D2.6.2 – Best practice in performance analysis and optimisation

WP2: Underpinning and cross-cutting technologies

Project Acronym	CRESTA
Project Title	Collaborative Research Into Exascale Systemware, Tools and Applications
Project Number	287703
Instrument	Collaborative project
Thematic Priority	ICT-2011.9.13 Exascale computing, software and simulation

Due date:	M12
Submission date:	30/09/2012
Project start date:	01/10/2011
Project duration:	36 months
Deliverable lead organization	TUD
Version:	1.0
Status	Final
Author(s):	Jens Doleschal (TUD)
Reviewer(s)	Ben Hall (UCL), Adam Carter (UEDIN)

Dissemination level	
<PU/PP/RE/CO>	*PU - Public*

© CRESTA Consortium Partners 2011

Version History

Version	Date	Comments, Changes, Status	Authors, contributors, reviewers
0.1	24/08/2012	First full version of the deliverable	Jens Doleschal (TUD)
0.1	03/09/2012	Review	Ben Hall (UCL)
0.1	13/09/2012	Review	Adam Carter (EPCC)
0.1.1	17/09/2012	Incorporate review comments from Adam Carter into document	Jens Doleschal (TUD)
0.1.2	21/09/2012	Incorporate review comments from Ben Hall into document	Jens Doleschal (TUD)
0.2	24/09/2012	Added section “Score-P measurement system” Added subsubsection’s to subsection 4.1 for clarification reasons Final modifications in all sections	Jens Doleschal (TUD)
1.0	26/09/2012	Final version of the deliverable	Jens Doleschal (TUD)

© CRESTA Consortium Partners 2011

Table of Contents

Index of Figures

© CRESTA Consortium Partners 2011

1 Executive Summary

This document describes the best practices in performance analysis and optimisation defined in Task 2.6.2 of WP 2 of the CRESTA project. This document should guide application developers in the process of tuning and optimising their codes for performance. It focuses on application performance optimisation and analysis, and describes which application performance monitoring techniques should be used in which situation, which performance issues may occur, how the issues can be detected, and which tools should be used in which order to accomplish common performance analysis tasks. Furthermore, this document presents the application performance analysis tools of the CRESTA project Score-P and Vampir. Scalasca, one of the profile analysis tools of Score-P, is also presented to provide a complete workflow of performance analysis tools for an application performance analysis. In general, the application performance optimisation and analysis starts with lightweight monitoring of the application, either by a job-monitoring tool or by a coarse-grained sample-based profiling to identify potential problem areas such as communication, memory, or I/O. Afterwards, a more-detailed call-path profiling should be used to identify phases and functions of interest, and also to locate the main performance problems. These functions and regions of interest can be analysed in more detail by using selective event tracing.

This document does not replace the user guides of individual performance analysis tools developed within or outside CRESTA.

Our investigations and lessons into auto-tuning and power optimisation are still on going and will be finished in the further progress of the project. Therefore, they cannot be addressed in the current version of this document, but when they are available they will be added within a later version of this document.

© CRESTA Consortium Partners 2011

2 Introduction

HPC systems are composed of hundreds of thousands of homogeneous or even heterogeneous processing elements. Running applications efficiently in such highly parallel and complex systems requires orchestrating different levels of concurrency (threads, message passing, I/O, etc.). Therefore, it will be necessary to discover performance bottlenecks originating from the increase of complexity of each level of concurrency and to correct them in the application source codes. Furthermore, the observation of performance problems that originate from the use of shared hardware (network, file system, etc.) becomes fundamental since a minority of processes or processing elements can disturb and affect the whole system.

This document should guide application developers in the process of tuning and optimising their codes for performance. It describes which application performance monitoring techniques should be used in which situation, which performance issues may occur, how the issues can be detected, and which tools should be used in which order to accomplish common application performance analysis tasks.

Section 3 gives a brief overview of common performance monitoring techniques and recommendations for their use. After that, section 4 describes typical performance issues and gives hints regarding which monitoring techniques and performance metrics should be used to identify a certain performance issue. Finally, section 5 presents a typical application performance analysis workflow.

2.1 Purpose

This document describes the best practices in performance analysis and optimisation defined in Task 2.6.2 of WP 2 of the CRESTA project and addresses therefore the following topics:

- Application performance monitoring and analysis techniques
- Application performance analysis tools workflow

2.2 Glossary of Acronyms

CRESTA	Collaborative Research Into Exascale Systemware, Tools and Applications
D	Deliverable
MPI	Message Passing Interface
OpenMP	Open Multi-Processing
OTF2	Open Trace Format Version 2
WP	Work Package

© CRESTA Consortium Partners 2011

3 Performance Monitoring Techniques

This section gives a brief overview of various application monitoring and performance data generation techniques and recommendations for the use of these techniques.

3.1 Data Generation Techniques

For the observation of the state and the behaviour of an application over runtime two main approaches basically exist. Information about an application can be generated either by using instrumentation, i.e. inserting pieces of code into the application source code or binary for an event based measurement, or by using a sampling approach, i.e. observing the state of the application frequently, or a combination of both. The selection of the right technique for a given performance issue is always a trade-off between intrusion and the level of detail. While intrusion does not just slow down the process of obtaining the information, it can actually change the application behaviour and ias a result the measurement information will lose its significance. In contrast, the level of detail is a significant factor in determinijng if a performance issue can be detected or not.

The main goal should be to select the best information generation technique for a given application and situation with emphasis on reducing the intrusiveness while providing enough information needed to detect different kinds of performance bottlenecks.

3.1.1 Sampling-based Monitoring

Sampling is a monitoring technique that is used to periodically observe the state of an application, i.e., which function is executed at observation time, without any need to modify the application. The sampling frequency is the important steering factor to control the level of detail and the intrusiveness. A low sampling frequency is ideal to get an overview of the application since the total amount of samples and their size is limited, but the detection of the root cause of a performance problem might become impossible. In contrast, a high sampling frequency will increase the possibility to detect the cause of a performance problem, but may also increase the intrusion significantly. As a result, the performance analyst has to choose the optimal sampling frequency depending on the level of detail, the point of time, and the processing element to be observed.

An ability of sampling is to dynamically change the sampling frequency during measurement to address the trade-off between intrusion and the level of detail. However, steering the sampling frequency without any knowledge about the application is a challenging task.

Recommendations in use:

- Lightweight monitoring of batch system jobs to get job overview information.
- Probe-based monitoring to gain insight into the application at a specific point in time

3.1.2 Event-based Monitoring

In contrast to sampling, event-based monitoring records only information if a specific pre-defined event occurs, e.g. function entry/exit. Therefore, small parts of code have to be inserted into the application, which requires often a rebuild of the application. The following are the most commonly-used ways to generate event information:

- Compiler instrumentation inserts user-defined code snippets at the very beginning and ending of each function;
- Source-to-source instrumentation transforms the original application and inserts code snippets at points/regions of interest;
- Library instrumentation intercepts public functions of an external shared library by using a dlopen interception mechanism;

© CRESTA Consortium Partners 2011

- Binary instrumentation modifies the executable either at runtime or before program execution to insert code snippets at function entries and exits; and
- Manual instrumentation.

The level of detail within event monitoring depends therefore on the events which should be monitored, their occurrence, and also duration. Using event-based monitoring can result in detailed information but in the same way the level of detail increases the intrusion which will become more and more critical, especially when tiny and often-used functions are monitored, e.g., constructors and static class methods. For millions of processing elements over a long monitoring period this monitoring technique can result in huge amounts of information.

Recommendations in use:

- Monitoring of tiny functions or regions with short parts of interest, e.g. OpeMP regions with implicit barriers.
- Selective monitoring of routines of interest, e.g. if the user is only interested in the MPI communication event-based MPI monitoring allows to monitor only these routines.

3.2 Performance Monitoring Techniques

Basically, there are two main approaches to monitor the performance behaviour of parallel applications: profiling and tracing.

3.2.1 Profiling

Profiling aggregates the measurement information and generates statistics for the whole application run or for phases of interest. Flat profiles provide statistical information in a list style with various metrics like inclusive runtime and number of invocations. For a more detailed analysis, in particular to analyse performance in the context of caller-callee relationships, call-path and call-graph profiles are scalable techniques to provide more insight into highly complex and parallel applications. Profiling with its nature of summarization offers an opportunity to be extremely scalable, since the reduction of information can be done during the application runtime. Nevertheless, profiles may lack crucial information about message runtimes and bandwidth, since message matching is usually infeasible during profiling. Therefore, analysis of communication-based performance issues is usually only possible by interpreting the aggregated time spent in the communication routines.

Recommendations in use:

- Profiling should be used to get an overview of the performance behaviour of highly parallel applications.
- Profiling may also help to identify potential performance problems since these issues or their effects are usually included in several performance indicators, e.g., occurrence and runtime information of communication routines can be used to identify potential communication issues. Identification of functions and regions of interest can be used to steer the upcoming monitor runs and to focus only on these parts of interest. For a detailed analysis of functions and parts of interest tracing should be used afterwards.

3.2.2 Event Tracing

Event tracing records each event of a parallel application in detail. Thus, it allows the dynamic interaction between thousands of concurrent processing elements to be captured and it is possible to identify outliers from the regular behaviour. As a result, tracing will produce an enormous amount of data and with this monitoring of long running applications is challenging.

Recommendations in use:

- Detailed performance monitoring of functions and regions of interest.

© CRESTA Consortium Partners 2011

- Monitoring the communication performance and behaviour of highly parallel applications.

3.3 The Score-P measurement system

The Score-P 0 measurement infrastructure (see Figure 1) is a highly scalable and easy-to-use tool suite for profiling, event tracing and online analysis of HPC applications. It is the common successor fro the monitoring systems from Vampir [2], Periscope [3], Scalasca [4], and TAU [5]. Score-P has been created in the projects SILC and PRIMA funded by the German Ministry of Education and Research and the US Department of Energy. It will be extended within CRESTA to address the challenges of exascale performance analysis (see WP 3.1 of the CRESTA project).

Score-P collects event data during the execution of an instrumented application and creates either a profile in the new CUBE4 format, or trace files, using the new parallel Open Trace Format Version 2 (OTF2) [6]. Both operation modes can be used without recompilation of the application. Score-P supports an extensive set of events such as function and library calls, communication events and hardware counters. To collect this information, Score-P supports various instrumentation methods, including instrumentation at source level, at compile/link time and at the resulting binary.

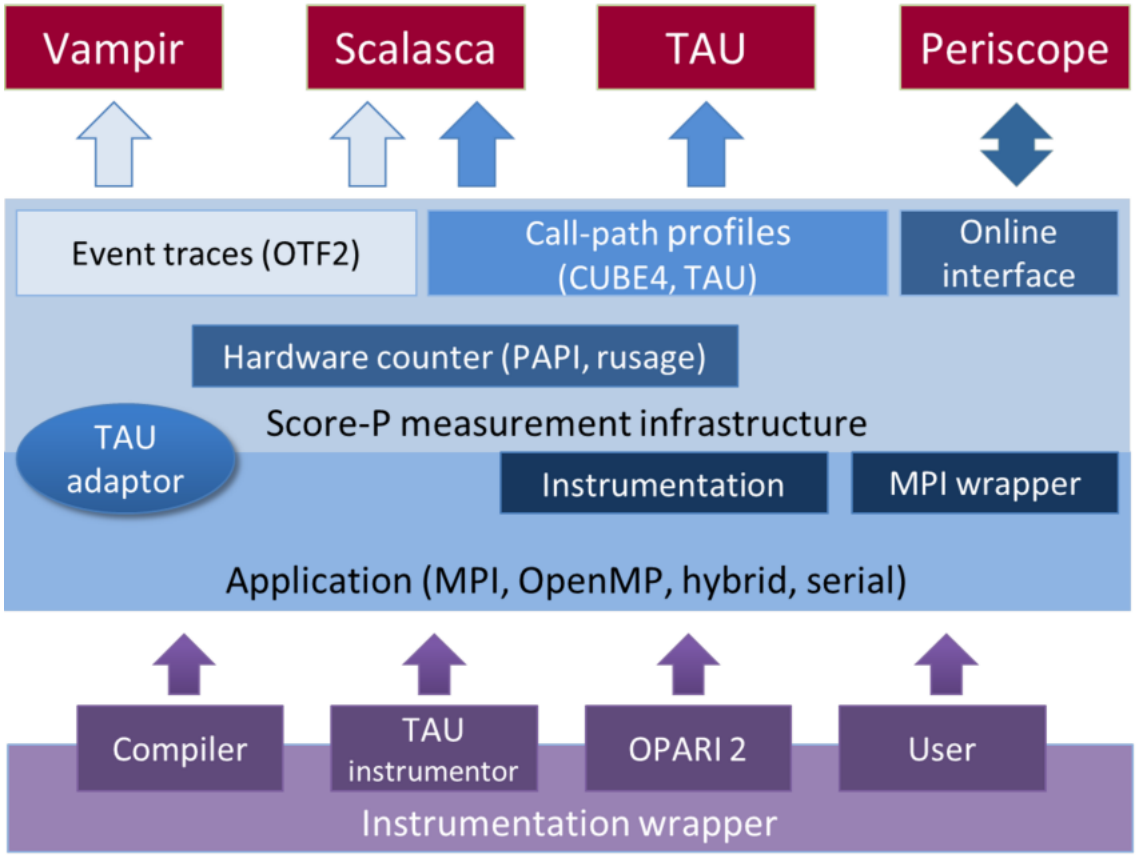

Figure 1: Overview of the Score-P measurement system.

© CRESTA Consortium Partners 2011

4 Performance Issues and Indicators

This section gives a compact overview of potential performance issues that may occur during the application runtime. In addition, indicators, e.g. performance metrics, are provided that may help to decide whether an application suffers from an inherent performance problem or whether application interference may have been at the root of unsatisfactory behaviour. It also should help to get a first assessment regarding the nature of a potential performance problem and help to decide on further analysis steps using the most appropriate monitoring and analysis techniques. In general, it is highly application-dependent whether a metric is too high or low. Therefore, it is not possible to define any fixed thresholds and it is up to the user to interpret the data.

The decision as to whether an application behaves inefficiently or not cannot be easily given. Often it is the sum of a multitude of factors and therefore it is advisable to identify essential parts of the application and to focus on important components of the code, e.g. communication, computation and I/O, first. After that, a hypothesis about potential performance problems can be created and checked. To identify a certain performance problem it is important to use well-suited monitoring techniques in combination with appropriate performance metrics to gain insight into the complex behaviour of the application. Performance issues originating from the interference between applications, e.g. reduced communication performance due to overall network saturation, are often only detectable by involving or comparing the global monitoring information of the entire machine with the application performance information.

4.1 Performance Issues in Communication

In general, communication as opposed to computation does not directly contribute to the calculation of results. Therefore, communication (which basically depends on the type and the number of communication routines, the size of the transferred data, and the communication scheme) should be minimized as much as possible and the fraction of time spent in communication routines like MPI kept low.

> **Performance issue:** The communication dominates the computation or the fraction of time spent in communication routines increases linearly or even worse with the number of processing elements. This usually results in limited scalability of the application.
>
> **Performance metric(s):** Exclusive time of communication routines.
>
> **Performance technique:** To identify this performance issue a profiling technique should be used first. Initially, a call-path profiling, e.g., from Scalasca (see A.1), should be used to identify which communication routines in which calling context are dominating the application runtime. After identification of the main problem(s) a more detailed analysis of the problem should be followed by using an event tracing approach that monitors the communication routines and finally can be analysed for example with Vampir's (see A.2) master timeline and function summary displays.

The cause for an increased communication time can have various reasons. These can vary from load or communication imbalances that result in asymmetric communication behaviour and in increased waiting times to inefficient and non-scalable communication schemes and unsuitable communication routines. Also, the number and size of the data sent within the communication routines influences the communication time significantly. These problems (which are described in more detail in the following subsections) usually prevent scaling to larger processor counts and should therefore be eliminated.

4.1.1 Communication waiting time

Load and communication imbalances typically result in an increased waiting time. Examples for performance issues for point-to-point messages with increased waiting time are the late-sender and the late-receiver problem. For each performance problem

one communication partner (either the sending or the receiving partner) arrives too late in the communication so that the other partner has to wait. An overview of communication performance issues can be found in [7] figure 2. The goal should be to identify load imbalances and to reduce the overall waiting time of the application either by using another communication scheme or an improved load balancing mechanism.

Performance issue: Increased communication time due to load or communication imbalance.

Performance metric(s): Minimum and maximum of the exclusive time spent in communication routines.

Performance technique: A first indicator of load or communication imbalances can be identified with call-path profiling by comparing the minimum and maximum time spent in the several communication routines. Also, the automatic communication wait-state-analysis of Scalasca (see A.1) or tracing the application in combination with Vampir's (see A.2) master timeline can help to identify load imbalances within the application.

4.1.2 Latency-bounded communication

The communication time of applications that rely on a huge number of small messages is influenced significantly by the latency of each message. This will limit the lower bound of the communication and in result also the scalability of the application. It is advisable to reduce the number of used messages and to pack multiple small messages into a larger message, if possible.

Performance issue: The communication is dominated by a large number of small messages; network latency can be a limiting factor of applications scalability.

Performance metric(s): Number of short messages, minimum message transfer time, message data rate.

Performance technique: To identify scalability issues caused by a large number of small messages, a profiling technique, which is able to distinguish the size of messages can be used or a tracing approach that monitors the communication in combination with Vampir's (see A.2) communication matrix and message summary displays is suited to identify this kind of performance issue.

4.1.3 Bandwidth-bounded communication

In contrast, if the majority of messages are large, the limiting factor may be network bandwidth and with this the possibilities to decrease the communication time are limited. Opportunities are the use of other communication schemes or the reduction of the overall message data size.

Performance issue: Increased communication time because the majority of messages are large and network bandwidth is a limiting factor.

Performance metric(s): message data rate, message data volume.

Performance technique: Communication performance issues of the application caused by the limited bandwidth of the network can be identified when the user knows the theoretical bandwidth of the network and uses a profiling technique that is able to provide information about the minimum, average, and maximum message data rate of a communication function in combination with the several message data size. Event tracing is also able to monitor the communication routines and Vampir provides the message data rates of different message size within its communication and message summary display.

© CRESTA Consortium Partners 2011

4.1.4 Unnecessary synchronisation

A further performance issue that increases the communication time and decreases the scalability is the use of unnecessary consecutive synchronisation routines like barriers. The goal should be to identify unnecessary communication routines and to remove them.

> **Performance issue:** The application spent a lot of time in unnecessary synchronisation routines, e.g. barriers.
>
> **Performance metric(s):** Number of synchronisation calls, exclusive time spent in synchronisation routines.
>
> **Performance technique:** The occurrence of synchronisation routines can be identified initially with a call-path profiling from Scalasca (see A.1) and can be analysed in more detail with event tracing in combination with Vampir's (see A.2) master timeline and function summary displays.

4.1.5 Job interference

Finally, low communication performance may also be caused by application interference when multiple jobs that run simultaneously compete for the network. Investigating global monitoring information of the underlying network and comparing with the communication performance of the application can verify this. Some measurement infrastructures like Score-P, the monitoring system of Scalasca, Vampir, TAU and Periscope, allow for the inclusion of global monitoring information of the entire machine state and as a result this information can be visualized within the counter displays of Vampir (see A.2).

4.2 Performance Issues in Computation

Efficient usage of today's highly complex multi- and many-core processors is a key component for efficient and highly parallel applications. On one side the application should utilize the hardware efficiently, i.e. the pipelines of the hardware units should be almost always busy. On the other side highly complex memory hierarchies have to be considered. Increasing the efficiency of memory usage, i.e. by decreasing the number of memory data misses like cache misses, and providing enough instructions per data location, e.g. by avoiding sparse loops, can reduce performance issues in computation.

Analysing the efficiency of hardware and memory usage can be done by almost every performance monitoring technique that allows monitoring of hardware performance counters. The techniques only differ in granularity of information. Most of the measurement tools use PAPI to request the hardware information. As a result this hardware performance information can tell how well the hardware and memory infrastructure of the underlying machine are utilized.

> **Performance issue:** Too many cycles per instruction more than the theoretical minimum can be caused be pipeline hazards or by the memory access latency.
>
> **Performance metric(s):** Number of instructions, Cycles per instruction.

> **Performance issue:** Inefficient usage of the memory hierarchy due to low locality.
>
> **Performance metric(s):** L1/L2/L3 hit and miss rates, Number of instructions.

> **Performance issue:** Increased computing time due to low floating-point performance.
>
> **Performance metric(s):** Floating-point operations per second, Floating-point instructions.

In addition to the previously mentioned hardware utilization and memory access issues, an unbalanced computation caused, for example, by serial parts of the computation or

processes/threads that take longer to compute their parts (“single late comer”), results in idle processes and threads that decrease in the end the parallel efficiency of the computational part and may also affect the waiting time in communication routines.

4.3 Performance Issues in I/O

In general, I/O performance of an application highly dependents on the current load of the I/O subsystem and may change significantly between runs. This means thatdiagnosing an I/O bottleneck usually requires multiple runs and may affect also the tuning results. Typical I/O performance issues are I/O bandwidth bounded computation parts, slow I/O operations, sequential I/O on a single process, which mostly results in idle time for all other concurrent processing elements, and last but not least I/O load imbalance may affect the parallel efficiency of an application. Scalasca (see A.1) may help identify expensive I/O calls, while Vampir (see A.2) can be used to analyse I/O patterns and their performance in more detail.

Global monitoring with load information of the file system can help to decide whether an application was disturbed by other applications or not. In the near future, it will be possible to add this information as external data within an application OTF2 trace file and finally it can be analysed with Vampir’s (see A.2) counter timeline and performance radar displays.

© CRESTA Consortium Partners 2011

5 Performance Analysis Workflow

In general, the performance analysis workflow starts with lightweight monitoring of the application, either by a job-monitoring tool or by a coarse-grained sample-based profiling to identify potential problem areas such as communication, memory or I/O. Afterwards, a more-detailed call-path profiling for example with Scalasca (see A.1) should be used to identify phases and functions of interest, and also to locate the main performance problems. These functions and regions of interest can be analysed in a more detail by using selective event tracing, where only events of interest are instrumented, and in-depth performance analysis with Vampir (see A.2). To distinguish between system and application-related problems, Vampir is also able to display system-level-related information and to correlate it with the application information.

A typical performance analysis workflow successively focuses the analysis to selective parts of the application and increases the level of detail at which performance data are collected. This helps to keep intrusion low and limit the data to be stored needed to identify a performance issue.

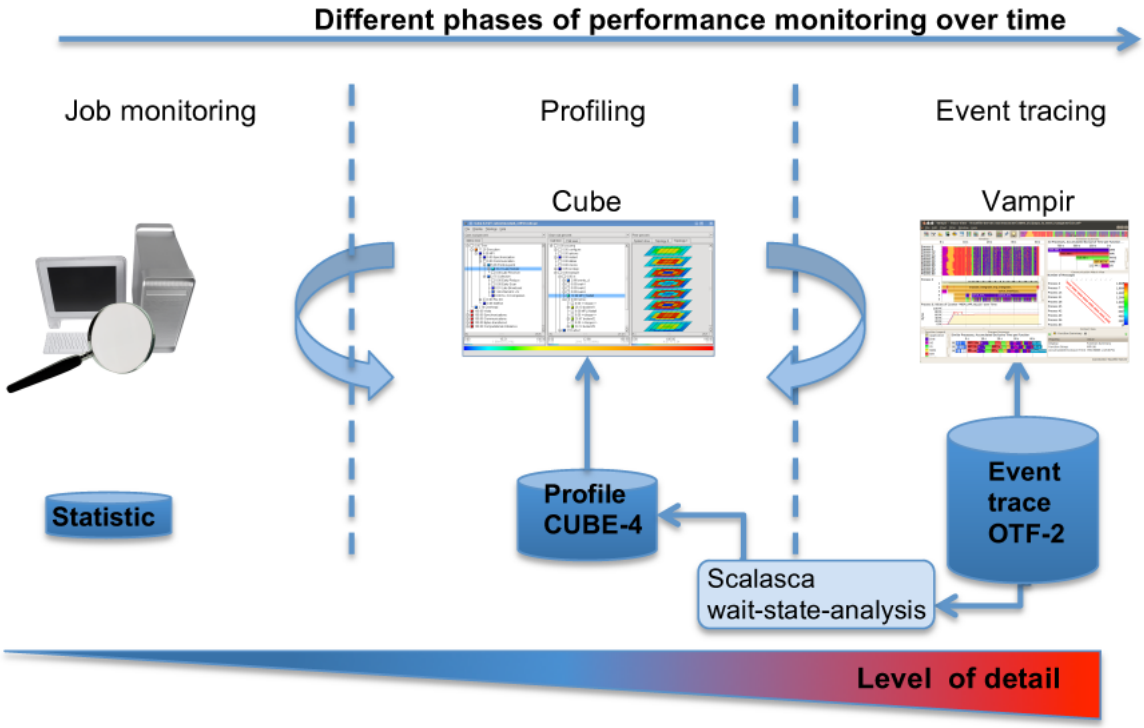

Figure 2: Performance analysis workflow over time with different monitoring and analysis techniques.

© CRESTA Consortium Partners 2011

6 References

[1] D. an Mey, S. Biersdorff, C. Bischof, K. Diethelm, D. Eschweiler, M. Gerndt, A. Knüpfer, D. Lorenz, A. D. Malony, W. E. Nagel, Y. Oleynik, C. Rössel, P. Saviankou, D. Schmidl, S.S. Shende, M. Wagner, B. Wesarg, and F. Wolf: Score-P: A Unified Performance Measurement System for Petascale Applications. In Proc. of the CiHPC: Competence in High Performance Computing, HPC Status Konferenz der Gauß-Allianz e.V., Schwetzingen, Germany, June 2010, pages 85–97. Gauß-Allianz, Springer, 2012.

[2] W. Nagel, M. Weber, H.-C. Hoppe, and K. Solchenbach. VAMPIR: Visualization and Analysis of MPI Resources. Supercomputer, 12(1):69–80, 1996.

[3] M. Gerndt and M. Ott. Automatic Performance Analysis with Periscope. Concurrency and Computation: Practice and Experience, 22(6):736–748, 2010.

[4] M. Geimer, F. Wolf, B.J.N. Wylie, E. Ábrahám, D. Becker, B. Mohr: The Scalasca performance toolset architecture. Concurrency and Computation: Practice and Experience, 22(6):702-719, April 2010.

[5] S. Shende and A. D. Malony. The TAU Parallel Performance System. International Journal of High Performance Computing Applications, 20(2):287–331, 2006. SAGE Publications.

[6] D. Eschweiler, M. Wagner, M. Geimer, A. Knüpfer, W. E. Nagel, and F. Wolf: Open Trace Format 2 - The Next Generation of Scalable Trace Formats and Support Libraries. In Proc. of the Intl. Conference on Parallel Computing (ParCo), Ghent, Belgium, 2011, volume 22 of Advances in Parallel Computing, pages 481–490, IOS Press, 2012.

[7] B. Mohr, A. Kühnal, M. Hermanns, and F. Wolf: Performance Analysis of One-sided Communication Mechanisms. In *Proc. of the Conference on Parallel Computing (ParCo), Malaga, Spain*, September 2005, Minisymposium Performance Analysis.

© CRESTA Consortium Partners 2011

Annex A. Description of Tools

A.1 Scalasca

Scalasca is a free software tool that supports the performance optimisation of parallel programs by measuring and analysing their runtime behaviour. The tool has been specifically designed for use on large-scale systems including IBM Blue Gene and Cray XE, but is also well suited for small- and medium-scale HPC platforms. The analysis identifies potential performance bottlenecks, in particular those concerning communication and synchronization. The user of Scalasca can choose between two different analysis modes: (i) performance overview on the call-path level via profiling and (ii) the analysis of wait-state formation via event tracing. Wait states often occur in the wake of load imbalance and are serious obstacles to achieving satisfactory performance. Performance-analysis results are presented to the user in an interactive explorer called Cube (Figure 3) that allows the investigation of the performance behaviour on different levels of granularity along the dimensions performance problem, call path, and process. The software has been installed at numerous sites in the world and has been successfully used to optimise academic and industrial simulation codes.

Typical questions Scalasca helps to answer

- Which call-paths in my program consume most of the time?
- Why is the time spent in communication or synchronisation higher than expected?
- Does my program suffer from load imbalance and why?

Supported programming models

Scalasca supports applications based on the programming interfaces MPI and OpenMP, including hybrid applications based on a combination of the two.

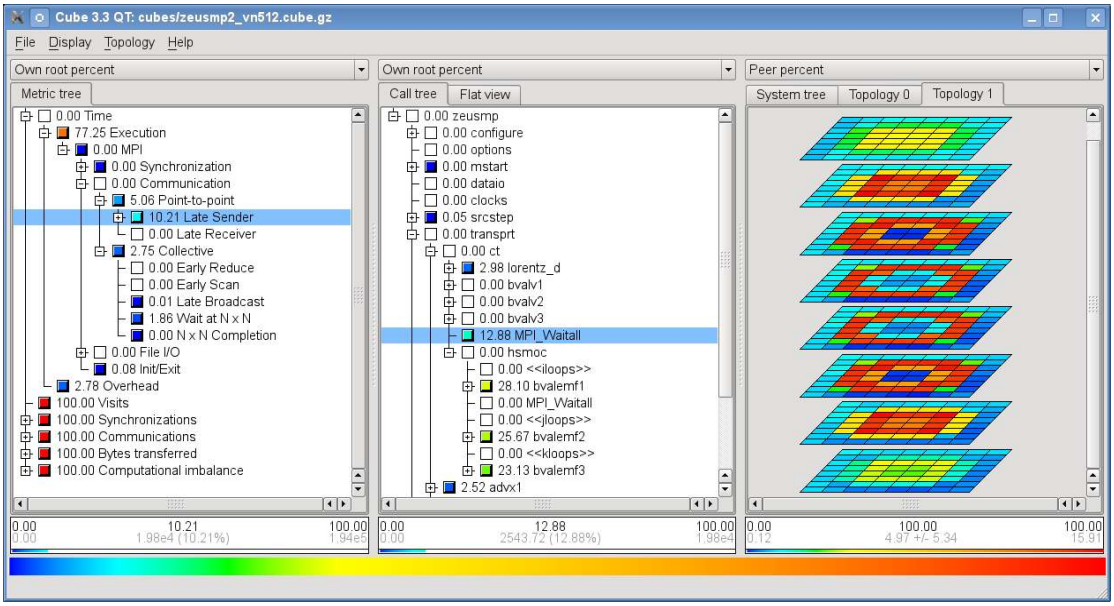

Figure 3: Interactive exploration of performance behaviour in Scalasca along the dimensions performance metric (left), call tree (middle), and process topology (right).

Input sources

The analyses offered by Scalasca rest on profiles in the CUBE-4 format and event traces in the OTF-2 format. Both performance data formats can be generated using Score-P.

Performance analyses

Summary profile: The summary profile can be used to identify the most resource-intensive call paths or processes. It tells how the execution time and other performance

metrics including hardware counters are distributed across the call tree and the set of processes or threads.

Time-series profile: The time-series profile can be used to analyse how the performance behaviour evolves over time – even if the application runs for a longer period. Essentially, a time-series profile provides a separate summary profile for every iteration of the main loop.

Wait state analysis: This analysis extracts from event traces the location of wait states. Detected instances are both classified and quantified. High amounts of wait states usually indicate load or communication imbalance.

Delay analysis: The delay analysis extends the wait-state analysis in that it identifies the root causes of wait states. It traces wait states back to the call paths causing them and determines the amount of waiting time a particular call path is responsible for. It considers both direct wait states and those created via propagation.

Critical-path analysis: This trace-based analysis determines the effect of imbalance on program runtime. It calculates a set of compact performance indicators that allow users to evaluate load balance, identify performance bottlenecks, and determine the performance impact of load imbalance at first glance. The analysis is applicable to both SPMD and MPMD-style programs.

Instrumentation

User code is instrumented in source code (automatically by compiler or PDT instrumentor, or manually with macros or pragmas). OpenMP constructs are instrumented in source code (automatically by the OPARI2 instrumentation tool). MPI calls are intercepted automatically through library interposition.

License model

The software is available under the New BSD license.

Further documentation

- Website: www.scalasca.org
- Support email: scalasca@fz-juelich.de
- Quick reference guide: installation directory under $SCALASCA_ROOT/doc/manuals/QuickReference.pdf
- Scalasca user guide: installation directory under $SCALASCA_ROOT/doc/manuals/UserGuide.pdf
- CUBE user guide: installation directory under $CUBE_ROOT/doc/manuals/cube3.pdf

A.2 Vampir

Vampir is a graphical analysis framework that provides a large set of different chart representations of event-based performance data. These graphical displays, including timelines and statistics, can be used by developers to obtain a better understanding of their parallel program's inner working and to subsequently optimise it. See Figure 4 for a color-coded visualisation of a parallel application with the Vampir GUI.

Vampir is designed to be an intuitive tool, with a GUI that enables developers to quickly display program behaviour at any level of detail. Different timeline displays show application activities and communication along a time axis, which can be zoomed and scrolled. Statistical displays provide quantitative results for the currently selected time interval. Powerful zooming and scrolling along the timeline and process/thread axis allows pinpointing the causes of performance problems. All displays have context-sensitive menus, which provide additional information and customisation options. Extensive filtering capabilities for processes, functions, messages or collective operations help to narrow down the information to the interesting spots. Vampir is based on Qt and is available for all major workstation operation systems as well as on most parallel production systems. The parallel version of Vampir, VampirServer, provides fast interactive analysis of ultra large data volumes.

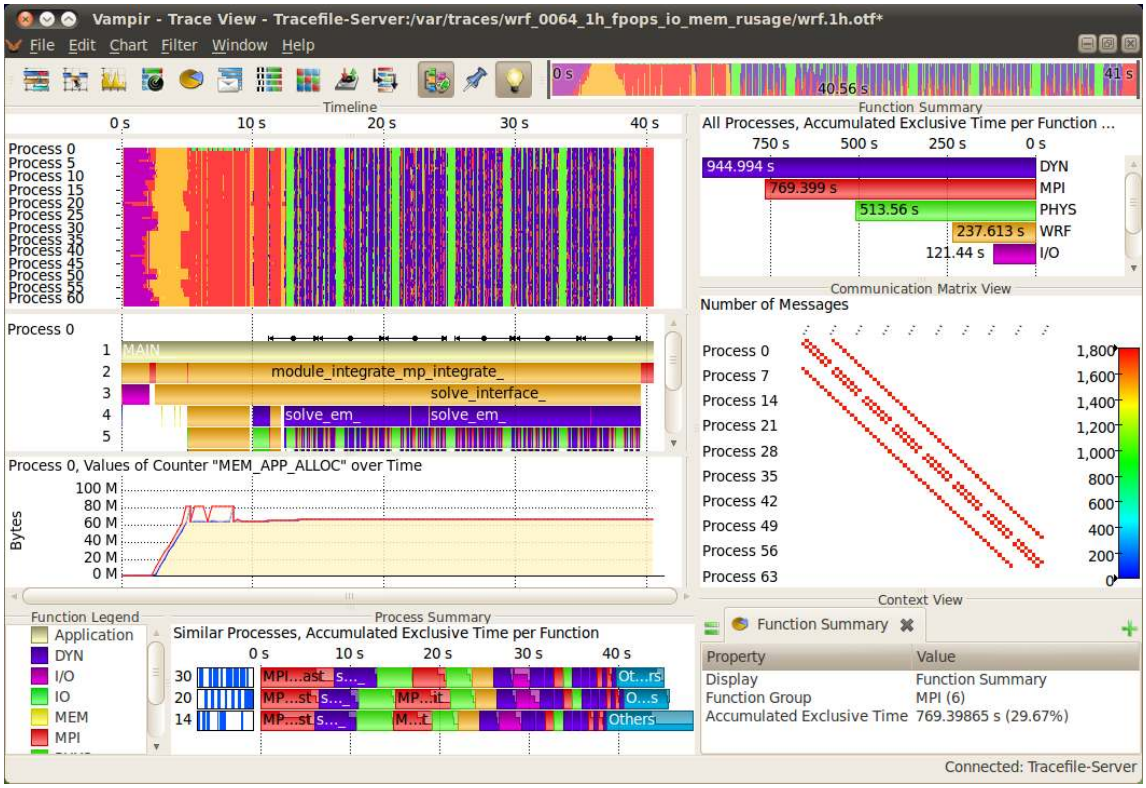

Figure 4: Color-coded visualisation of a parallel application run with timeline and statistic displays of the Vampir GUI.

Typical questions Vampir helps to answer

- What happens in my application execution during a given time in a given process or thread?
- How do the communication patterns of my application execute on a real system?
- Are there any imbalances in computation, I/O or memory usage and how do they affect the parallel execution of my application?

Supported programming models

Vampir supports applications based on the programming interfaces MPI and OpenMP, including hybrid applications based on a combination of the two. Furthermore Vampir also analyses hardware-accelerated applications using CUDA and/or OpenCL.

Input sources

The analyses offered by Vampir rest on event traces in the OTF/OTF-2 format generated by the runtime measurement system VampirTrace/Score-P.

Performance analysis via timeline displays

The timeline displays show the sequence of recorded events on a horizontal time axis that can be zoomed to any level of detail. They allow an in-depth analysis of the dynamic behaviour of an application. There are several types of timeline displays.

- Master timeline: This display shows the processes of the parallel program on the vertical axis. Point-to-point messages, global communication, as well as I/O operations are displayed as arrows. This allows for a very detailed analysis of the parallel program flow including communication patterns, load imbalances, and I/O bottlenecks.
- Process timeline: This display focuses on a single process only. Here, the vertical axis shows the sequence of events on their respective call-stack levels, allowing a detailed analysis of function calls.

© CRESTA Consortium Partners 2011

- Counter data timeline: This chart displays selected performance counters for processes aligned to the master timeline or the process timelines. This is useful to locate anomalies indicating performance problems.
- Performance radar timeline: This chart displays selected performance counters overall processes of the parallel program over time. This is useful to locate differences in the performance behaviour between the processes.

Performance analysis via statistical displays

The statistical displays are provided in addition to the timeline displays. They show summarised information according to the currently selected time interval in the timeline displays. This is the most interesting advantage over pure profiling data because it allows specific statistics to be shown for selected parts of an application, e.g., initialisation or finalisation, or individual iterations without initialisation and finalisation. Different statistical displays provide information about various program aspects, such as execution times of functions or groups, the function call tree, point-to-point messages, as well as I/O events.

Instrumentation

Application code can be instrumented by the compiler or with source-code modification (automatically by the PDT instrumentor, or manually using the VampirTrace/Score-P user API). OpenMP constructs can be instrumented by the OPARI tool using automatic source-to-source instrumentation. MPI calls are intercepted automatically through library interposition.

License model

Vampir is a commercial product distributed by GWT-TUD GmbH. For evaluation, a free demo version is available on the website.

Further documentation

- Website: www.vampir.eu
- Support email: service@vampir.eu
- Vampir manual: installation directory under $VAMPIR_ROOT/doc/vampir-manual.pdf

© CRESTA Consortium Partners 2011