D5.6 Monitoring and Control API Implementation
Version 1.0

Document Information

Contract Number	288777
Project Website	www.montblanc-project.eu
Contractual Deadline	PM24
Dissemination Level	PU
Nature	O
Authors	Xavier Martorell (BSC)
Contributors	Harald Servat, Roger Ferrer, Xavier Teruel (BSC) Mathias Nachtmann, Jose Gracia (HLRS), Bernd Mohr, Markus Geimer (JUELICH)
Reviewer	Simon McIntosh-Smith (University of Bristol)
Keywords	Performance Analysis, Debugging, Monitor and Control API

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.

Change Log

Version	Description of Change
V0.1	Initial Draft (document structure + main ideas)
V0.2	Jülich changes (Score-P & monitoring interaction)
V0.3	HLRS changes (Ayudame/Temanejo, monitoring and debug interaction)
V0.4	General contents; Mercurium and Nanos++ continuation. Extrae contents.
V0.5	Small changes, executive summary & conclusions. Ready for partners review.
V0.6	Additional changes. Partner’s corrections.
V1.0	Final version

Table of Contents

Executive Summary

This document describes the work done to implement the monitoring and control API specified in the previous D5.1 Mont-Blanc 2 deliverable. First, we present the description of the software components that define the API from the programming model perspective (OmpSs, Mercurium, and Nanos++) and the monitoring/debugging tools (Extrae/Paraver, Ayudame/Temanejo, and Score-P/Scalasca). Then, we go into details of the implementations developed in this period, to provide the functionality associated with the API.

We divided the implementation description into two different sections: one devoted to performance and monitoring tools and the other to debugging tools. Extrae and Score-P have been extended with the support of the OMPT specification described in the OpenMP technical report 2. The Ayudame/Temanejo interface has been extended to support task management (TCA) for the debugging of OmpSs programs, including the abilities to stop and resume tasks.

Introduction

In the second phase for task T5.1 of work package 5 (“development tools”), we continue with the implementation of a common interface to allow performance and debug tools to interact with the OmpSs runtime system.

In deliverable D5.1, a monitoring and control API specification was established. This API is derived from the OMPT specification published by the OpenMP Architecture Review Board and used to monitor OpenMP applications. Within the Mont-Blanc project we have proposed an extension to monitor task dependencies. We also started discussions on the introduction of additional events into OMPT to track the accelerator activity; however this work has been already proposed to OMPT.

With respect to debugging control, several services have been implemented in the core of the Nanos++ runtime library and the Mercurium compiler. Most of the changes are devoted to handle dependencies and to step program execution. The compiler has been extended as well to allow users to follow the original source code (instead of compiler generated code) when debugging their applications. The model has been also extended with the Tasking Control API (TCA), as defined in D5.1. Its design is inspired by the OMPT philosophy and, in the future, it could become part of the OMPT instrumentation plugin.

The following sections will describe the software components involved in this task and the corresponding implementations of the performance monitoring and debugging control interfaces.

1 Software components

1.1 OmpSs main components

OmpSs is an effort to integrate features from the StarSs programming model developed by BSC into a single programming model. In particular, our objective is to extend OpenMP with new directives to support asynchronous parallelism and heterogeneity (devices like GPUs). However, it can also be understood as new directives extending other accelerator based APIs like CUDA or OpenCL. Our OmpSs environment is built on top of our Mercurium compiler and Nanos++ runtime system

1.1.1 Mercurium source-to-source compiler

Mercurium is a source-to-source compilation infrastructure aimed at fast prototyping. Current supported languages are C, C++, and Fortran. Mercurium is mainly used in the Nanos environment to implement OpenMP, but since it is quite extensible it has been used to implement other programming models or compiler transformations. Examples include Cell Superscalar, Software Transactional Memory, Distributed Shared Memory or the ACOTES project, just to name a few.

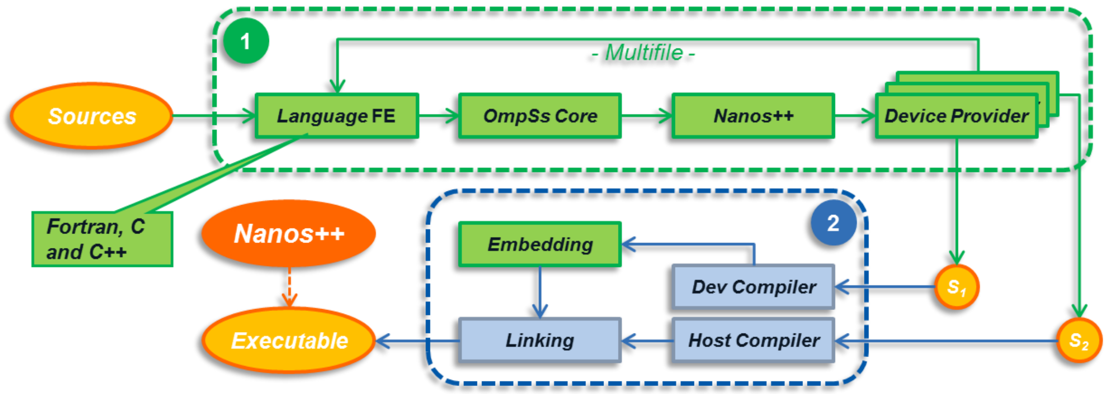


Figure 1: OmpSs compilation tool-chain, including (1) source-to-source transformation phase and (2) native compilation and link phase.

Extending Mercurium is achieved using a plugin architecture, where plugins represent several phases of the compiler. These plugins are written in C++ and dynamically loaded by the compiler according to the chosen configuration. Code transformations are implemented in terms of source code thus there is no need to modify or know the internal syntactic representation of the compiler.

1.1.2 Nanos++ Runtime Library

The Nanos++ Runtime Library has been designed to serve as runtime support in parallel environments and is mainly used to support the OmpSs programming model. It provides several services to support data and task parallelism: creating and scheduling tasks, synchronizing them using point-to-point mechanisms (such as taskwait and data dependencies) and group-to-group mechanisms (including atomic, critical sections and locks), maintaining coherence across different address spaces automatically (through software cache/directory), allowing task context switches (user-level threads, when the native architecture allows it), or supporting other common worksharing constructs by mapping them on top of the tasking support.

The main goal of Nanos++ is to be used in research of parallel programming environments by enabling easy development through modular components. In addition, some of these components have been designed to be extensible by means of plugins, which allow external developers to include new components which can be used by setting the corresponding environment variable during the application execution. The instrumentation module is one of these extensible components. Its main purpose is to provide useful information about program execution. Such collected data is represented internally using Nanos++ events.

A Nanos++ event occurs at the time (timestamp) and from the thread (thread identifier) it is generated and provides a key-value pair. The key is a unique identifier for the type of the event (e.g. task creation) and the value is either the data that is identified or a pointer to the location of that data (e.g. task identifier).

Nanos++ also provides an interface to allow the Mercurium compiler to introduce new events. These events can be generated from the user's code after the compiler transformation. This functionality could be useful to add events when creating the outlined code for a task region, among other cases. For every task, the compiler introduces a begin task event at the beginning of the task body, and an end task event at the end of the task body.

1.2 Performance and debug tools

1.2.1 The BSC performance tool-suite

The BSC performance tool-suite consists of several packages that include Paraver, Dimemas and Extrae in addition to several satellite tools that are able to process Paraver trace-files to extract additional performance insight. From all these tools, this deliverable involves some implementation changes in Extrae and uses Paraver to depict the outcome of these changes.

Extrae is the instrumentation package of the BSC tool-suite and generates Paraver trace-files. This package instruments multiple parallel programming paradigms including MPI, OpenMP, OmpSs, POSIX threads, CUDA, OpenCL, and some combinations of these paradigms. The package also includes sampling mechanisms that are fired using regular alarms or the processor performance counters. The information captured by Extrae typically includes information from the parallel programming model (for instance, message information from MPI) as well as performance counters (through the PAPI interface) and the call-stack information to correlate the measurements with the actual source code.

Paraver is the visualization tool from the BSC tool-suite. This tool was developed to respond to the need to have a qualitative global perception of the application behavior by visual inspection and then to be able to focus on the detailed quantitative analysis of the problems. Expressive power, flexibility and the capability of efficiently handling large traces are key features addressed in the design of Paraver. The clear and modular structure of Paraver plays a significant role towards achieving these targets. Its analysis power is based on two main pillars. First, the trace format has no semantics; extending the tool to support new performance data or new programming models requires no changes to the visualizer, just to capture such data in a Paraver trace. The second pillar is that the metrics are not hardwired in the tool but programmed. To compute them, the tool offers a large set of time functions, a filter module, and a mechanism to combine two time lines. This approach allows displaying a huge number of metrics with the available data. To capture the expert’s knowledge, any view or set of views can be saved as a Paraver configuration file. After that, re-computing the view with new data is as simple as loading the saved file. The tool has been demonstrated to be very useful for performance analysis studies, giving much more details about an application’s behaviour than most performance tools.

1.2.2 Ayudame/Temanejo toolchain

Temanejo started as a graphical debugger for the task-parallel, data-dependency-driven programming model StarSs. The foremost goal was to display the task-dependency graph of StarSs applications, and to allow simple interaction with the SMPSs runtime system in order to control some aspects of the parallel execution of an application.

Nowadays it supports several programming models that can meaningfully define the concepts of tasks and dependencies between them. Notably, this includes OpenMP and MPI, the working horses in high-performance computing, and also targets StarSs/OmpSs, StarPU and PaRSEC.

Temanejo actually is only the graphical frontend. Most of the real work is done by a library called Ayudame which is used to receive information, so called *events*, from supporting runtime systems, and to exert control over a runtime system by issuing requests to it.

1.2.3 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 (Jülich Supercomputing Centre, GRS Aachen, TU Darmstadt), Vampir (TU Dresden), and TAU (University of Oregon).

Figure 2 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, manually through the user, by leveraging 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. For each type of instrumentation, Score-P implements an instrumentation wrapper – a so-called adapter – which maps the specific instrumentation events onto more generic event functions provided by the Score-P measurement core. Here, the specific event information is enriched with timestamps and hardware counter information (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.

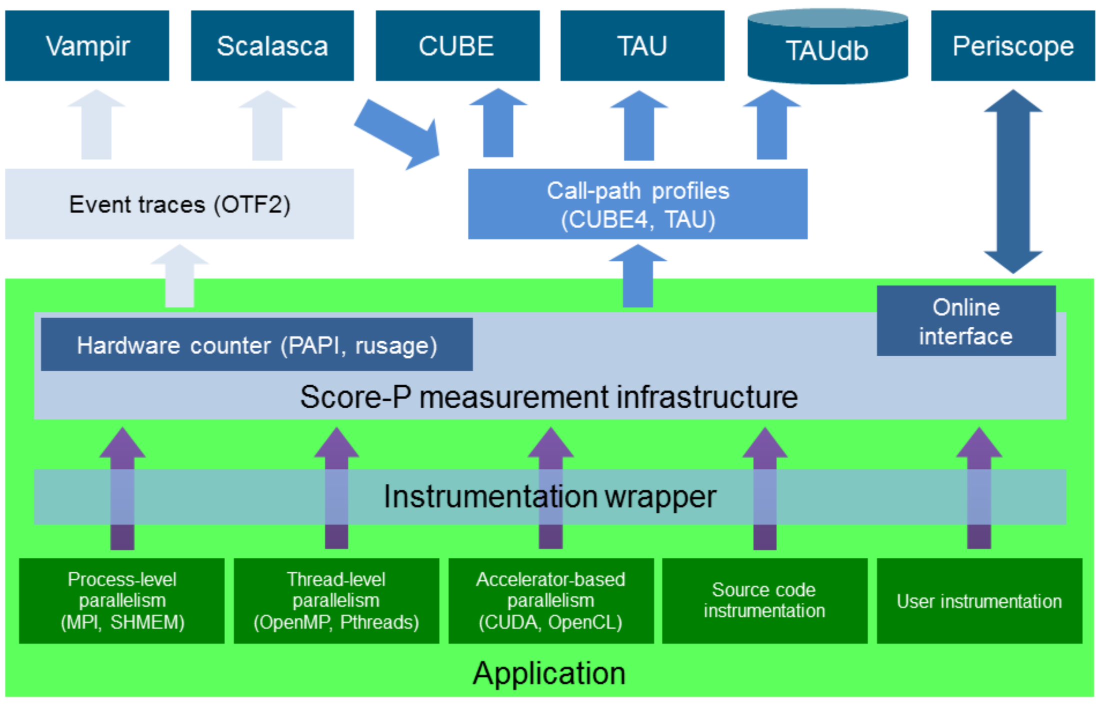

Vampir
Scalasca
CUBE
TAU
TAUdb
Periscope
Event traces (OTF2)
Call-path profiles (CUBE4, TAU)
Online interface
Hardware counter (PAPI, rusage)
Score-P measurement infrastructure
Instrumentation wrapper
Process-level parallelism (MPI, SHMEM)
Thread-level parallelism (OpenMP, Pthreads)
Accelerator-based parallelism (CUDA, OpenCL)
Source code instrumentation
User instrumentation
Application

Figure 2: Overview of the Score-P instrumentation and measurement system architecture and the interfaces to the supported analysis tools.

2 Performance monitoring API

In this section we will discuss the implementation of the Performance Monitoring API specification into the Nanos++ Runtime Library and the Mercurium compiler.

The Performance Monitoring API specification originally derived from the OMPT technical report released by the OpenMP Architecture Review Board on its official web site (www.openmp.org). This technical report specifies several mechanisms to allow performance tools to capture application activity. The information provided by performance tools will then be interpreted by programmers in order to allow them to understand what the main issues regarding their program executions are and can thus potentially be used to improve application performance. The specification has been extended to instrument task dependencies (as reported in D5.1). Also, the OMPT working group is currently discussing an extension to instrument accelerators and we will follow its specification to see how to implement it in OmpSs.

2.1 *Mercurium/Nanos++ performance monitoring implementation*

As stated in the OMPT technical report, section 6.1 Initialization of a Tool: “A tool must register itself with an OpenMP runtime by providing an implementation of the following [`ompt_initialize()`] function”. The initialization of the Nanos++ RTL occurs in the global object construction phase (before the user's program entry point has started the execution).

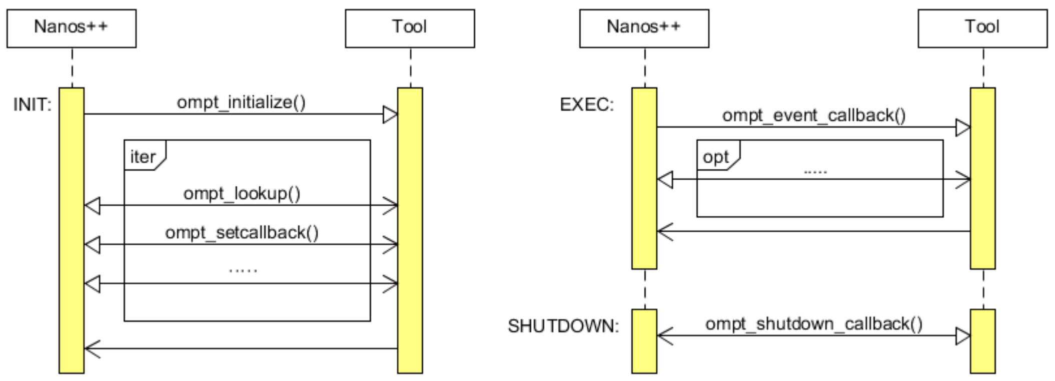


Figure 3: Interaction between the Nanos++ RTL and the instrumentation tool during the OMPT instrumentation (initialization, execution and shutdown).

During the initialization phase (labelled as `INIT` in Figure 3) the runtime calls the `ompt_initialize()` routine and expects several calls from the tool. At this point the tool will query, using the `ompt_lookup()` routine, the implementation status of several potential services implemented by the runtime (inquiry services). The tool will also set, through the `ompt_setcallback()` routine, the different tool routines that will be triggered when the corresponding event occurs (event callbacks).

The `ompt_lookup()` routine receives as the only parameter the name of the service the tool is requiring, and will return the routine address if it is implemented. These callbacks are inquiry services used by the tool in order to determine the current status of the runtime. Inquiry services which have been already implemented by the Nanos++ runtime are described in Section 2.1.2.

```
ompt_interface_fn_t lookup(
   const char *interface_function_name
);
```

The `ompt_setcallback()` routine will update an internal table containing the tool's routine addresses invoked when a given event occurs. So, with that mechanism, the tool is configuring the level of monitoring detail for the current execution. At this point the runtime also informs about the current status concerning the event monitoring (i.e. the event is implemented, it is not implemented or it is partially implemented). The event generation callbacks which are already monitored in Nanos++ runtime are described in Section 2.1.1.

```
OMPT_API int ompt_set_callback(
   ompt_event_t event,
   ompt_callback_t callback
);
```

In addition Nanos++ implements the `ompt_getcallback()` routine. This is a special inquiry service which may be used to check whether a callback has been registered or not.

```
OMPT_API int ompt_get_callback(
   ompt_event_t event,
   ompt_callback_t *callback
);
```

If the requested callback has been registered, the routine will return 1 (true) and will set the callback parameter to the address of the corresponding callback function; otherwise the routine will return 0 (false).

During the execution of the application (labelled as `EXEC` in Figure 3), Nanos++ will only generate the events that the instrumentation tool requested in the previous phase. When the Nanos++ runtime calls one of these services the instrumentation tool may query the status of the current execution using the inquiry services described in Section 2.1.2.

During the shutdown phase (labelled as `SHUTDOWN` in Figure 3), the Nanos++ runtime informs the instrumentation tool that there will be no more event callbacks. This callback enables a tool to clean up its state and finalize its execution as appropriate.

2.1.1 Event callbacks

This section describes the different events generated by the Nanos++ RTL if the instrumentation tool has registered a callback function during the initialization phase. Callbacks for different events may have different type signatures and may provide different information when invoked (through the corresponding parameters).

A Nanos++ event is a pair of type and value items. In some situations, several of these events are generated at the same time and the OMPT plugin has to look for the different components to assemble the final `ompt_callback()`. The following example shows how to assemble the `ompt_event_dependence()` callback. In this example, we are merging the information of three different Nanos++ events. First, a dependence event used to provide task identifiers for sender and receiver, a second Nanos++ event that provides the object address involved in the dependence, and a third Nanos++ event indicating the direction of this dependence (in, out, or inout).

```
void addEventList ( unsigned int count, Event *events )
{
   // ...
   switch ( e.getType() ) {

      case NANOS_POINT:
         // ...
         if ( e.getKey() == dependence ) {

            // Getting sender and receiver
            nanos_event_value_t dependence_value = e.getValue();
            int sender_id = (int) ( dependence_value >> 32 ) & 0xFFFFFFFF;
            int receiver_id = (int) ( dependence_value & 0xFFFFFFFF );

            void * address_id = 0;

            // Getting dep_address event
            if ( dep_address != 0xFFFFFFFF ) {
               unsigned int j = i;
               while ( (j<count)&&
                   ((nanos_event_key_t)(events[j]).getKey()!=dep_address) )
                   j++;
               if ( j<count) address_id=(void *)((events[j]).getValue());
            }

            // Getting dep_direction event
            int direction = 0;
            if ( dep_direction != 0xFFFFFFFF ) {
               unsigned int j = i;
               while ( (j<count)&&
                   ((nanos_event_key_t)(events[j]).getKey()!=dep_direction) )
                   j++;
               if (j<count) direction = (int)((events[j]).getValue());
            }

            // Invoking the ompt callback
            ompt_nanos_event_dependence(
                   (ompt_task_id_t) sender_id,
                   (ompt_task_id_t) receiver_id,
                   (ompt_dependence_type_t) direction,
                   (void *) address_id
            );

      }
      break;
      // ...
   }
   //...
}
```

The list of mandatory events generated by the Nanos++ RTL is:

- `ompt_event_parallel_begin()`: the runtime invokes this callback after a task encounters a parallel construct but before any implicit task starts to execute the parallel region's work.
- `ompt_event_parallel_end()`: the runtime invokes this callback after a parallel region executes its closing synchronization barrier but before resuming execution of the parent task.

- `ompt_event_task_begin()`: the runtime invokes this callback after a task encounters a task construct but before the new explicit task executes.
- `ompt_event_task_end()`: the runtime invokes this callback after an explicit task completes but before the thread resumes execution of another task.
- `ompt_event_thread_begin()`: the runtime invokes this callback in the context of an initial thread just after it initializes the runtime for itself, or in the context of a new thread created by the runtime system just after the thread initializes itself.
- `ompt_event_thread_end()`: the runtime invokes this callback after an OpenMP thread completes all of its tasks but before the thread is destroyed.
- `ompt_event_control()`: if the user program calls ompt_control(), the runtime invokes this callback.
- `ompt_event_shutdown()`: the runtime system invokes this callback before it shuts down the runtime system.

The list of optional events generated by the Nanos++ RTL is:

- `ompt_event_task_switch()`: the runtime system invokes this callback after it suspends one task and before it resumes another task.
- `ompt_event_implicit_task_begin()`: the runtime system invokes this callback after an implicit task is fully initialized but before the task executes its work.
- `ompt_event_implicit_task_end()`: the runtime system invokes this callback after an implicit task executes its closing synchronization barrier but before returning to idle or the task is destroyed.
- `ompt_event_barrier_begin()`: the runtime system invokes this callback before an implicit task begins execution of a barrier region.
- `ompt_event_barrier_end()`: the runtime system invokes this callback after an implicit task exits a barrier region.

The additional event (dependencies) generated by Nanos++ RTL is:

- `ompt_event_dependence()`: the runtime system invokes this callback when a blocking dependence (i.e. a dependence blocking the successor task) has been produced.

The implementation of these services required, in some cases, a modification in the runtime core instrumentation module. One example is the pair parallel begin/end. In this case the OMPT specification required the outlined parallel routine be generated by the compiler for the corresponding parallel region. In our previous implementation, this information was not available at team creation, so we needed to include this information just for instrumentation purposes.

2.1.2 Inquiry services

In order to enable a performance tool to identify all the possible states that Nanos++ supports, the `ompt_enumerate_states()` routine is provided. This function allows iterating through all the different values which can potentially be returned as current thread state. It also relates the state with a string description. Given the current_state, the runtime will return the value and description for the next state.

```
OMPT_API int ompt_enumerate_state(
   ompt_state_t  current_state,
   ompt_state_t *next_state,
   const char   **next_state_name
);
```

An example of how to enumerate the states supported by an OpenMP runtime system is shown below:

```
ompt_state_t state = ompt_state_first;
const char *state_name;
while (ompt_enumerate_state(state, &state, &state_name)) {
   // runtime supports ompt_state_t "state"
   // associated with "state_name"
}
```

The function ompt_get_thread_id() is the inquiry function to determine the thread ID of the current thread. Performance tools can use this service to determine the calling thread of a given callback_event routine.

```
OMPT_API ompt_thread_id_t ompt_get_thread_id();
```

The function ompt_get_state() is the inquiry function to determine the state of the current thread. This routine allows us to determine if the current thread is executing sequential code (main code, first level task), potentially parallel code (an explicit task), or the idle loop (no assigned work to execute).

```
OMPT_API ompt_state_t ompt_get_state(ompt_wait_id_t *wait_id);
```

It is important to note that OmpSs has a different approach than OpenMP with respect to the fork-join model and the semantics for parallel and serial executions are different. In OmpSs the parallel directive is ignored (no new team of threads is created) but a global implicit team exists for the execution of the whole program. Tasks created during program execution can be run by any thread in this implicit team. In OmpSs the serial state is used for the implicit initial task (as is the case of OpenMP), and the parallel state for any explicit task (included or not in a parallel construct, due any task can potentially be executed in parallel with any other task).

Nanos++ only distinguishes between three different state values:

- ompt_state_work_serial
- ompt_state_work_parallel
- ompt_state_idle

The function `ompt_get_parallel_id()` is the inquiry function to determine the parallel identifier for the current (or any ancestor) parallel region. Two different parallel regions must have different identifiers if they are executed simultaneously. Once a parallel region has finished its execution, the identifier can be reused for a future one.

```
OMPT_API ompt_parallel_id_t ompt_get_parallel_id(int ancestor_level);
```

The function `ompt_get_parallel_team_size()` is the inquiry function to determine the number of threads executing the current (or any ancestor) parallel region.

```
OMPT_API int ompt_get_parallel_team_size(int ancestor_level);
```

The function `ompt_get_task_id()` is the inquiry function to determine the task identifier for the current (or any ancestor task, according to the depth parameter) task region. Two different tasks must have different identifiers only if they can be executed simultaneously. As in the case of parallel identifiers, once a task region has finished its execution, the identifier can be reused for a future task.

```
OMPT_API ompt_task_id_t *ompt_get_task_id(int depth);
```

2.1.3 Entry/exit points

Besides the OMPT specific implementation and in order to improve debugging and tracing, as required by our partners, we extended the Nanos++ runtime and the Mercurium compiler to provide instrumentation of the program entry point (i.e. `main()` in C/C++ and the `PROGRAM` unit in Fortran) of OmpSs applications.

On the runtime side we added the following services notifying the Nanos++ runtime when the entry point of the program has been reached or is about to be exited.

```
void ompss_nanox_main_begin();
void ompss_nanox_main_end();
```

It is the responsibility of the Mercurium compiler to add these calls to the OmpSs program. Thus, we extended it with a new compiler phase which adds these calls. The call to `ompss_nanox_main_begin()` is placed at the beginning of the application entry point. Conversely we want to call `ompss_nanox_main_end()` when the program ends even if it does not exit through main (e.g. by calling `exit()`). For this purpose we register an `atexit()` handler which will call `ompss_nanox_main_end()`.

While `atexit()` is available in C and C++, it is not in Fortran. In order to get the same functionality in OmpSs programs written in Fortran, we added a convenience API `nanos_atexit()` that can be called from Fortran programs and has the same functionality as C `atexit()`.

2.2 Extrae/Ompss performance monitoring interaction

2.2.1 Previous OmpSs instrumentation

The Extrae instrumentation package already offered an instrumentation library for the OmpSs programming model. This library was created as an ad-hoc solution taking advantage of the Extrae API to generate events, and it was necessary to implement some specific APIs to support OmpSs. In this direction, all the events are generated from the instrumentation plugin from the Nanos++ run-time system.

2.2.2 Extending Extrae to support OMPT

In this work, Extrae has been extended to support the OMPT TR-2 specification that is being discussed for inclusion in the OpenMP language specification. Since OMPT matches the OpenMP run-time and Extrae already instrumented OpenMP, the OMPT callbacks have been leveraged into the existing OpenMP probes. The forthcoming listings serve as an example of setting up the OMPT interface within Extrae and how the forwarding of the OMPT callbacks is implemented. See the forthcoming Tables as an example of setting up the OMPT interface and forwarding the OMPT callbacks into the OpenMP probes as it is included in Extrae version 3.1.0.

In the initialization process (see the following listing) the hooks are paired with the callbacks in the OMPT_callbacks and OMPT_callbacks_locks mapping structures. Since instrumenting locks involves a large overhead, Extrae allows activating or deactivating the lock instrumentation at the user's request. Therefore, the second structure will be processed only if it is requested by the user.

```
static struct OMPT_callbacks_st ompt_callbacks[] = {
  CALLBACK_ENTRY(ompt_event_parallel_begin, OMPT_event_parallel_begin),
  CALLBACK_ENTRY(ompt_event_parallel_end, OMPT_event_parallel_end),
  CALLBACK_ENTRY(ompt_event_barrier_begin, OMPT_event_barrier_begin),
  CALLBACK_ENTRY(ompt_event_barrier_end, OMPT_event_barrier_end),
  …
  { "empty,", (ompt_event_t) 0, 0 },
};

struct OMPT_callbacks_st ompt_callbacks_locks[] = {
  CALLBACK_ENTRY(ompt_event_master_begin, OMPT_event_master_begin),
  CALLBACK_ENTRY(ompt_event_master_end, OMPT_event_master_end),
  CALLBACK_ENTRY(ompt_event_single_others_begin,
                 OMPT_event_single_others_begin),
  CALLBACK_ENTRY(ompt_event_single_others_end,
                 OMPT_event_single_others_end),
  …
  { "empty,", (ompt_event_t) 0, 0 },
};

int ompt_initialize( ompt_function_lookup_t lookup,
                     const char *runtime_version_string,
                     unsigned ompt_version )
{
  int i,r;

  ompt_set_callback_fn = (int(*)(ompt_event_t, ompt_callback_t))
                       lookup("ompt_set_callback");
  assert (ompt_set_callback_fn != NULL);

  ompt_get_thread_id_fn = (ompt_thread_id_t(*)(void))
                        lookup("ompt_get_thread_id");
  assert (ompt_get_thread_id_fn != NULL);

  i = 0;
  while (ompt_callbacks[i].evt != (ompt_event_t) 0) {
    r = ompt_set_callback_fn (ompt_callbacks[i].evt, ompt_callbacks[i].cbk);
    i++;
  }
  if (getTrace_OMPLocks()) {
    i = 0;
    while (ompt_callbacks_locks[i].evt != (ompt_event_t) 0) {
      r = ompt_set_callback_fn (ompt_callbacks_locks[i].evt,
                                ompt_callbacks_locks[i].cbk);
      i++;
    }
  }
}
```

The forwarding methodology is exemplified in the following listing depicting two callbacks (for parallel regions and barriers) and two placements (begin and end) which are forwarded to the Extrae OpenMP instrumentation infrastructure.

```
void OMPT_event_parallel_begin (ompt_task_id_t tid, ompt_frame_t *ptf,
                                ompt_parallel_id_t pid,
                                uint32_t req_tsize, void *pf)
{
  UNREFERENCED_PARAMETER(ptf);
  UNREFERENCED_PARAMETER(tid);
  UNREFERENCED_PARAMETER(req_tsize);
  PROTOTYPE_MESSAGE(" (%ld,%p,%ld,%u,%p)", tid, ptf, pid, req_tize, pf);
  Extrae_OMPT_register_ompt_parallel_id_pf (pid, pf);
  Extrae_OpenMP_ParRegion_Entry ();
  Extrae_OpenMP_EmitTaskStatistics();
}

void OMPT_event_parallel_end (ompt_parallel_id_t pid, ompt_task_id_t tid)
{
  UNREFERENCED_PARAMETER(tid);
  PROTOTYPE_MESSAGE(" (%ld, %ld)", pid, tid);
  Extrae_OMPT_unregister_ompt_parallel_id_pf (pid);
  Extrae_OpenMP_ParRegion_Exit();
  Extrae_OpenMP_EmitTaskStatistics();
}

// …

void OMPT_event_barrier_begin (ompt_parallel_id_t pid, ompt_task_id_t tid)
{
  UNREFERENCED_PARAMETER(pid);
  UNREFERENCED_PARAMETER(tid);
  PROTOTYPE_MESSAGE(" (%ld, %ld)", pid, tid);
  Extrae_OpenMP_Barrier_Entry ();
}

void OMPT_event_barrier_end (ompt_parallel_id_t pid, ompt_task_id_t tid)
{
  UNREFERENCED_PARAMETER(pid);
  UNREFERENCED_PARAMETER(tid);
  PROTOTYPE_MESSAGE(" (%ld, %ld)", pid, tid);
  Extrae_OpenMP_Barrier_Exit ();
}
```

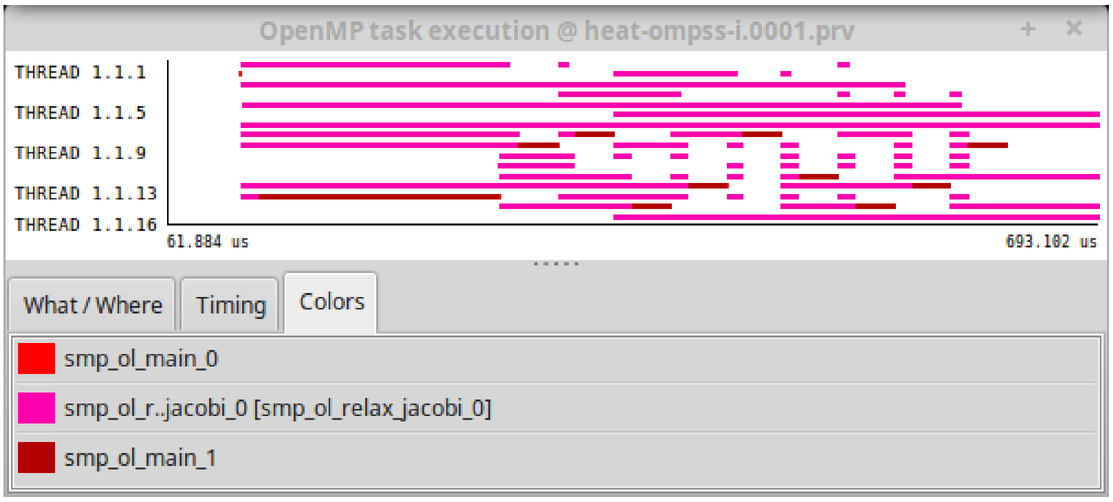
The callback for the entry in a parallel region includes registering the parallel region through the `Extrae_OMPT_register_ompt_parallel_id_pf()` routine, generating an event identifying the entry calling the Extrae service to specify a parallel entry point: `Extrae_OpenMP_ParRegion_Entry()`; and also emitting some statistics at that point using `Extrae_OpenMP_EmitTaskStatistics()`. Other callbacks, such as the barriers, are much simpler because they only invoke a single call to the Extrae OpenMP instrumentation counter-part.

The inclusion of OMPT into Extrae required additional helper structures and functions. These functions are aimed at bookkeeping information regarding the thread, the parallel regions, and the task identifiers.

On the one hand, threads are identified within OpenMP and OmpSs in the range [0..N-1] where N is the maximum number of threads. However, thread identifiers in OMPT are an arbitrary unsigned 64 bit value (ompt_thread_id_t), and therefore Extrae needs to map thread ids into a thread identifier within the [0..N-1] range. This map gets populated every time a new thread creation is notified through the `OMPT_event_thread_begin()`.

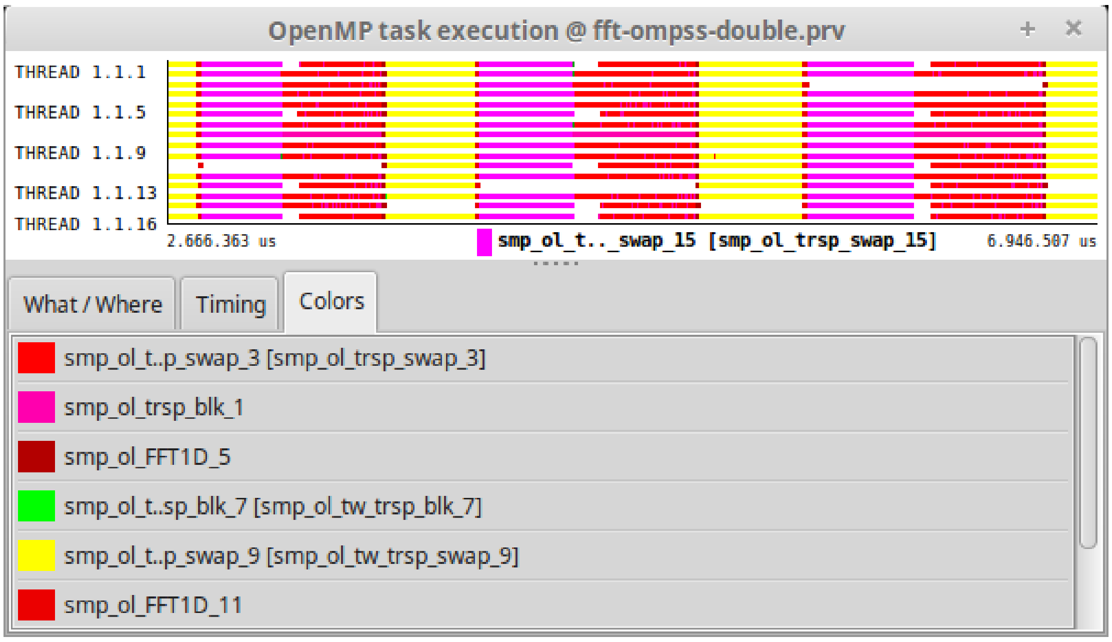
On the other hand, Extrae captures the pointer to the outlined OpenMP function to report the active OpenMP region of code to the user. This information is only received in a subset of the OMPT callbacks (such as `ompt_event_task_begin`, `ompt_event_workshare_begin` and `ompt_event_parallel_begin`) in addition to an OMPT identifier which is passed to every callback. Extrae, however, requires the address of the outlined routine and therefore needs to create a map between the address received and the given identifier. For instance, the callback `OMPT_event_parallel_begin` from the previous listing registers the parallel region identifier with the outlined parallel function, which is unregistered in `OMPT_event_parallel_end`. This map will be queried at any point that the instrumentation package needs this information.

Figure 4, Figure 5 and Figure 6 exemplify several time-lines for three OmpSs benchmarks executed on one node of the MareNostrum3 system [4]. These time-lines show in the X-axis the time, in the Y-axis the application thread, and the coloring indicates the task that was executing. The first figure indicates that most of the time is spent in the smp_ol_relax_jacobi_0 routine (which is depicted in pink). The white coloring in the time-line indicates that threads are idle waiting for more work to process. This is likely to happen because the tasks are waiting for their dependencies. The applications shown in the two latter figures indicate a repetitive execution pattern along time. Interestingly, the execution of the FFT benchmark (Figure 5) shows that there is always one thread idling while waiting for work to process, and that this thread varies over time.



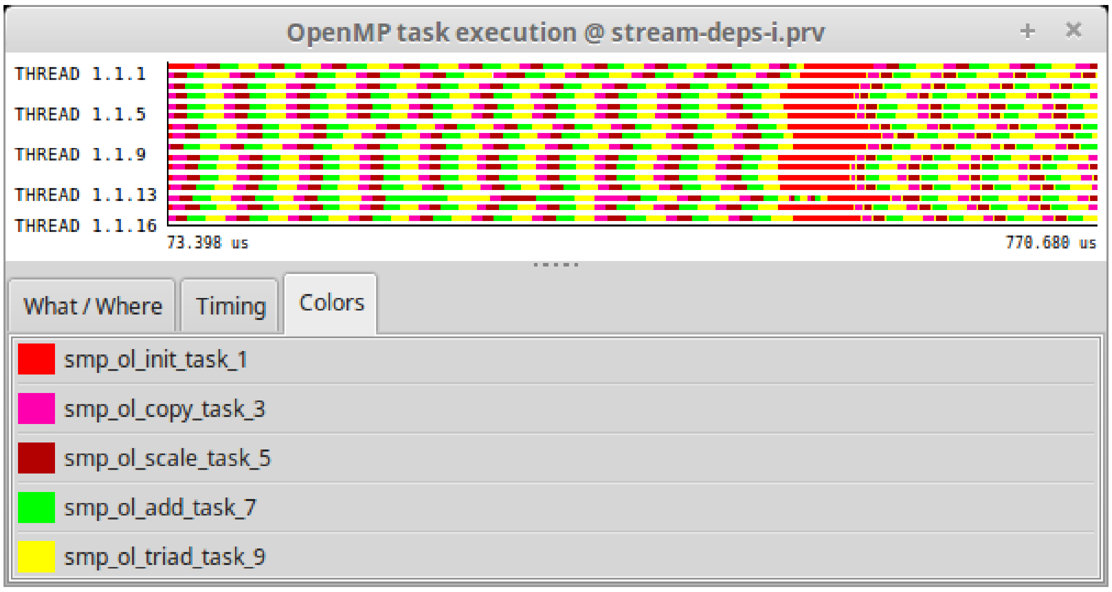
🟥 smp_ol_main_0
🟪 smp_ol_r..jacobi_0 [smp_ol_relax_jacobi_0]
🟥 smp_ol_main_1

Figure 4: Paraver time-line depicting the activity for the Heat test case included in the Extrae distribution package.



🟥 smp_ol_t..p_swap_3 [smp_ol_trsp_swap_3]
🟪 smp_ol_trsp_blk_1
🟥 smp_ol_FFT1D_5
🟩 smp_ol_t..sp_blk_7 [smp_ol_tw_trsp_blk_7]
🟨 smp_ol_t..p_swap_9 [smp_ol_tw_trsp_swap_9]
🟥 smp_ol_FFT1D_11

Figure 5: Paraver time-line displaying the progression through different OmpSs tasks on the FFT Mont-Blanc microbenchmark.



🟥 smp_ol_init_task_1
🟪 smp_ol_copy_task_3
🟥 smp_ol_scale_task_5
🟩 smp_ol_add_task_7
🟨 smp_ol_triad_task_9

Figure 6: Paraver time-line displaying the progression through OmpSs tasks on the Stream benchmark.

2.2.2.1 Modifications in Extrae to track task dependencies

Within this project, OMPT has been extended to track OmpSs (and OpenMP 4.0) dependencies that block the execution of the application. This required further modifications in Extrae. These modifications are mainly divided into two parts: First, Extrae needs handlers to respond to the newly added `ompt_event_dependence()` callback defined in deliverable D5.1, and second Extrae has to generate a pair-wise correspondence between two executing tasks.

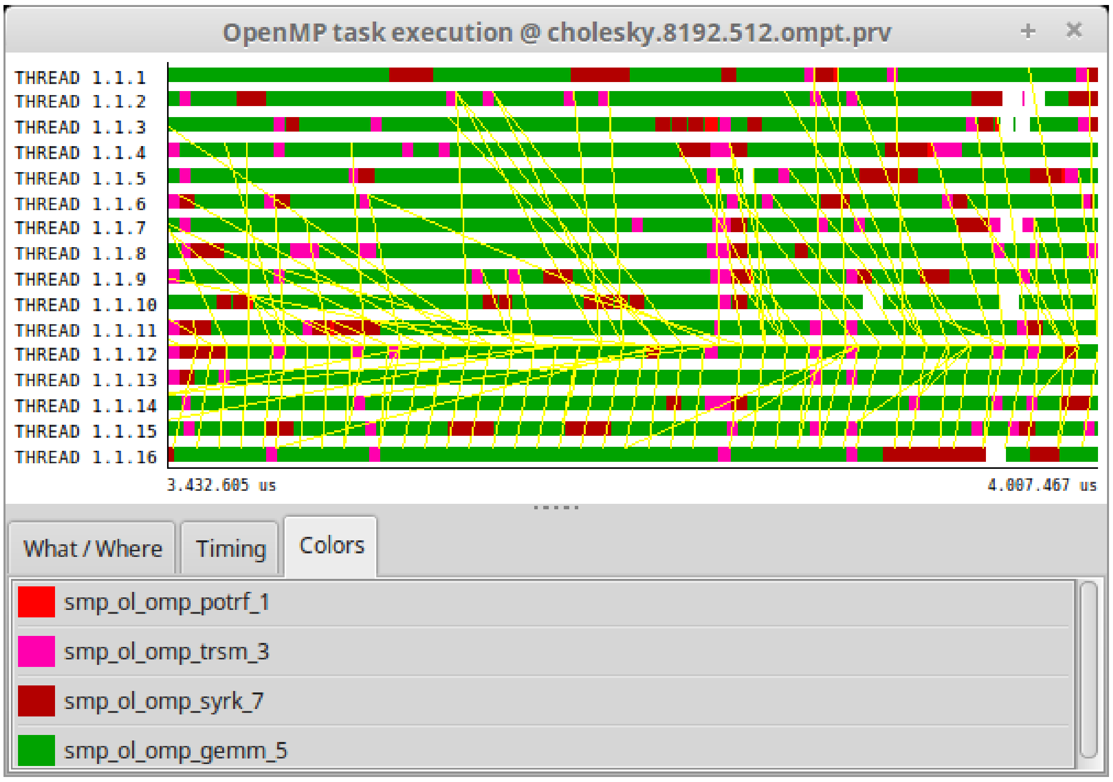
With respect to the first part, the code in the following listing shows the basic capture of the dependence. Similarly to the previously shown code fragments, this routine forwards the captured information to a probe that was added to the OpenMP instrumentation layer. This probe simply generates an event that links the two tasks, the type of the dependence and the data pointer.

```
void OMPT_event_dependence(      /* for new dependence instrumentation */
   ompt_task_id_t pred_task_id, /* ID of predecessor task */
   ompt_task_id_t succ_task_id, /* ID of successor task */
   ompt_dependence_type_t type, /* Type of dependence */
   void *data                   /* Pointer to related data */ )
{
   UNREFERENCED_PARAMETER(data);
   PROTOTYPE_MESSAGE("pred_task_id=%lx, succ_task_id=%lx, type=%d, data=%p",
                     pred_task_id, succ_task_id, type, data);

   Extrae_OMPT_dependence (pred_task_id, succ_task_id, type, data);
}
```

The trace-generation process has also been modified to interpret this newly captured data. It is worth to remember that the Paraver trace format allows establishing a relationship between two threads through communication records. Extrae reuses this record to annotate the relationship between these tasks. In order to track dependencies in Extrae, we added a dependency list that gets updated according to the OMPT events that are processed:

- `ompt_event_dependence()` identifies that a task is being held because another task has to generate its result. Whenever this happens, the trace-generation process allocates a new dependency between the predecessor and the successor tasks.
- `ompt_event_task_end()` when a task finishes, it releases all the tasks that were waiting for it. Whenever this occurs, the process prepares a communication record for every dependence registered that matches the predecessor. These communication records only have information for one partner and must be completed by the corresponding partner when it starts executing.
- `ompt_event_task_switch()` when the task is switched for the first time by the run-time, it means that the thread is ready to run because the dependencies have been satisfied. This step completes the communication record previously initiated by ompt_event_task_end() counterpart for all the tracked dependencies that match the successor.



🟥 smp_ol_omp_potrf_1
🟪 smp_ol_omp_trsm_3
🟥 smp_ol_omp_syrk_7
🟩 smp_ol_omp_gemm_5

Figure 7: Paraver time-line showing the progression of the tasks and the dependencies (shown as yellow lines) that blocked the execution of the tasks that executed on thread 12.

Figure 7 shows a Paraver time-line for an execution of the Cholesky benchmark on the MareNostrum3 system. The figure depicts how tasks are executed with dependencies between tasks represented as yellow lines. In this particular screenshot, we have focused on dependencies that block the execution of the tasks by thread 12. The reader may notice that there is a repetitive pattern of dependencies between threads 12 and 16 in addition to some dependencies with tasks executed in other threads. Paraver allows timeless exploration of the captured results using histograms as depicted in Figure 8. The histogram shows a dependency matrix between threads, thus rows and columns refer to threads and cells indicate the number of dependencies between pairs. The cell coloring uses a gradient that ranges from green (low values) to blue (high values). The blue diagonal in the matrix indicates that most of the threads execute tasks that wait for dependencies generated on the same thread. Additionally, the blueish last row indicates that there are many dependencies between tasks executed in the last thread and other threads, which may be a result of thread-task affinity issues.

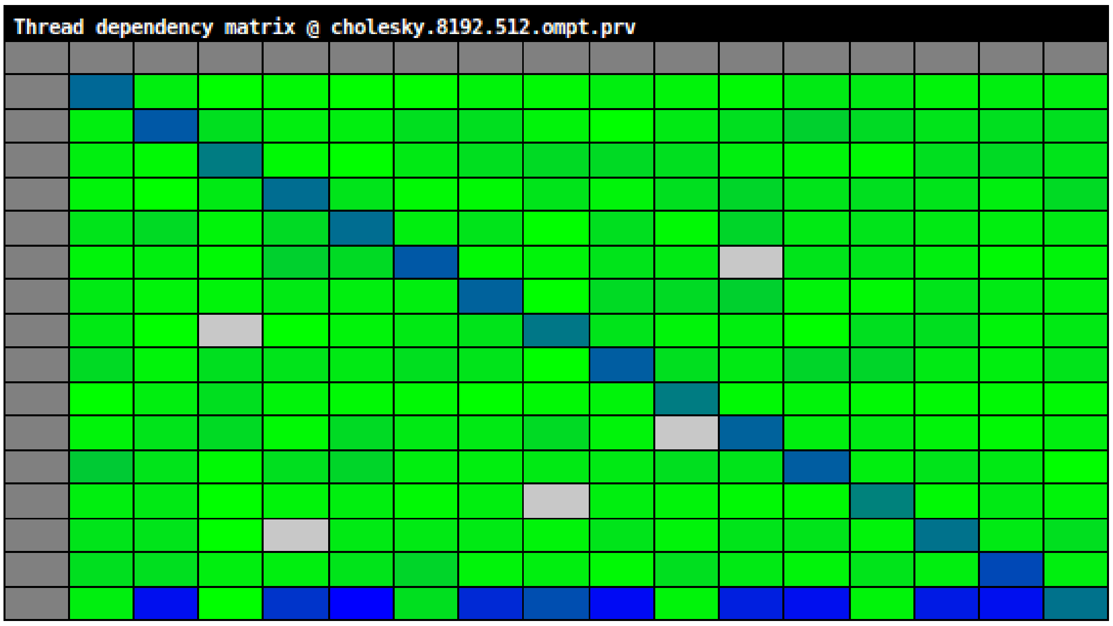


Figure 8: Paraver histogram depicting a thread dependency matrix for the Cholesky benchmark.

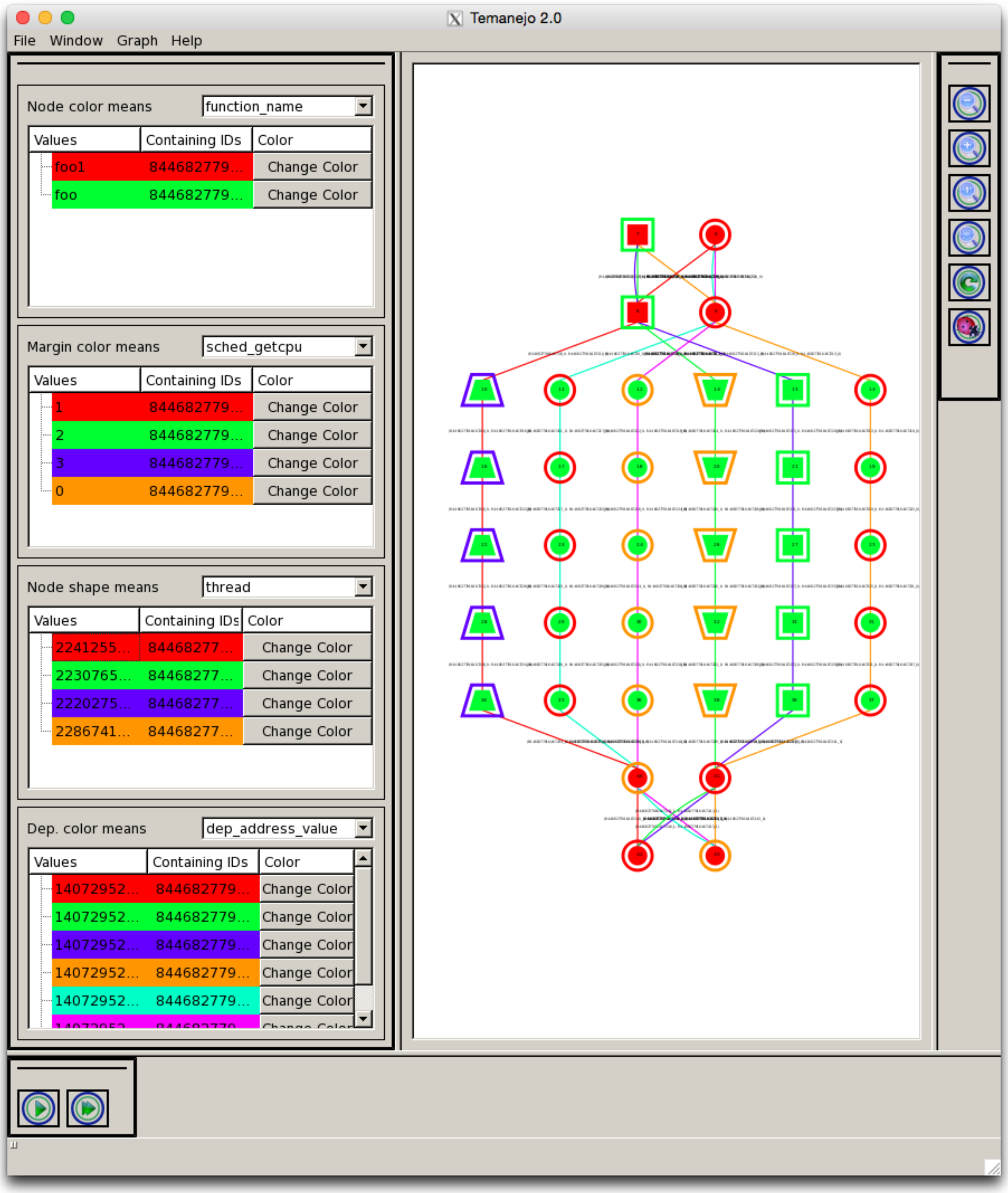
2.3 Ayudame monitoring implementation

A Nanos++ instrumentation plugin has been developed for Temanejo/Ayudame allowing basic monitoring and debugging functionalities. This instrumentation extracts information on tasks (including id, link to source code of tasks, task state, etc.), synchronization primitives (as "taskwait" and "taskwait on"), and dependencies (id, address of corresponding data, etc.). The OmpSs specific dependencies patterns as "commutative" and "concurrent" are also instrumented. The extracted information is translated into Ayudame specific events. In the future, the plans are to implement a fully functional OMPT instrumentation for Ayudame.

In the following example we show how to intercept task creation and to generate the necessary information for Temanejo. The `ayu_event()` call sends the extracted information through Ayudame to Temanejo. The most necessary information's inside the instrumentation are extracted through the `addEventList()` function.

```
void addEventList ( unsigned int count, Event *events ) {
   for (unsigned int i = 0; i < count; i++) {
      switch ( events[i].getType() ) {
         case NANOS_POINT:
            if ( events[i].getKey() == create_wd_ptr ) {
               WD *wd = (WD *) events[i].getValue();

               ayu_event_data_t data;
               data.add_task.task_id = wd->getId();
               ayu_event(AYU_ADDTASK, data);
            }
      }
   }
}
```



Node color means: function_name

Values	Containing IDs	Color
foo1	844682779...	Change Color
foo	844682779...	Change Color

Margin color means: sched_getcpu

Values	Containing IDs	Color
1	844682779...	Change Color
2	844682779...	Change Color
3	844682779...	Change Color
0	844682779...	Change Color

Node shape means: thread

Values	Containing IDs	Color
2241255...	84468277...	Change Color
2230765...	84468277...	Change Color
2220275...	84468277...	Change Color
2286741...	84468277...	Change Color

Dep. color means: dep_address_value

Values	Containing IDs	Color
14072952...	844682779...	Change Color
14072952...	844682779...	Change Color
14072952...	844682779...	Change Color
14072952...	844682779...	Change Color
14072952...	844682779...	Change Color

Figure 9: Dependence graph as show by Temanejo

Figure 9 shows a dependence graph for a synthetic benchmark. Nodes represent tasks and edges represent dependencies. Node color and shape as well as margin and edge colors may change their semantics to represent different information, for example function name, CPU, thread, dependence address, etc.

The following events are generated by the Ayudame OmpSs instrumentation:

- Add Task
- Add Dependency
- Add Property: properties contain all additional information for tasks or dependencies. These could be:

- Function name
- Dependency type (concurrent, commutative)
- Thread
- State (queued, running, finished)

2.4 Score-P performance monitoring interaction

As outlined in Section 1.2.3, every type of instrumentation supported by the Score-P measurement system is handled by an adapter, which maps the specific events triggered by the instrumentation onto more generic function calls of the Score-P measurement core. Thus, we developed a new adapter supporting the OMPT interface specification.

As required by the specification and described in Section 2.1, the Score-P adapter implements the `ompt_initialize()` function which is called by the runtime system during initialization. Here, the adapter queries which call-backs are supported by the runtime and registers appropriate handler functions. These handler functions are then responsible for mapping the events defined by the OMPT interface onto the corresponding measurement functions provided by the Score-P core.

One challenge consists in having one adapter for a specification based on implementations that have slightly different execution models. In this instance, OmpSs and OpenMP can both be classified as fork-join thread models. However OpenMP starting serially creates parallel regions on demand, while the OmpSs threading model can be represented as one parallel region spanning the complete execution of the application. In the context of OMPT, this leads to slightly different call-back requirements when mapping to the Score-P measurement system, in particular with regard to the threading backend. For the OmpSs model, the mandatory events marking begin and end of the threads are sufficient, as the thread team does not change throughout the execution. In the OpenMP case, however, the Score-P implementation relies on the optional events signalling the start and end of a parallel region on each thread to track the corresponding thread teams. Currently, the prototypical implementation in the Intel/LLVM OpenMP runtime provides these events, but they are not required by the specification. Fortunately, the `ompt_initialize()` function provides a name string for the runtime version, which allows to distinguish different runtime implementations and to employ different strategies in the OMPT callbacks.

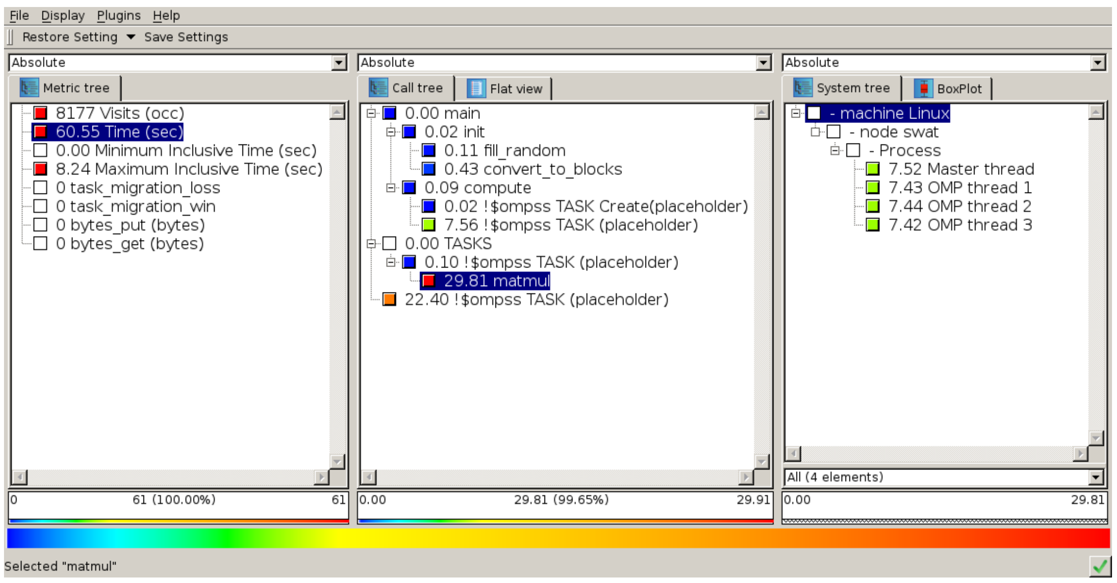
The code example below shows a slightly simplified excerpt from the Score-P OMPT adapter. It demonstrates multiple elements of the general approach to implement the OMPT specification. The basic strategy consists of mapping the interesting OMPT events, here `ompt_thread_begin()` and `ompt_thread_end()`, in registered call-back functions to the fork/join model of Score-P by calling the respective functions of the measurement core. It also shows the use of the runtime identifier supplied to the `ompt_initialize()` function (stored in an internal global variable `scorep_ompt_runtime`) to perform runtime-specific actions – here to track the forking and joining of a single parallel region at program start and end if the OmpSs runtime is being used as mentioned before. In the OpenMP case, these steps are done in the respective `ompt_parallel_begin/end` and `ompt_implicit_task_begin/end` call-backs, as OpenMP creates parallel regions on demand. Note that in the Score-P event model, the FORK and JOIN events are only created on the master thread, while the TEAM_BEGIN and TEAM_END events occur on all threads participating in the created thread team.

```
static void scorep_ompt_thread_begin( ompt_thread_type_t thread_type,
                                      ompt_thread_id_t   thread_id   )
{
  if (scorep_ompt_runtime == OMPT_OMPSS)
  {
    if (thread_type == 1) )   // Master Thread
    {
      SCOREP_ThreadForkJoin_Fork( SCOREP_PARADIGM_OPENMP,
                                  omp_get_num_threads() );
    }
    SCOREP_ThreadForkJoin_TeamBegin( SCOREP_PARADIGM_OPENMP,
                                     thread_id);
  }
}

static void scorep_ompt_thread_end( ompt_thread_type_t thread_type,
                                    ompt_thread_id_t thread_id )
{
  if (scorep_ompt_runtime == OMPT_OMPSS)
  {
    SCOREP_ThreadForkJoin_TeamEnd( SCOREP_PARADIGM_OPENMP );
    if (thread_type == 1)    // Master Thread
    {
      SCOREP_ThreadForkJoin_Join( SCOREP_PARADIGM_OPENMP );
    }
  }
}
```

Tasks are another area where a mapping between the Score-P internal model and the provided set of call-backs takes place. In the OMPT specification, three task-related callbacks are defined: the `task_begin` in the context of the creating thread, and the `task_switch` and `task_end` in the context of the executing thread. The runtime provides the relevant thread identifiers, unique across all threads, to these call-backs. Score-P itself internally uses a thread-local variable to avoid additional locks on every task event. The uniqueness is ensured by generating a 64 bit identifier out of a 32 bit thread ID and a 32 bit thread-local task generation count. For the initial prototype, the match between those task identifier definitions is done by the use of a hash table, accepting the additional lock overhead that is required to avoid data races. More efficient alternatives to this approach are currently under investigation.

Figure 10 shows the profile of a simple task test case using the OMPT prototype adapter. As can be seen, the measurement system employs the generic task model representation where the asynchronous tasks and their respective subtrees are placed within the artificial root node TASKS. Currently, the result still contains placeholders for creation and execution stub nodes, which will be used to identify call sites in later steps of the work. For identification purposes at the task creation point, the OMPT task begin callback provides a stack frame reference of the current location. This can be used to retrieve the source-code location of the task call.


File Display Plugins Help
Restore Setting Save Settings

Absolute | Absolute | Absolute
Metric tree | Call tree | Flat view | System tree | BoxPlot

```
8177 Visits (occ)
60.55 Time (sec)
0.00 Minimum Inclusive Time (sec)
8.24 Maximum Inclusive Time (sec)
0 task_migration_loss
0 task_migration_win
0 bytes_put (bytes)
0 bytes_get (bytes)
```

```
0.00 main
  0.02 init
    0.11 fill_random
    0.43 convert_to_blocks
  0.09 compute
    0.02 !$ompss TASK Create(placeholder)
    7.56 !$ompss TASK (placeholder)
0.00 TASKS
  0.10 !$ompss TASK (placeholder)
    29.81 matmul
22.40 !$ompss TASK (placeholder)
```

```
- machine Linux
  - node swat
    - Process
      7.52 Master thread
      7.43 OMP thread 1
      7.44 OMP thread 2
      7.42 OMP thread 3
```

All (4 elements)

0   61 (100.00%)   61 | 0.00   29.81 (99.65%)   29.91 | 0.00   29.81

Selected "matmul"

Figure 10: Example for the prototype OMPT adapter of Score-P using tasks in a matrix multiplication test case.

The OMPT prototype is still to be considered work in progress and will be extended with additional features and adapted to changes in the specification. The focus of further steps is work on tasks -- and in particular task dependencies -- which are currently ignored. This prototype also doesn't support the target construct for accelerators, as its definition is still under discussion as one of the newest elements of the specification.

3 Debugging Control API

The Tasking Control API (TCA), as defined in deliverable D5.1 has been implemented on both sides (on the runtime and also on the tool side). An OmpSs plugin is created to raise the corresponding events/breaks for a given execution. When an event or break occurs the OmpSs plugin calls the TCA runtime which finally defines all the functionality required by the TCA specification.

As some of the functionality of debugging may cause a change in the behaviour of the program execution, the implementation of several internal services in the Nanos++ RTL was required to allow blocking, stepping or changing task properties.

3.1 *Mercurium/Nanos++ debugging implementation*

3.1.1 Runtime support to allow debug OmpSs programs

The stepping mechanism allows the instrumentation tool to configure a given number of steps before calling a specific routine. In this context, two core services are provided:

```
typedef void (*callback_t)(void);

void setSteps( unsigned short s );
void setCallBack( callback_t cb );
```

After the specified number of steps, the scheduler will call a `callback_t` routine provided as a parameter to the `setCallBack()` routine. Then the instrumentation tool will need to perform the corresponding actions and, before return, set the next steps counter before the next callback.

Nanos++ allows debuggers to change the priority of a given task using the `nanos_wd_t` handler. An instrumentation tool can set or check the priority of any task already created (but not executed). The corresponding services are:

```
void nanos_set_wd_priority(nanos_wd_t wd, int prio);
int  nanos_get_wd_priority(nanos_wd_t wd);
```

3.1.2 Improved debug information for C/C++ programs

Due to the source-to-source nature of Mercurium, the executable program is not generated directly from the source code of the application. Instead, it comes from the code emitted by Mercurium after it has applied the required transformations to implement OmpSs.

While this compilation strategy works well it can have some detrimental effects when debugging the code: the source code shown in the debugger is not the same as the code the user has written, but the one generated by the Mercurium compiler. While this is clearly a problem for a human user, some analysis and debugging tools do not work well in this scenario either.

Obviously this problem is not specific to source-to-source compilers; any tool that generates source code from some other input will have this mismatch (i.e. parser generators like yacc/bison).

C (C11 §6.10.4) and C++ (C++11 §16.4) provide a standard mechanism of line control in the form of #line directives. Since Mercurium keeps track of the location of the input code, it can emit the corresponding directives. However, since Fortran does not have a pre-processor, #line directives cannot be used, and so this feature is not available for Fortran programs.

After all the analysis and transformation phases of Mercurium have run, a code generation phase is executed. This phase is responsible for emitting the Mercurium intermediate representation of the program into C/C++ source-code. We extended this code generation phase in the compiler to emit #line directives. This change did not require changes in the runtime. We verified that the debugging experience is much better.

Line markers are not enabled by default, they have to be requested using the Mercurium command parameter `--line-markers'.

3.2 Ayudame/Temanejo debug interaction

For OmpSs and Ayudame the runtime-side implementation is done inside the Ayudame instrumentation plugin. As TCA is designed as an extension of OMPT, this code could later become part of the OMPT instrumentation plugin.

3.2.1 Instrumentation plugin

On the runtime/instrumentation-side we have to include the TCA runtime to get the necessary functionality. Inside the instrumentation constructor, the `tca_initialize()` function is called. This function is an extern “C” function, implemented on the tool side (Ayudame)

```
#include "tca_runtime.h

extern "C"
{
   void tca_initialize( tca_function_lookup_t lookup,
                        const char *runtime_version,
                        int tca_version )  attribute__ ((weak));
}

InstrumentationAyudame()
{
   tca_initialize(lookup, "Nanos++", TCA_VERSION);
}
```

3.2.2 TCA runtime

The TCA runtime implements the API specified in deliverable D5.1 and also the lookup function called by the tool side (Ayudame). The lookup function returns a function pointer for the requested API call. In this section we only show the implementation of the `tca_request_step()` routine, but we also provide a list of all the services that have been already implemented in this additional runtime.

```
tca_success_t tca_request_step()
{
   increase_step_counter();
   return TCA_OK;
}

tca_interface_fn_t lookup(const char *entry_point)
{
   if (0 == strncmp(entry_point, "tca_request_step")) {
      return (tca_interface_fn_t) &tca_request_step;
   }
}
```

The following requests are implemented in the TCA runtime and have effects on the runtime:

- tca_request_continue
- tca_request_break
- tca_request_step
- tca_request_block_task
- tca_request_unblock_task

The following requests are implemented in the TCA runtime but will be ignored by the runtime:

- tca_request_continue_thread

- tca_request_break_thread
- tca_request_shutdown
- tca_request_run_task
- tca_request_insert_dependency
- tca_request_remove_dependency
- tca_request_break_at_task
- tca_request_unbreak_at_task
- tca_request_break_at_dependency
- tca_request_unbreak_at_dependency
- tca_request_step_thread
- tca_request_set_priority

3.2.3 Tools side (Ayudame)

For OmpSs and Ayudame the tool-side implementation is done inside Ayudame. The Ayudame instrumentation plugin will call the `tca_initialize()` function during its construction. Ayudame will then request the functionality Ayudame wants to use though the lookup function. The function pointers returned by the lookup function are stored inside Ayudame. In our example the `req_step()` function pointer is used to increase the step counter.

```
tca_request_step req_step;

void tca_initialize(tca_function_lookup_t lookup,
                    const char  *runtime_version,
                    unsigned int tca_version)
{
   req_step = (tca_request_step) lookup("tca_request_step");
}

req_step();
```

4 Conclusions and Future Work

This deliverable is the result of a close collaboration between the HLRS, Jülich and BSC institutions and presents the implementation experience of including performance monitoring and debugging support in the OmpSs programming model and the tool set participating in the project. Performance monitoring tools capture the events generated by the OmpSs runtime in order to describe the behaviour for a given execution. The OmpSs runtime also implements additional services which allow the modification of the current execution status for debugging purposes. These changes allow a better analysis of the user code.

As the present work is partially based on a Technical Report published by the OpenMP ARB we have sent and received feedback from them. As part of the feedback we sent to them, we have to specifically point out the task dependence extension, which has been proposed to the OpenMP Tools Subcommittee. This group has recently also worked on a task dependence extension with a different approach as the one presented in deliverable D5.1. As part of the future work derived from this implementation we plan to implement and compare both approaches.

In terms of accelerator monitoring, OmpSs will implement the instrumentation of the accelerator-side when the currently ongoing discussions in the OMPT working group on capturing kernel executions on accelerators have settled. Despite the fact that OmpSs does not match directly on top of OpenMP in terms of accelerators, we believe that OmpSs could reuse the work specified by the OMPT working group and therefore still comply with the OMPT specification.

With respect to the Tasking Control API, there are some missing features that we plan to implement in the near future. The list of non-supported runtime control services is shown in Section 3.2.2 and more discussion will be needed to determine which of these services will allow a better analysis of OmpSs applications. We also plan to merge the performance monitoring (OMPT) and debug (TCA) plugins into a single combined plugin; first offering the capability to generate runtime events to describe program execution, and also allowing to modify the program execution *Just In Time*.

Acronyms and Abbreviations

- **API** Application Programming Interface
- **Extrae** Parallel program instrumentation and measurement package of BSC
- **Nanos++** Task-oriented runtime system of BSC (used for OmpSs)
- **OpenMP** Industry standard for pragma-based parallel programming paradigm for shared memory computers
- **OmpSs** OpenMP extension (including task dependencies and accelerator support) of BSC
- **OMPT** Draft standard OpenMP performance Tools interface, OpenMP ARB
- **Paraver** Trace visualizer of BSC
- **Scalasca** Parallel performance analyzer of JUELICH
- **Score-P** Parallel program instrumentation and measurement package of JUELICH, TU Dresden and other partners
- **TCA** Task Control Interface, proposed by HLRS, BSC and Allinea
- **Temanejo** Task-based debugger of HLRS

References

[1] Alexandre Eichenberger, John Mellor Crummey, Martin Schulz, Nawal Copty, John Del Signore, Robert Dietrich, Xu Liu, Eugene Loh, Daniel Lorenz and other members of the OpenMP Tools Working Group. OMPT and OMPD: OpenMP Tools Application Programming Interfaces for Performance Analysis and Debugging. *Available at http://openmp.org/mp-documents/ompt-tr.pdf*, April 2013.

[2] Alexandre Eichenberger, John Mellor Crummey, Martin Schulz, Nawal Copty, Jim Cownie, Robert Dietrich, Xu Liu, Eugene Loh, Daniel Lorenz and other members of the OpenMP Tools Working Group. OpenMP Technical Report 2 on the OMPT Interface - OMPT: An OpenMP Tools Application Programming Interface for Performance. *Available at http://openmp.org/mp-documents/ompt-tr2.pdf*, March 2014.

[3] Alexandre E. Eichenberger, John Mellor-Crummey, Martin Schulz, Michael Wong, Nawal Copty, Robert Dietrich, Xu Liu, Eugene Loh, Daniel Lorenz. OMPT: An OpenMP Tools Application Programming Interface for Performance Analysis. In *OpenMP in the Era of Low Power Devices and Accelerators, 9th International Workshop on OpenMP, IWOMP 2013*, Canberra, ACT, Australia, September 16-18, 2013, pp 171-185, DOI 10.1007/978-3-642-40698-0_13.

[4] MareNostrum3 System Architecture *Available at http://www.bsc.es/marenostrum-support-services/mn3*