D5.8 Integrated Hybrid Scalasca Analysis
Version 1.0

Document Information

<table>
<thead>
<tr>
<th>Contract Number</th>
<th>610402</th>
</tr>
</thead>
<tbody>
<tr>
<td>Project Website</td>
<td><a href="http://www.montblanc-project.eu">www.montblanc-project.eu</a></td>
</tr>
<tr>
<td>Contractual Deadline</td>
<td>PM24</td>
</tr>
<tr>
<td>Dissemination Level</td>
<td>PU</td>
</tr>
<tr>
<td>Author</td>
<td>Marc Schlüter (JUELICH)</td>
</tr>
<tr>
<td>Contributors</td>
<td>Markus Geimer, Ilya Zhukov (JUELICH)</td>
</tr>
<tr>
<td>Reviewer</td>
<td>Harald Servat (BSC), Brice Videau (CNRS)</td>
</tr>
<tr>
<td>Keywords</td>
<td>Performance Analysis, Profiling, Tracing</td>
</tr>
</tbody>
</table>

**Notices:**
The research leading to these results has received funding from the European Community’s Seventh Framework Programme [FP7/2007-2013] under grant agreement n° 610402.

© Mont-Blanc 2 Consortium Partners. All rights reserved.
## Change Log

<table>
<thead>
<tr>
<th>Version</th>
<th>Description of Change</th>
</tr>
</thead>
<tbody>
<tr>
<td>V0.1</td>
<td>Initial draft</td>
</tr>
<tr>
<td>V0.2</td>
<td>Full version (to Mont-Blanc reviewers)</td>
</tr>
<tr>
<td>V1.0</td>
<td>Incorporated feedback from H. Servat &amp; B. Videau</td>
</tr>
</tbody>
</table>
# Table of Contents

Executive Summary ................................................................................................................. 4

1 Introduction ......................................................................................................................... 5

2 Software components ........................................................................................................ 5
   2.1 Score-P instrumentation and measurement system ....................................................... 5
   2.2 Scalasca Tracing Tools ................................................................................................. 6

3 Integrated hybrid analysis .................................................................................................. 8
   3.1 Generic support for task-based programming models .................................................. 8
      3.1.1 Score-P measurement and profiling ............................................................... 9
      3.1.2 Scalasca trace analysis ............................................................................... 9
   3.2 Integrated result presentation of hybrid asynchronous applications ....................... 10
      3.2.1 Analysis report post-processing ................................................................. 10
      3.2.2 Call tree visualization ........................................................................ 12
   3.3 Support for additional programming models ......................................................... 13
      3.3.1 OpenCL ........................................................................................................ 13
      3.3.2 OmpSs ....................................................................................................... 14
      3.3.3 Programming model support developed outside of Mont-Blanc ....................... 17

4 Conclusion and future work ............................................................................................... 19

Acronyms and Abbreviations ............................................................................................... 20

References ............................................................................................................................ 21
Executive Summary

To make effective use of current high-performance computing (HPC) architectures, developers of scientific simulation codes often have to use multiple parallel programming models in combination. To assist application developers in optimizing such hybrid codes, it is important to provide them with powerful performance analysis tools that are capable of dealing with multiple parallel programming models concurrently.

In this deliverable, we describe our modifications and enhancements to the Score-P instrumentation and measurement infrastructure as well as the Scalasca Tracing Tools package implemented within the Mont-Blanc project towards an integrated analysis of hybrid applications using multiple parallel programming models in combination. In particular, we focus on the support for the OmpSs and OpenCL programming models as well as the challenges introduced by the asynchronous nature of create/wait-type threading and task-based programming. Various examples highlight that Score-P and Scalasca now effectively support the performance analysis of hybrid codes using a single, coherent workflow and a unified result presentation.
1 Introduction

To make effective use of current high-performance computing (HPC) architectures, developers of scientific simulation codes often have to use multiple parallel programming models in combination. For example, simulation codes may use the Message Passing Interface (MPI) for inter-node communication, OpenMP for parallelizing work across multiple cores within a node, and CUDA or OpenCL to leverage accelerators like GPGPUs or Intel Xeon Phi many-core co-processors attached to the node. To assist application developers in optimizing such hybrid codes, it is important to provide them with performance analysis tools that are capable of dealing with multiple parallel programming models concurrently.

In this deliverable, we describe the work done to accomplish this goal in the Score-P instrumentation and measurement system as well as the Scalasca trace analysis package. The document is structured as follows: Section 2 provides a brief overview of the Score-P and Scalasca packages. Next, Section 3 describes our modifications and enhancements in both components towards an integrated analysis. Finally, Section 4 concludes the deliverable and outlines future work.

2 Software components

This section provides a brief overview of the two software components that have been modified in the context of this deliverable and how they interact with each other. First, the Score-P instrumentation and measurement system is described, followed by an overview of the Scalasca Tracing Tools package.

2.1 Score-P instrumentation and measurement system

Score-P is a portable and highly scalable instrumentation and performance measurement infrastructure jointly developed by a consortium of partners from Germany and the US under a 3-clause BSD open-source license. It supports profile and detailed event trace generation as well as an online interface for accessing profile data at runtime. Due to the common data formats CUBE4 for profiles and the Open Trace Format 2 (OTF2) for event traces, Score-P supports a number of analysis tools with complementary functionality. Currently, Score-P works with the Periscope Tuning Framework (TU Munich), Scalasca & Cube (jointly developed by Jülich Supercomputing Centre, GRS Aachen, and TU Darmstadt), Vampir (TU Dresden), and TAU (University of Oregon).

Figure 1 shows an overview of the Score-P architecture. Before performance data can be collected, the target application needs to be instrumented and linked to the Score-P measurement libraries, that is, extra code is inserted into the application’s code to intercept relevant events at runtime. This process can be accomplished in various ways, for example, by

- source-code annotations manually inserted by the user,
- leveraging automatic instrumentation functionality provided by the compiler,
- source-to-source pre-processing,
- linking to pre-instrumented libraries,
- function wrapping through symbol renaming at link time, or
- registering call-back functions with a particular runtime system.
Figure 1: Overview of the Score-P instrumentation and measurement system architecture and the interfaces to the supported analysis tools.

For each type of instrumentation, Score-P implements a corresponding instrumentation wrapper — a so-called adapter — which maps the specific instrumentation events onto more generic event handling functions provided by the Score-P measurement core. Here, the event-specific information (e.g., the source-code region being entered or the number of bytes transferred) is enriched with timestamps and hardware counter data (if configured), and then passed on to the profiling and/or tracing substrates. At the end of measurement, the collected profile and/or event trace data is written to disk, from where it can be consumed by the supported analysis tools. In addition, the online interface provides access to the profiling data already at runtime for use with online tools.

2.2 Scalasca Tracing Tools

The Scalasca Tracing Tools are a collection of trace-based performance analysis tools distributed under a 3-clause BSD open-source license that have been specifically designed for use on large-scale systems such as the IBM Blue Gene series or Cray XT and successors, but also suitable for smaller HPC platforms. A distinctive feature of the Scalasca Tracing Tools is its scalable automatic trace-analysis component which provides the ability to identify wait states that occur, for example, as a result of unevenly distributed workloads [1]. Especially when trying to scale communication intensive applications to large process counts, such wait states can present severe challenges to achieving good performance. In addition to identifying wait states and their root causes [2], the trace analyzer is also able to identify the activities on the critical path of the target application [3], highlighting those routines which determine the length of the program execution and therefore constitute the best candidates for optimization.
In contrast to previous versions of the Scalasca toolset – which used a custom measurement system and trace data format – the Scalasca Tracing Tools 2.x release series is based on the Score-P instrumentation and measurement infrastructure, which significantly improves interoperability between Scalasca and other performance analysis tool suites through the use of a common measurement system and data formats.

![Scalasca performance analysis workflow](image)

**Figure 2: Scalasca performance analysis workflow.**

Figure 2 depicts the basic Scalasca performance analysis workflow. First, the application has to be instrumented and linked to the Score-P measurement libraries as outlined in Section 2.1. The instrumented executable can then be run on the target system, by default generating a summary profile report in CUBE4 format. Using the `scalasca -examine` helper command, the resulting report file will be post-processed to split various generic metrics stored in the Score-P report, such as “Time”, into a hierarchy of more specific metrics, such as “MPI time” or “OpenMP barrier time”. This post-processed report is then opened in the Cube report viewer for interactive analysis by the user. Besides providing an initial performance overview, this profiling report can also be used to optimize the measurement configuration (e.g., by defining filters to exclude functions from measurement to reduce the runtime overhead introduced by the instrumentation). Once an appropriate measurement configuration has been found, and the profile analysis shows that a further in-depth analysis using event traces is required or worthwhile, the Score-P measurement mode can be changed to enable the generation of detailed event traces. After the application execution has finished and the event traces are flushed to disk, the Scalasca trace-analysis component can be run to automatically analyze the trace data. Since the trace analyzer is a parallel program in its own right and requires the same number of processes/threads than the target application, it is typically started as part of the same batch job. Therefore, Scalasca provides a convenience command which wraps the application launch command and automates the execution of the analyzer. As a result, the trace analyzer produces an analysis report (also in CUBE4 format) very similar to Score-P’s summary report, but enriched with additional, trace-based metrics. Again, a detailed metric hierarchy is generated during the report post-processing step before opening the Cube result browser.
3 Integrated hybrid analysis

In this section, we detail our modifications and enhancements in both Score-P and Scalasca towards an integrated analysis of applications using multiple parallel programming models in combination. This covers generic enhancements to support task-based programming models, instrumentation and measurement support for specific models, extended trace analyses, as well as improvements in the result visualization. We also briefly mention work carried out in the context of other projects to provide the full picture.

As the basis of our work, we started with the Score-P 1.2 and Scalasca 2.0 releases. Score-P v1.2 provided support for MPI 2.2 using PMPI function interposition, OpenMP 3.0 (except for untied tasks) using the OPARI2 source-to-source instrumenter, as well as CUDA using the CUPTI interface. Scalasca v2.0 included a scalable wait-state analysis for MPI 2.2 and OpenMP 2.5, except for nested OpenMP parallel regions. In addition, we built upon the experiences gained while developing the prototypical OmpSs support for Score-P during the first phase of the Mont-Blanc project.

3.1 Generic support for task-based programming models

In their general definition, tasks are functions or outlined code regions that can be executed independently from the location where they are created. That is, whenever a thread encounters a (model-specific) task creation construct, it creates a specific instance of executable code and its data environment, and adds it to the task queue of a runtime system for later execution on an arbitrary thread. This represents an inherently asynchronous execution model. The specifics of the task scheduling process, the task execution order, dependencies between tasks, or requirements to be side-effect free may vary between different runtime implementations or programming models. In this context, the basic objective for a measurement tool is to reconnect the creation and execution points of tasks to correctly attribute execution times and provide the user with a coherent view of his application run.

```c
void SCOREP_ThreadForkJoin_TaskCreate( SCOREP_ParadigmType paradigm, 
                                       uint32_t  threadId, 
                                       uint32_t  genNumber );

void SCOREP_ThreadForkJoin_TaskSwitch( SCOREP_ParadigmType paradigm, 
                                       SCOREP_TaskHandle  task );

SCOREP_TaskHandle SCOREP_ThreadForkJoin_TaskBegin( SCOREP_ParadigmType paradigm, 
                                              SCOREP_RegionHandle regionHandle, 
                                              uint32_t  threadId, 
                                              uint32_t  genNumber );

void SCOREP_ThreadForkJoin_TaskEnd( SCOREP_ParadigmType paradigm, 
                                        SCOREP_RegionHandle regionHandle, 
                                        SCOREP_TaskHandle  task );
```

Listing 1: Prototypes of the task event handling functions provided by the Score-P measurement core.
To create a generic task model for Score-P with low overhead, a minimal set of additional event calls has been defined. The four function calls shown in Listing 1 are provided by the Score-P measurement core and responsible for keeping the connection between creation and execution points. The TaskCreate call marks the creation point and generates a unique identifier out of thread-local information (identifier of the creating thread and a thread-local task creation counter), thus avoiding the need for locking. Based on this identifier, the TaskBegin and TaskEnd calls can manage task-local data with relation to the task creation point. The TaskSwitch call is responsible for tracking the actual context switch from one task to another, in particular since in the general case tasks can be suspended and resumed multiple times.

Due to its generic nature, Score-P’s task model can be used on any tasking system as long as it can be semantically mapped to this structure. Currently, it is employed for OpenMP, OmpSs, and MTAPI [4].

3.1.1 Score-P measurement and profiling

During a Score-P measurement run, the model-specific runtime events are mapped by the corresponding adapter to the core task calls shown above, which are responsible for tracking tasks and the connection between their creation and execution. This information is combined with events generated by other means of instrumentation, for example, automatic compiler-based function instrumentation, to create the corresponding call paths for task creation and the execution of the task itself.

To support untied tasks, that is, tasks that might migrate between threads during their execution, we extended Score-P’s measurement core to maintain a global task table rather than thread-local data structures to track all currently running tasks. This required a significant amount of refactoring and special care to avoid introducing too much runtime overhead, as tasks are usually only small units of work and thus task switching occurs very frequently. At this point, Score-P works for untied tasks according to the OpenMP 3.1 specification, i.e., suspension, resumption, and migration of tasks is only allowed at defined task scheduling points.

The further processing of the generic task events depends on the substrate used. In the case of tracing, the task events as well as the ENTER and LEAVE events of the respective regions are directly written to the OTF2 trace to be used for display and analysis in tools like Scalasca and Vampir. In the case of profiling, the events are post-processed before creating the resulting CUBE4 file. In light of the asynchronous nature of tasks and with particular regard to the possibility of suspending and resuming their execution, displaying a task’s possibly partial sub-call tree in the context of the creation point is often not consistently possible and also impedes readability. Therefore, we opted for moving the task executions and their call trees to a separate (artificial) root node in the Cube call tree display (see Section 3.2.2).

3.1.2 Scalasca trace analysis

With Scalasca v2.2, basic support for analyzing traces that contain tasking events has been released. One of the core requirements for Scalasca’s automatic trace analysis is the consistency of the trace, as the analysis is based on a replay approach and inconsistent data may lead to unexpected crashes or deadlocks during the analysis. Since tasks represent asynchronous execution, they basically have their own call stacks and history which is
independent from the main call stack. Therefore, we implemented a per-task call history and the independent management of the tasks’ respective call stacks. As the task executions are handled like functions, and thus the ENTER and LEAVE events of their regions are indistinguishable from “regular” function executions, the specific tasking events mentioned above as well as knowledge about the expected event order are used to trigger the call-stack management. With this knowledge the Scalasca analyzer is able to replay the event streams and analyze the trace data. However, due to inherent limitations of its current implementation only tied tasks can be supported. The results of the trace analysis follow the presentation style of the Score-P profile, using an artificial root node collecting the tasks’ sub-call trees as well as stub nodes to represent task executions.

3.2 Integrated result presentation of hybrid asynchronous applications

Besides the pure instrumentation, measurement, and analysis capabilities, providing an integrated presentation of the analysis results of hybrid asynchronous applications poses its own challenges. In this section, we outline the enhancements made to improve the user experience in such scenarios.

3.2.1 Analysis report post-processing

As outlined in Section 2.2, post-processing of Score-P summary reports as well as Scalasca trace analysis reports is an integral part of the Scalasca performance analysis workflow and supported by the scalasca –examine convenience command. Under the hood, this command performs a so-called remapping, which describes the process of restructuring and recalculation of metric tree of a CUBE4 result file. The objective of this restructuring is to increase the specificity of the information provided by the metrics, for example, by breaking the generic “Time” metric down into more fine-grained metrics such as “MPI communication time” or “OpenMP critical section time”. The remapping process takes the flat metric hierarchy of an input CUBE4 result file as produced by either Score-P or Scalasca and calculates such sub-metrics based on calculation rules defined in an XML specification file. The output of this remapping process is again a CUBE4 file.

Although being very similar, the Score-P and Scalasca reports obviously differ in the amount of detail they provide (i.e., in the number of included metrics). Thus, different calculation rules have to be applied during the remapping process. The scalasca –examine command of the Scalasca toolchain automatically chooses the correct specification file depending on the input and transparently executes the remapping process when a report is opened for the first time. The post-processed report is then cached in the experiment directory and opened directly if the result is examined again.

In the context of this deliverable, the remapping process has been improved in various ways:

- With Score-P supporting more and more parallel programming paradigms, additional model-specific sub-hierarchies were added to the remapper calculation specifications. In particular, we defined new sub-hierarchies for OmpSs, OpenCL, CUDA, and POSIX threads.

- Originally, all model-specific hierarchies were included in the post-processed analysis report file and subsequently shown to the user in the Cube browser. To improve the user experience, we reworked the remapping process such that only the respective
metric hierarchies of the programming models that are actually used by the application are calculated and shown.

- We defined an alternative metric hierarchy and implemented a corresponding remapper calculation specification (see the example below).

- With increasing numbers of processes and threads, the size of the generated Cube files also increases significantly — and with it the duration of the remapping process. However, since the remapping is carried out as part of the interactive result presentation, it has to be reasonably fast. As a side effect of reworking the calculation specification files, we were able to identify and eliminate a number of inefficiencies such as redundant calculations. In addition, several optimizations were applied to the remapping calculation routines in the Cube library. These efforts led to a significant reduction in both the time and memory required for the post-processing.

Figure 3 shows a comparison of the two alternative “Time” metric hierarchies for an example Score-P profile measurement run of the Gromacs simulation code. Gromacs is a package for molecular dynamics calculations with a primary focus on biochemical molecules like proteins, lipids, and nucleic acids. The simulation was configured to use MPI, OpenMP, and CUDA in combination. On the left hand side, the original “Time” metric hierarchy including the newly defined CUDA sub-hierarchy is shown. It breaks the overall execution time down into computation and time spent in activities related to the different programming models in use. This model-specific time is then split further into more specific categories (if applicable). The screenshot on the right shows the alternative metric hierarchy, which on the top level categorizes the execution time into computation and higher-level activities such as communication, synchronization, management, etc. The breakdown by programming model happens on the following level of the hierarchy.

The advantage of this alternative hierarchy is that it allows for an easier overview and direct answers to common questions such as “How much time does my application spend in communication or synchronization compared to computation?” even in the presence of multiple programming models. Thus, it can be considered to provide a more integrated view on the application’s performance behavior. It should be noted that these questions can also be answered using the original hierarchy by selecting multiple nodes in the metric hierarchy, however, this becomes tedious and error-prone the more programming models are involved.

We are currently in the process of evaluating the pros and cons of both hierarchies in close collaboration with a number of application developers, and also plan to present these two alternatives at upcoming workshops to get further feedback. Preliminary results indicate that both views may be appropriate depending on the user’s specific interest, and thus, a convenient way of selecting the preferred view or even interactively switching between them may be required.

---

1 The difference in computation time between the two hierarchies can be explained by the fact that the alternative specification does not yet handle CUDA kernel launches (4.05s) and uncategorized CUDA activities (6.19s). Thus, this time is still attributed as computation. This limitation will be fixed in the near future.
Figure 3: Original “Time” metric hierarchy (left) and alternative hierarchy (right) for an example profile measurement run of the Gromacs molecular dynamics simulation using MPI, OpenMP, and CUDA in combination.

3.2.2 Call tree visualization

The call tree view of the Cube display represents the call path information aggregated over all threads or in more general terms over all locations. In the traditional synchronous execution models, the call paths visited by the individual threads share some common parent call path, for example, an OpenMP parallel region construct. Thus, the corresponding sub-call trees can be presented as children of this parent call path. This also fits with the inclusive/exclusive property of the Cube call tree display, that is, a collapsed OpenMP parallel region call path node aggregates the inclusive value of the selected metric (e.g., time) across all threads, as the threads’ execution is entirely nested within the parallel region. The breakdown for the individual threads/locations then happens in the rightmost panel of the Cube display, the system tree.

This conformity of the call paths is lost when threads execute distinctly different call paths. For example, a POSIX thread started in one call path on the master thread executes completely independently from the master, typically runs longer than the function where the pthread_create call occurred, and can even be joined by another thread at a very different call path. This also happens for code run on accelerator locations, which execute their kernels while the CPU thread continues its own execution, or for threading locations when tasks are used and the worker threads just perform a sequence of task executions. While tasks, and to a degree also kernels, can have their own sub-call trees, they usually have relatively flat hierarchies since they are typically small execution units to support the automatic load balancing approach.
With the rising number of asynchronous parallel programming models, we concluded that a standardized representation of the asynchronous elements is necessary to direct the user to recognizable and recurring elements during the analysis of his hybrid code. Our current implementation design therefore introduces artificial top-level call tree nodes named THREADS, TASKS, and KERNELS. These nodes do not have a region counterpart in the application code, that is, only serve structural purposes by grouping the sub-call trees of the corresponding asynchronous constructs, and are always shown at the bottom of the Cube call tree panel. Moreover, we intentionally only distinguish between these coarse categories and not individual programming models to improve readability. While support for the THREADS and TASKS nodes is already available, we are currently finalizing the implementation to introduce the KERNELS node.

3.3 Support for additional programming models

To provide a comprehensive hybrid performance analysis to the application developer, Score-P was extended to support instrumentation and measurement of additional programming models besides MPI and OpenMP. In particular, support for OmpSs and OpenCL was developed as part of the Mont-Blanc project and is detailed in the following two subsections. Finally, the third subsection gives an overview of further programming models whose support was developed and added to Score-P in the context of other projects.

3.3.1 OpenCL

Support for instrumenting and measuring OpenCL applications was developed as part of the second phase of the Mont-Blanc project and has been released with Score-P v1.4. The instrumentation strategy is twofold. OpenCL API functions are intercepted by library wrapping, that is, symbol renaming at application link time. Additionally, kernel executions are buffered by the OpenCL runtime and processed by the Score-P measurement system at specific synchronization points or at the end of the measurement run, respectively. Aside from the kernel information, Score-P is also able to record OpenCL buffer read/writes and the memory usage of OpenCL through counting of allocations. Various measurement settings (e.g., the size of the event and command queue buffers or which OpenCL functionality to track) can be configured through environment variables.

In Figure 4, a screenshot of an example profile run of Gromacs is shown. Here, the simulation was configured to use MPI, OpenMP, and OpenCL. As can be seen, the OpenCL activity is recorded on a separate location in the system tree, identified by the device name (i.e., “Tesla K20Xm”), which is attached to the controlling process. The API calls appear at their respective positions in the main call tree. The user has to keep in mind that some of these, for example, memory transfers, due to their asynchronous nature happen in the background controlled by the OpenCL runtime. Therefore, the measured regions represent the scheduling point and reflect the respective source code locations.

---

2 To be precise, each OpenCL command queue created by the application will be represented by a separate location in the system tree.
Figure 4: Hybrid Gromacs run using MPI, OpenMP and OpenCL. Highlighted are exemplary OpenCL API calls executed by the master thread as well as the two OpenCL kernels executed on a Tesla K20Xm GPU.

3.3.2 OmpSs

Instrumentation of OmpSs applications is based on a runtime event system, in contrast to library wrapping or source-based instrumentation as used by OPARI2. Therefore, the task of supporting such a model consists of interpreting, filtering, and mapping the events generated by the runtime system to the semantics of the internally used concepts, e.g., task model, threading events, and call path regions. To allow access to the event system, the Nanos++ OmpSs runtime provides a plugin interface which enables tool developers to create a plugin to interface with their own measurement system.

A prototypical plugin for the Nanos++ OmpSs runtime as well as a corresponding Score-P adapter had already been developed during the first phase of the Mont-Blanc project. Throughout the second phase, we enhanced this interface in various ways and continuously updated the implementation to account for new developments in Nanos++ and Score-P.
Figure 5: Score-P and OmpSs runtime component interaction

Figure 5 shows how the OmpSs support based on the OmpSs plugin interface is implemented in Score-P. While the OmpSs plugin interface provides direct and detailed information from the runtime, it also introduces a strong dependency on the runtime implementation. Therefore, the plugin, although its source is packaged with Score-P, is built separately — if necessary with a different compiler. The plugin interacts with the Score-P measurement system through a minimal interface connecting to the OmpSs adapter, thus reducing the coupling through an additional level of abstraction. In this constellation, the plugin gathers the information of the runtime and sends a filtered set of events to the adapter, which maps these events to the corresponding Score-P region, threading, and task events. Since the OmpSs runtime is focused on tasks, almost all events passed through this interface are either task or basic thread information; user function or MPI instrumentation is provided through the respective existing Score-P adapters.

As Score-P’s OmpSs adapter maps the events provided by the plugin onto the generic task handling functions outlined in Section 3.1, the enhanced Scalasca trace analysis also works on OTF2 trace experiments generated from OmpSs applications without further modification, also when used in combination with MPI.

Figure 6: Example of a Scalasca trace analysis of a hybrid MPI+OmpSs n-body solver measured the Mont-Blanc prototype cluster.
The example in Figure 6 shows the result of a Scalasca trace analysis of an n-body solver. Although the program has been taskified and does the bulk of the computation in the task function `calculate_force_func`, the focus of the analysis shown in this screenshot is on the MPI waiting times. The post-processed metric tree in the left panel shows the wait-state patterns Scalasca identified in this measurement run, while the middle and right panel highlight where in call tree and system location these patterns occur.

In parallel to the OmpSs-specific plugin implementation, we participate in the discussions about a standardized OpenMP tools interface (OMPT). Details on the OMPT interface specification, the current state of the implementation in the Nanos++ runtime, and the prototypical tool implementations can be found in deliverable D5.6. When the OMPT interface is ratified and able to replace the aforementioned functionality, the necessity to provide an own tool-specific runtime plugin ceases to exist, as a standard OMPT plugin would be provided by the OmpSs runtime.

### 3.3.2.1 OmpSs using OpenCL targets

Within the Mont-Blanc project, OmpSs is also used in conjunction with OpenCL to leverage the computational power of the GPU available on the ARM-based Mont-Blanc cluster prototype. On the level of the OmpSs programming model, this results in the use of the `target device(opencl)` clause in the task statement. This clause indicates that the following task should be executed on the specified device instead of a CPU thread. However, since all OpenCL API calls are hidden inside the OmpSs runtime and are thus outside the area of influence of the user, it is not advisable to intercept and present all OpenCL calls using the method described in Section 3.3.1 to avoid confusion on the user’s side who did not write these OpenCL calls explicitly. Instead, the OmpSs runtime creates an additional CPU thread as a stand-in for the GPU target location and generates appropriate events to represent the execution of tasks as kernels on the GPU.

To create consistent and helpful results with Score-P, the measurement system on the one hand has to treat this OmpSs GPU thread internally as an additional CPU thread, and on the other hand has to visibly mark it as the GPU representative such that the user can easily distinguish between kernel executions on the CPUs and on the GPU. Otherwise, the event stream on the GPU thread is — from the Score-P perspective — indistinguishable from any other worker thread, showing a sequence of task executions. This also allows for a Scalasca trace analysis, which is not yet capable of handling “real” GPU locations.

Figure 7 shows an example profile of an n-body solver test case run on a single ARM node of the Mont-Blanc prototype cluster. In this screenshot, the integration of the GPU tasks into the Score-P task model can be seen. The GPU kernel `calculate_force_kernel` is treated in the same way as any other task (middle column), but can still be matched to the GPU by using the system tree location as reference (right column). The kernel name itself is chosen by the user, so it does not necessarily indicate a GPU kernel. In the metric tree (left column), task execution is categorized as computation, while task creation and the `taskwait` construct are considered synchronization and management overhead of the OmpSs programming model.
3.3.3 Programming model support developed outside of Mont-Blanc

3.3.3.1 MPI
Support for MPI 2.2 was already part of the initial Score-P design and is available since the first release. Instrumentation is done by providing a pre-instrumented library which leverages the standard PMPI profiling interface. Support for the latest MPI 3.1 standard version is currently under development.

3.3.3.2 OpenMP
Likewise, support for OpenMP 3.0 was a core functional requirement for the initial Score-P release. The primary way for instrumenting OpenMP constructs is via the OPARI2 source-to-source pre-processor. With respect to tasks, Score-P initially only supported tied tasks, however, as outlined in Section 3.1.1, support for untied task (i.e., task migration) has subsequently been added and was released with Score-P v1.4.

3.3.3.3 CUDA
CUDA instrumentation and measurement in Score-P follows a similar strategy as the OpenCL support described in Section 3.3.1. The CUPTI interface of CUDA provides the buffered device activity and these events are post-processed by the Score-P measurement system either at specific synchronization points or at the end of measurement. A major difference, however, is the way in which API functions are recorded, as CUPTI allows access to host-side runtime and driver events through call-back functions. Through a set of environment variables similar to the ones provided by the OpenCL support, the user can control the level of detail for the CUDA measurements, including a varying number of host-side call-backs as well as memory usage information. CUDA support was developed as part of the EU ITEA2 project H4H and released with Score-P v1.1.
3.3.3.4 SHMEM
As alternative inter-node paradigm to MPI, support for SHMEM was added with Score-P v1.3. It allows the instrumentation of SHMEM library calls for one-sided communication by library wrapping at link time. Currently, the implementation is known to work with the OpenSHMEM reference implementation, the Open MPI implementation, SGI SHMEM, and Cray SHMEM.

3.3.3.5 POSIX threads
Basic support for POSIX threads (Pthreads) instrumentation and measurement was developed in the context of the RAPID project [5] and introduced with Score-P v1.3. At present, Score-P only tracks the most important Pthreads routines, that is, functions that are responsible for creating, joining, synchronizing, and terminating POSIX threads. These routines are intercepted using symbol renaming at application link time. In addition to the instrumentation and measurement capabilities in Score-P, basic Pthreads support was also added to the Scalasca trace analyzer and released in version 2.2. In particular, the analyzer was extended to detect and quantify lock contention overheads – a new analysis that due to the generic event model used by OTF2 also works for OpenMP locks.

Figure 8: Example profile of the PEPC tree code for solving n-body problems (32 MPI ranks with 14 POSIX threads per rank on JUROPA).

Figure 8 shows a profile of PEPC [7] configured to use 32 MPI ranks with 14 simultaneous POSIX threads each. PEPC is a tree code for solving n-body problems and one of the Mont-Blanc applications. The most challenging component of PEPC in terms of scalability is the tree
traversal routine, which in this particular implementation was parallelized using a combination of MPI and Pthreads. In each iteration step, a group of Pthreads is created, one thread for communication and 13 threads for calculations. Each thread invokes the thread_helper routine, shown in a separate subtree under the artificial THREADS node in the call tree view (middle column). The run_communication_loop routine is called only by the communication thread, whereas all other threads call walk_worker_thread. As soon as all threads invoked the thread_helper routine, the main application thread waits for the termination of all child threads before proceeding to the next iteration. The system tree view (right column) displays all individual threads created throughout the execution. Since PEPC is an iterative solver, their overall number can be quite substantial. To address this issue, Score-P provides an experimental feature to reuse Pthreads locations, which can be enabled by setting a corresponding environment variable (i.e., SCOREP_PTHREAD_EXPERIMENTAL_REUSE).

3.3.3.6 Other create/wait threading models
Also in the context of the RAPID project, support for further create/wait threading models (i.e., Windows threads, Qt threads, and ACE threads) was integrated into Score-P. While Qt and ACE threads use either POSIX or Windows threads underneath, the goal in this project was to present the threading calls on the level of abstraction actually used by the application developer. To intercept the threading routines, the dynamic binary instrumentation tool Intel Pin [8] was used. Score-P and Intel Pin were coupled in order to produce profiles and traces compatible with Score-P, Cube, and Vampir. Results of this project are available to the project partners in a development branch of Score-P.

3.3.3.7 EMB² / MTAPI
Likewise, prototypical support for EMB² [6], a particular implementation of the Multicore Task Management API (MTAPI), was developed in the context of the RAPID project. Similar to other programming models, instrumentation is again based on symbol renaming at link time. This implementation is currently available in a Score-P development branch to the project partners.

4 Conclusion and future work

In this deliverable, we have detailed our modifications and enhancements to the Score-P instrumentation and measurement infrastructure as well as the Scalasca Tracing Tools package implemented within the Mont-Blanc project towards an integrated analysis of hybrid applications using multiple parallel programming models in combination. In particular, we focused on the support for the OmpSs and OpenCL programming models as well as the challenges introduced by the asynchronous nature of create/wait-type threading and task-based programming. Various examples show that Score-P and Scalasca effectively support the performance analysis of hybrid codes using a single, coherent workflow and a unified result presentation.

Obviously, supporting many different and evolving programming models is an ongoing effort. Therefore, we will update and enhance both Score-P and Scalasca to support functionality added with new revisions of the programming model specifications. A particular focus will be on supporting the final OMPT specification, including accelerator support via the target directive. In addition, we will continue to evaluate the result visualization alternatives in the Cube browser in close collaboration with application developers.
Acronyms and Abbreviations

- **ACE** Adaptive Communication Environment; Object-oriented network programming toolkit
- **API** Application Programming Interface
- **CUDA** Compute Unified Device Architecture; API to for GPGPU programming (NVIDIA)
- **CUPTI** CUDA profiling tools interface (NVIDIA)
- **MPI** Message passing interface standard (MPI Forum)
- **MTAPI** Multicore Task Management API (Multicore Association)
- **Nanos++** Task-oriented runtime system of BSC (used for OmpSs)
- **OPAPI2** Source-to-source preprocessor for OpenMP pragma and region instrumentation (JUELICH and others)
- **OpenCL** Open Computing Language; Industry standard for parallel programming of heterogeneous architectures (Khronos Group)
- **OpenMP** Industry standard for pragma-based parallel programming paradigm for shared memory computers (OpenMP ARB)
- **OmpSs** OpenMP extension (including task dependences and accelerator support) of BSC
- **OMPT** Draft standard OpenMP performance Tools interface (OpenMP ARB)
- **POSIX** Portable Operating System Interface (IEEE)
- **Scalasca** Parallel trace-based performance analyzer of JUELICH
- **Score-P** Parallel program instrumentation and measurement package of JUELICH, TU Dresden and other partners
- **SHMEM** Symmetric Hierarchical Memory access; family of parallel programming libraries, providing remote memory access via one-sided communications.
References


