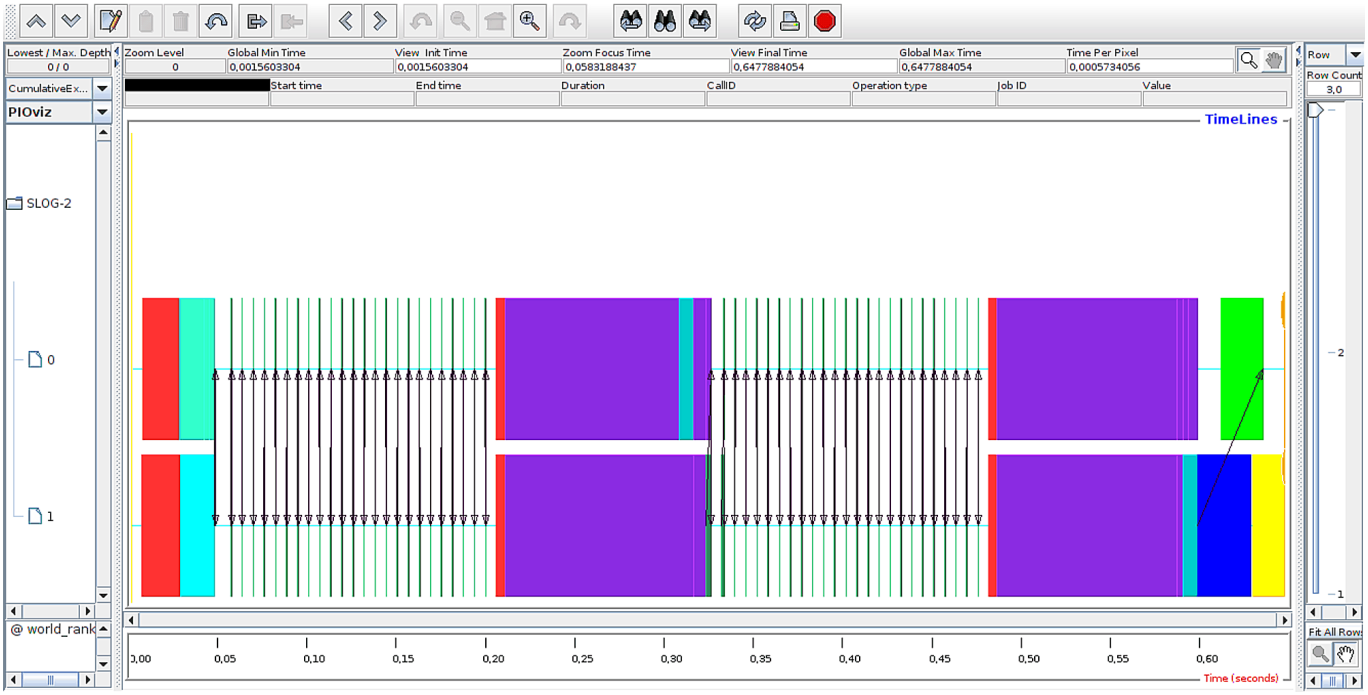
The *timeline window* shows the activity of each process over time. Processes are enumerated in a tree view and mapped to the timelines according to a *ViewMap*. A screenshot of the timeline window is given in Figure 2.21c. On the left side, the tree view shows two processes, the activity is drawn in 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()`.

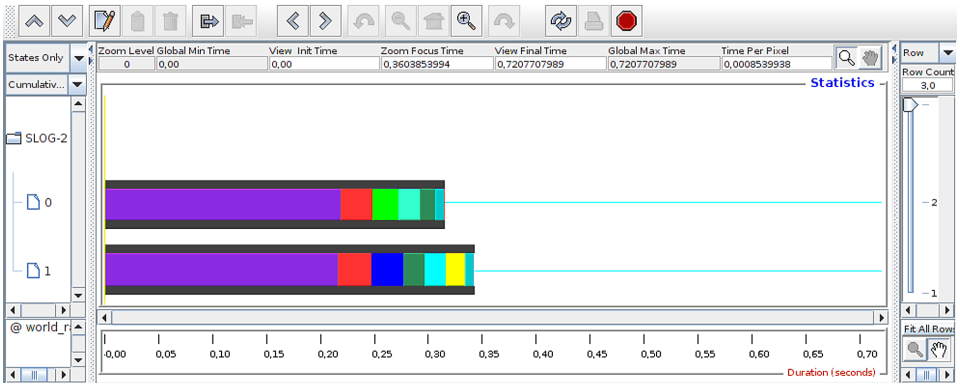
Both windows offer functionality to zoom and scroll in the window, this is provided by the icons in the toolbar. Timelines can be enlarged with the slider on the right of the windows. Additionally, individual timelines can be deleted or moved around; to move a timeline, it must be cut by the user and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



(a) Main window (b) Legend window



(c) Timeline window



(d) Profile window

Figure 2.21.: Jumpshot windows.

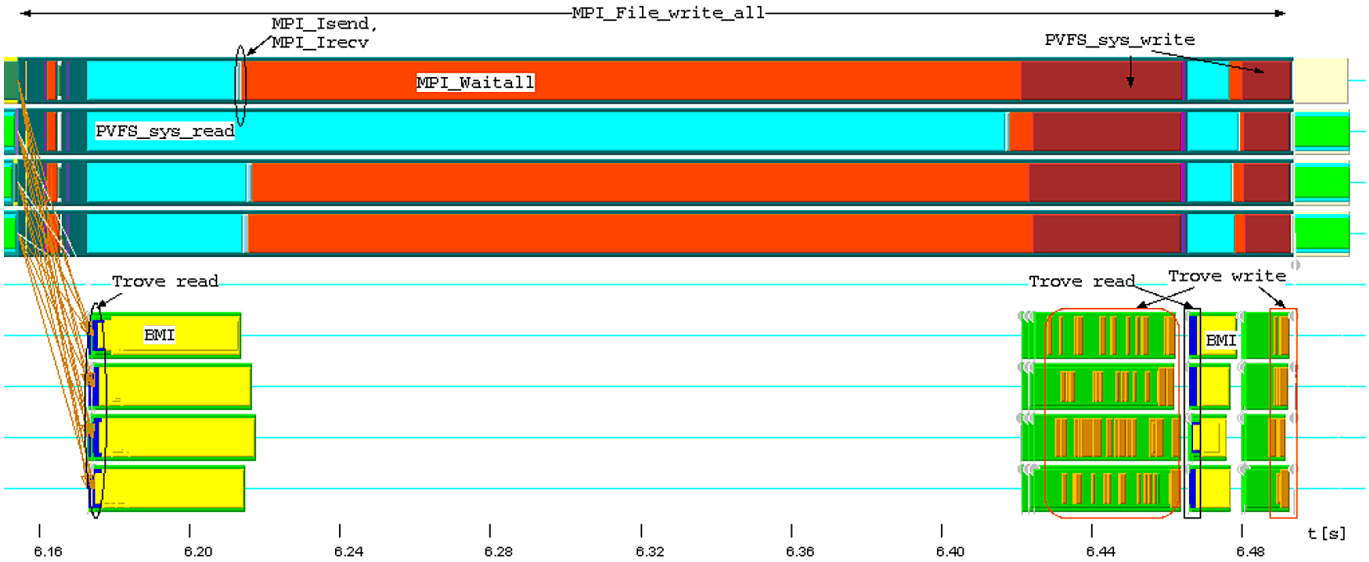


Figure 2.22.: Modified screenshot of PIOviz visualizing the interaction between 4 clients and 4 servers with explanations [KTML09].

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].

processes.

- *Partitioning of trace data* into sections of related information. This fosters the previous points by enabling independent and probably faster generation and analysis of those sections. In general, a section should be independent of others but it might be related to activities recorded in another section.
- *Stability* of the interface and the format. An adjustment of recorded events and their related information or an extension of the basic trace format should be possible without rewriting the complete tracing and analysis toolchain.
- *Flexibility* in terms of the stated requirements, that means gearing the format towards any specific purpose should be possible.

By itself, each requirement can be realized with simple approaches. However, several requirements are conflicting with each other, for instance recording of all potentially relevant information results in large files, which must be analyzed later.

When work on this thesis has been started, no trace format existed that fulfilled all requirements for the simulator, therefore, a new trace format has been developed (the format is described in Section 4.2). In the following, the concepts behind OTF are introduced, to foster understanding of restrictions imposed by trace formats and their interfaces.

Open Trace Format

OTF [KBB+06, KBB09] is developed in cooperation between the TAU tool group of the *University of Oregon* and *The Center for Information Services and High Performance Computing* (ZIH) of the TU-Dresden. Initially, OTF is designed to record trace information for MPI and OpenMP parallel applications. The API and implementation and contained tools are licensed under a BSD open source license. First, a few basic concepts of OTF are discussed:

- Traces of processes and threads are split into *streams* (consider them as independent sections), which are mapped into a set of files, i.e., each file contains the trace information from one or multiple streams. A *master file* contains information about the mapping from processes to streams.
- Within a trace file, an event is recorded as a single text line, the detailed structure of a record is defined separately. A new type can be defined for a stream in a *local definition*, or it can be stored globally. Definitions are stored in additional files. For some functions, such as MPI functionality, the types are predefined. To reduce the amount of redundant data, several common properties, like timestamps and information about threads, are written independently of event information, they are valid for all subsequent events until they are reassigned.
- Files are encoded in ASCII to ensure platform independence and readability by third-party tools. Random access is realized by performing binary search over the timestamp. To decrease the file size blocks of data can be transparently compressed with *zlib*[56].
- Auxiliary information, locally defined records, snapshot records and statistics can be optionally added. Since information about the state of the program is stored implicitly in a stream, random access is difficult. For example, the call graph is recorded by issuing several nested start events. When a file is accessed at a random position, then the earlier events are skipped and should not be read but they are essential to understand the current state of the process. Thus, in OTF, derived information can be added in additional files, a *snapshot* holds the current state. That means the snapshot keeps all variable information about the nested function invocation, or I/O activities. *Statistics* store summary information from beginning up to the current time, arbitrary intervals can be derived from these values by subtracting start from end statistics. Thus, information before the snapshot is not needed and random access is possible.

[56] http://zlib.net/

- A low-level and a high-level C-interface are provided to access OTF files.
- Several tools are provided to access and manipulate trace files: to filter processes, to compress or decompress traces, to dump tracefiles as text, to add statistics or snapshots, or to change the number of streams per file.

API The low-level API is briefly introduced based on the source code of OTF and the included documentation (version 1.7.0rc1 as of 2010-03-30).

Important OTF functions and an excerpt of the API documentation are provided in Section A.1. Exemplary usage of the API to write a trace file is shown in Listing 2.7. In the example, a single function entry and exit for the main function of one process are written.

Listing 2.7: OTF example code (as provided by the API documentation)

```
1  #include <assert.h>
2  #include "otf.h"
3
4  int main( int argc, char** argv ) {
5  // Declare a file manager and a writer.
6          OTF_FileManager* manager;
7          OTF_Writer* writer;
8  // Initialize the file manager. Open at most 100 OS files.
9          manager= OTF_FileManager_open( 100 );
10         assert( manager );
11 // Initialize the writer. Open file "test", writing one stream.
12       writer = OTF_Writer_open( "test", 1, manager );
13       assert( writer );
14 // Write some important Definition Records.
15 // Have a look at the specific functions to see what the parameters mean.
16       OTF_Writer_writeDefTimerResolution( writer, 0, 1000 );
17       OTF_Writer_writeDefProcess( writer, 0, 1, "proc one", 0 );
18       OTF_Writer_writeDefFunctionGroup( writer, 0, 1000, "all functions" );
19       OTF_Writer_writeDefFunction( writer, 0, 1, "main", 1000, 0 );
20 // Write an enter and a leave record. time = 10000, 20000
21 // process = 1 function = 1
22 // Sourcecode location doesn't matter, so it's zero.
23       OTF_Writer_writeEnter( writer, 10000, 1, 1, 0 );
24       OTF_Writer_writeLeave( writer, 20000, 1, 1, 0 );
25 // Clean up before exiting the program.
26       OTF_Writer_close( writer );
27       OTF_FileManager_close( manager );
28       return 0;
29 }
```

The API does not support mapping of processes to hardware topologies. However, a potential mapping could be recorded by forming process groups, or by storing comments according to the OTF developers. Relations between two processes are created explicitly by invoking the functions `OTF_Writer_write[Recv|Send]Msg()`, or for a collective MPI call via `OTF_Writer_writeBeginCollectiveOperation()`.

OTF is under constant development but the developers worry of modifying the existing interface or file format because that could break comparability. Support to add arbitrary data to an event was added recently, in new versions key-value pairs can be added to a call [KGS+10]. An example with these extensions is provided in listing 2.8. First the key and the datatype of the value pair are defined by using `OTF_Writer_writeDefKeyValue()` (Line 22), then a value must be set (Line 28). To additionally store the defined key/value list to the trace records, the API function with the suffix *KV* can be invoked (Line 30). Those new functions have been added for most previously existing API functions. Prior to this approach,

it was cumbersome to extend existing records, eventually comments could be used, but that was tedious and inefficient.

Listing 2.8: Adding arbitrary information with key/value pairs (as provided in the API documentation)

```
#include <assert.h>
#include "otf.h"

int main( int argc, char** argv ) {

  OTF_FileManager* manager;
  OTF_Writer* writer;
  OTF_KeyValueList* KeyValueList;

  manager= OTF_FileManager_open( 100 );
  assert( manager );

  writer = OTF_Writer_open( "mytrace", 1, manager );
  assert( writer );

// Initialize the prior declared OTF_KeyValueList.

  KeyValueList = OTF_KeyValueList_new();

// Write a DefKeyValue record that assigns key=1 to name="first_arg" with
  →description="first argument of function" and type=OTF_INT32.

  OTF_Writer_writeDefKeyValue( writer, 0, 1, OTF_INT32, "first_arg", "first␣
    →argument␣of␣function" );

// Append a signed integer for key=1 to the initialized KeyValueList.

  OTF_KeyValueList_appendInt32( KeyValueList, 1, 25);

// Write the entries of the KeyValueList together with the enter record.
  →Afterwards the KeyValueList will be empty!

  OTF_Writer_writeEnterKV( writer, 10000, 100, 1, 0, KeyValueList );

// Clean up before exiting the program. Close the OTF_KeyValueList.

  OTF_KeyValueList_close( KeyValueList );
  OTF_Writer_close( writer );
  OTF_FileManager_close( manager );

  return 0;
}
```

In the Birds of a Feather [KWGS10], the future direction of (OTF) tools has been discussed. OTF evolves into a new (OTF2) format with funding from the BMBF and DOE. A unified performance measurement infrastructure is envisioned in form of the Score-P[KO11] measurement environment, which handles creation of event traces in OTF2 and call graph profiles for Cube-4. Similar approaches to unify the infrastructure have been tried in the past, OMIS [LWSB97] is an example for an earlier interface, which also supported debuggers and load management systems.