Performance Measurement for the OpenMP 4.0 Offloading Model

Robert Dietrich[1], Felix Schmitt[1], Alexander Grund[1], and Dirk Schmidl[2]

[1] Center for Information Services and High Performance Computing, Technische Universität Dresden, 01062 Dresden, Germany
{robert.dietrich,felix.schmitt}@tu-dresden.de, alexander.grund@mailbox.tu-dresden.de
[2] IT Center, RWTH Aachen University, 52056 Aachen, Germany
schmidl@itc.rwth-aachen.de

Abstract. OpenMP is one of the most widely used standards for enabling thread-level parallelism in high performance computing codes. The recently released version 4.0 of the specification introduces directives that enable application developers to offload portions of the computation to massively-parallel target devices. However, to efficiently utilize these devices, sophisticated performance analysis tools are required. The emerging OpenMP Tools Interface (OMPT) aids the development of portable tools, but currently lacks the support for OpenMP 4.0 target directives. This paper presents a novel approach to measure the performance of applications utilizing OpenMP offloading. It introduces *libmpti*, an OMPT-based measurement library for Intel MIC target devices. For host-side analysis we extended the OPARI2 instrumenter and prototypically integrated the complete approach into the state-of-the-art tool infrastructure Score-P. We demonstrate the effectiveness of the presented method and implementation with a Conjugate-Gradient (CG) kernel on an Intel Xeon Phi coprocessor. Finally, we visualize the obtained performance data with Vampir.

Keywords: performance analysis, offloading, OpenMP 4.0, Intel MIC, Score-P.

1 Introduction

The directive-based programming model OpenMP is a popular way to develop multi-threaded applications. Version 4.0 [9] of the specification introduces directives for computation offloading; thus, taking the availability of accelerator hardware in recent computing systems and processors into account. Although OpenMP 4.0 provides an interface for programming of heterogeneous hardware, it does not ensure that the available resources are efficiently exploited, e.g. load-balancing is getting more tedious. To identify and resolve new potential inefficiencies performance tools are challenged to support the offloading directives.

For the simple reason that OpenMP does not provide a standardized performance monitoring interface yet, several individual analysis approaches have emerged. Depending on the approach they come with inherent limitations and advantages. Although it is not yet part of the specification, the OpenMP Architecture Review Board released

L. Lopes et al. (Eds.): Euro-Par 2014 Workshops, Part II, LNCS 8806, pp. 291–301, 2014.

the OpenMP Tools Interface (OMPT) as a technical report [2]. OMPT specifies an application programming interface (API) that enables tool builders to develop portable libraries for performance monitoring. First implementations of OMPT are available for open-source OpenMP runtimes. However, OMPT is built on version 3.1 of the OpenMP specification and lacks support for offloading directives.

This work presents a portable method to measure the performance of OpenMP 4.0 computation offloading. We further contribute with a prototypic implementation into the measurement infrastructure Score-P. With regard to the OpenMP standard we build our approach upon OMPT. As it is already a part of the Score-P infrastructure we use the source-to-source instrumenter OPARI2 to implement features that are missing in OMPT. We evaluate the proposed methods with an OMPT implementation in the open-source version of the Intel OpenMP runtime [1] on the Intel Many Integrated Core (MIC) architecture using a Conjugate-Gradient kernel.

The remainder of this paper is organized as follows: Section 2 presents related work in the area of OpenMP performance analysis. In section 3 we depict the two OpenMP instrumentation approaches this work is based on. Our contribution is discussed in section 4 and the integration into the Score-P infrastructure in section 5. We demonstrate the practicality of the proposed method by applying it to a use case in section 6. Finally, we conclude this paper and outline future work in section 7.

2 Related Work

Score-P [6] is a unified performance measurement infrastructure for several tools like Vampir [4] and Scalasca [3]. It supports different programming models such as MPI, OpenMP and CUDA and it allows to generate profiles in CUBE format as well as traces in OTF2 format. Considering OpenMP measurement, Score-P uses OPARI2 to instrument application code and implements the POMP2 interface.

Scalasca [3] is a scalable performance analysis toolset which can handle Score-P generated profiles and traces. It supports the analysis of hybrid MPI+OpenMP applications on the Intel Xeon Phi coprocessor using Intel's symmetric execution model for MIC [10], i.e. MPI communication is used between host and coprocessor. Our work is different as we focus on the asymmetric offloading model based on OpenMP 4.0. Nevertheless, our approach can be used for hybrid MPI+OpenMP programs on multiple hosts and coprocessors.

Vampir [4] is a scalable visualization tool for OTF and OTF2 trace files. It consists of a client front end and a parallel server back end. Information is visualized in various displays, including an event timeline, function and message statistics and call stacks. The integration of our measurement approach into Score-P allows us to use Vampir to display performance data for offloaded regions.

HPCToolkit [5] is a set of sampling-based tools for measuring, evaluating and visualizing performance data for MPI, OpenMP and CUDA applications. Considering OpenMP, HPCToolkit uses the OMPT interface to query state information for OpenMP threads. States can include for example if a thread is currently executing a parallel region, it is idle or waiting on another. As OMPT currently does not provide support for OpenMP target devices HPCToolkit cannot gather respective state information.

The CUDA Profiling Tools Interface (CUPTI) [8] is a proprietary tools interface by NVIDIA for their CUDA architecture, which is used by many tools like e.g. Vampir-Trace and Score-P. It provides an API that allows tools to register for event callbacks, measure performance counters, metrics and activity records. Since the CUPTI library resides in the host address space, the tool is not required to transfer performance information from the target device explicitly.

3 OpenMP Instrumentation

The OpenMP specification does not define a performance monitoring interface yet, but a technical report (TR) which covers such an interface has been released by the OpenMP Architecture Review Board. As upcoming OpenMP specifications will eventually include this TR we base our work on it. However, OpenMP 4.0 offloading directives are neither defined in the current proposal nor is their implementation available in publicly accessible OpenMP runtimes. To instrument OpenMP offloading anyway we prototypically extended the instrumenter OPARI2.

3.1 OMPT

OMPT [2] addresses two strategies for performance data collection: asynchronous sampling and event-based monitoring. For tools that employ asynchronous sampling OMPT provides routines to query information about the state of each OpenMP thread. States are classified to be either mandatory, optional or flexible. In contrast to mandatory states, an OpenMP implementation does not need to maintain optional states. Aside from that it has some freedom when reporting a transition to a flexible state.

Event-based tools, like e.g. Score-P, can register function callbacks for particular events that are triggered by the OpenMP runtime system. OMPT provides begin and end events for most OpenMP constructs. However, the set of mandatory events is small and allows tools to collect only basic information about the runtime behavior of OpenMP programs. To gather more performance-critical information tools have to register for optional events that might not be available for a given OpenMP runtime system.

OMPT is intended to be implemented by a compiler, an OpenMP runtime system or a mixture of both. Therefore the interface defines function pointer addresses for outlined functions of parallel regions and tasks as the only meta information on constructs. The function pointers can be used to distinguish OpenMP constructs of the same type, respectively identify regions of the same construct and to obtain source-code information if available.

3.2 OPARI2

OPARI2 is the current version of the source-to-source instrumentation tool OPARI (OpenMP Pragma and Region Instrumenter) [7], which inserts calls to the POMP2 monitoring interface at OpenMP pragmas and library calls. Similar to the events defined in the OMPT interface the POMP2 event model provides events for the begin and

the end of an OpenMP construct, enabling tools to gather performance information for OpenMP programs.

As the OPARI2 instrumentation modifies the source code directly it is independent of a specific OpenMP implementation but requires recompilation of the application. Additionally, OPARI2 creates *POMP2 region handles* for OpenMP constructs, e.g. *parallel regions* and *tasks*, which include source information such as the file and line number of the construct. Tools can utilize these handles to correlate performance information directly with the source code, thereby aiding developers to easily identify performance-critical code.

4 Measuring the OpenMP 4.0 Offloading Model

The OpenMP 4.0 specification introduces several new directives. This work focuses on the measurement of the offloading model. We use the terms *host device* and *target device* according to the specification. The *host device* is the system *from which* code within an OpenMP *target* construct is Since applications are started from the *host device*, this is furthermore where the measurement environment executes. A *target device* describes an accelerator or coprocessor *to which* the mentioned *target* region is offloaded. Regularly, *host device* and *target device* do not share a common address space, which must be taken into account when designing adequate tool support.

4.1 OpenMP 4.0 Target Directives

OpenMP 4.0 introduces the target directives to enable computation offloading. Encountering a *target* construct implicitly creates a device data environment and the subsequent statement, loop or structured block is executed on the target device. The *target data* construct explicitly creates a device data environment which can be used to avoid implicit data transfers between host and target device for enclosed target regions.

When a *map* clause is present for a *target* or *target data* construct and the data have not been mapped in a surrounding data environment before, they are mapped explicitly, according to the specified variables and map-types, at the beginning and end of the block. Map-types are *alloc*, *to*, *from*, and *tofrom*. Depending on the hardware configuration and the OpenMP runtime implementation, the mapping invokes a data transfer. Variables not declared but referenced in a *target* construct are treated as if they appeared in a *map* clause with a map-type *tofrom*, thus, they are implicitly transferred to and from the target device. The *target update* construct is a stand-alone directive and makes data on the host and target device consistent, according to the variables specified in the *motion-clause*. Motion-clauses are *to* and *from* and update data on the target or on the host, respectively. If a *device* clause is present in a target directive, it specifies the target device. Otherwise the default device is used. When an *if* clause is present and its expression evaluates to false the target directive does not take effect, as data are not mapped nor is the execution offloaded to the target device.

There are other new directives in OpenMP 4.0 that might influence the execution efficiency, such as e.g. the *teams*, the *distribute* and the *simd* directive. However, we do not observe them in terms of performance measurement within the scope of this paper.

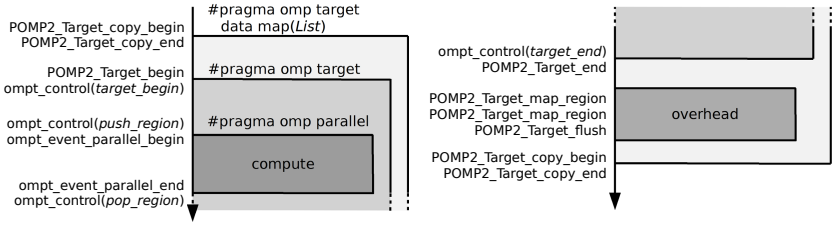

Fig. 1. Execution sequence of measurement routines for an OpenMP parallel region enclosed in a target region. A separate device data region has been added to measure the explicit mapping of variables. OpenMP constructs are instrumented with POMP2 and OMPT calls.

4.2 Measurement Approach

The measurement of OpenMP target regions and other enclosed regions can be achieved with a combination of OPARI2 instrumentation and OMPT callbacks. Figure 1 shows the execution sequence of measurement routines for a simple program with a *parallel* construct enclosed by a *target* construct. To record the runtime of a *target* region, timing routines are added before the directive and after the associated code (statement, loop or structured block). For *target data* regions a time stamp is recorded before the respective begin directive and at the beginning of the associated code as well as at the end of the associated code and after the *target data* region. As *target data* regions are executed on the host device, the deployed measurement environment can directly record the resulting data transfer times. If data transfers are only asynchronously invoked (e.g. for GPGPUs), this approach does not measure the effective mapping. However, these types of data transfers can be measured by other means (e.g. libcupti for CUDA devices).

Measuring data transfers for a *target* construct with a *map* clause is a bit more tedious. To obtain the transfer time for explicit data transfers specified in the *map* clause, we move the respective clause to a newly generated *target data* construct enclosing the original *target* construct. However, there might occur implicit data transfers that are invoked simply by referencing variables that are not declared for the target device. This implicit data mapping can be recorded as part of the *target* region overhead, which is measured by calling a timing routine before the execution of the respective directive and at the beginning of the associated code as well as at the end of the associated code and after the *target* region.

To record the execution of OpenMP constructs that are enclosed in a *target* region we register callbacks for OMPT events. We record events on the master thread of a thread team executing a parallel region and each explicit task. Additionally, we measure some optional events when available, such as barriers. Source-code correlation for these events is added by inserting *ompt_control* calls passing a region handle that is later mapped to the corresponding statically created POMP2 region handle. After the target region finishes execution, the target device buffer is flushed (*POMP2_Target_flush*). This introduces most of the measurement overhead, but only at synchronous points in the program execution. Furthermore, we insert synchronization points before and after a target region that are necessary to convert the target device time stamps to host device time stamps.

4.3 Extending OMPT with Support for Target Directives

It is possible to measure the offloaded computation using OMPT without prior instrumentation. However, this would remove the possibility to correlate performance data with the source-code location from where it originates. Even though OMPT's *outlined function pointer* enables the tool to identify the calling function, the Intel MIC software stack for example does not provide any tools that are required to evaluate a backtrace to identify source file and line for a memory address.

Without instrumentation, performance tools that use OMPT must be able to register callbacks for *synchronization points* at which performance data can safely be collected from the target device. Such synchronization points could be the begin and end of a target region. Similar callbacks would be beneficial to measure the data transfers induced by *target update* directives. Note that none of these are yet available but they are likely to be added to a future version of OMPT. Within the callback, the host tool could notify the target device to flush its buffers and transfer the collected records to the host.

5 Integration into Score-P and OPARI2

5.1 Measurement Control Flow

We integrated our approach into the measurement infrastructure Score-P since it already supports OpenMP performance analysis using OPARI2. When the Score-P compiler wrapper detects OpenMP code, it invokes the OPARI2 instrumenter which has been extended to enable the instrumentation of new OpenMP 4.0 directives. At application start a small measurement library called *libmpti* that implements the OMPT interface is preloaded on the target device. This library is responsible for capturing performance-related events on the target device using OMPT. Once control is returned to the host device after the *target* region has been executed, performance records are transferred from the target to the host device by Score-P. The complete control flow is illustrated in figure 2.

5.2 Extensions to the POMP2 Interface

To properly support the measurement of OpenMP 4.0 *target* constructs, we added six functions to the POMP2 interface (cf. figure 1). *POMP2_Target_begin/end* are inserted before and after the *target* construct in order to perform host/target time synchronization and setup appropriate data structures. Several calls to *POMP2_Target_map_region* are inserted after the *target* construct to map runtime identifiers for OpenMP constructs, which are integral numbers, to their corresponding POMP2 region handles. The latter contain source code information but cannot be used directly on the target device because the OMPT interface allows to pass only values of type *uint64_t* to *ompt_control*. *POMP2_Target_flush* is called directly before *POMP2_Target_end* and initiates the transfer of target device records from *libmpti* to Score-P on the host device. Finally, *POMP2_Target_copy_begin/end* calls are added by OPARI2 around both begin and end of a *target data* directive to measure the execution time of data transfers.

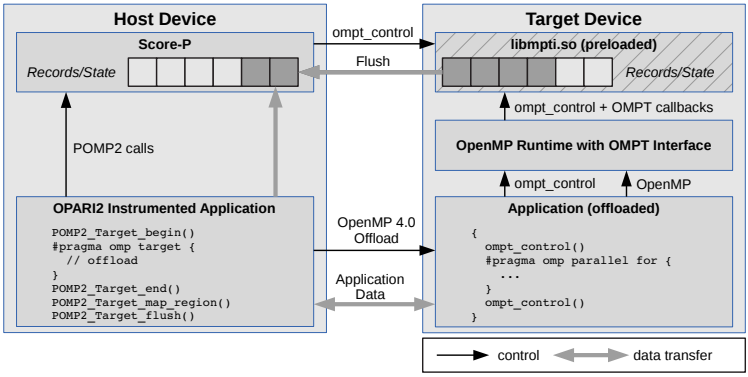

Fig. 2. Control and Data Flow: Host OpenMP activities are captured by Score-P using the POMP2 interface. Performance records for offloaded application parts are measured using *libmpti*, which implements OMPT on the target device, and transferred to the host during *POMP2_Target_flush*. OpenMP constructs enclosed in a *target* construct are instrumented with *ompt_control* calls to track their execution. The hatched area is the only platform-dependent part.

5.3 MIC Performance Tools Interface

The **M**IC **P**erformance **T**ools **I**nterface (*libmpti*) is a small library which implements the OMPT interface for the Intel MIC architecture and is compiled as native MIC code (i.e. using the *-mmic* compiler argument). Since the OMPT-instrumented open-source version of the Intel OpenMP runtime cannot be compiled as a fat binary for the host and the MIC, the library cannot be linked against the created executable directly. Instead, *libmpti* is preloaded at application start using Unix' *LD_PRELOAD* mechanism. With regard to Score-P, *libmpti* is not used directly in offloaded measurement code but only by means of the portable OMPT control mechanism.

Registering for OMPT events, the library can record execution times for parallel regions and explicit tasks on the target device. Per-thread data is available via OMPT but not recorded as this would incur significant runtime overhead. Furthermore, *libmpti* tracks the current state of each thread executing on the target device. This state includes only a stack of identifiers designating the currently executed construct. A region identifier is pushed on this stack using *ompt_control* calls inserted via OPARI2 instrumentation directly before an OpenMP *parallel* or *task* construct. Similarly, the current identifier is popped from the stack after the respective construct has been left. This allows *libmpti* to correlate generic region identifiers with target device records. When Score-P receives those records from *libmpti*, it can map them to POMP2 region handles which include information such as the source file and line number on the host device.

5.4 Visualization

Integration of our offloading measurement approach into Score-P allows developers to take advantage of its OTF2 trace output. Resulting traces can be readily visualized in the Vampir trace viewer. For offloading records created using *libmpti*, Score-P can internally utilize the same mechanisms and data structures as for traditional OpenMP performance data, resulting in a homogeneous Vampir experience for the user.

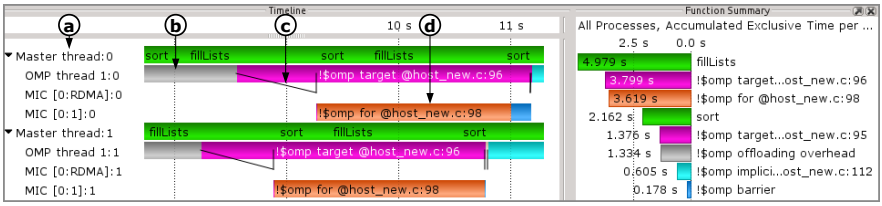

Fig. 3. Visualization: Two MPI processes using OpenMP 4.0 *task* and *target* directives for heterogeneous computation: (a) naming of new MIC offloading locations (b) target device initialization overhead (c) explicit and implicit data transfers between host and target visualized as RDMA messages (d) parallel region (*parallel for*) and implicit barrier on target device

Vampir allows to display hierarchies of processes and threads (*locations*) to present OpenMP threads as children of the spawning process. Therefore, we create new child locations for offloaded OpenMP, too. We add a new target location for each explicit task and each master thread in a thread team executing a parallel region. This also includes measurement and visualization of nested parallelism. As a result, we create similar experience to the visualization of CUDA kernels with dynamic parallelism. As threads on an OpenMP target device are similar to locations (*streams*) on different CUDA devices, we furthermore create a resembling naming for those executing on the MIC architecture (see Figure 3 (a)).

Differences between the visualization of traditional and offloaded OpenMP code are primarily in data transfers. OpenMP 4.0 uses both implicit and explicit data transfers between host and target. In the case of Intel's Xeon Phi coprocessor device, the connection is realized using the PCI-Express interface and data transfers over this interface can suffer from both latency and bandwidth restrictions. Hence, developers must be able to identify OpenMP constructs that result in poor application performance due to such transfers.

For data transfers, one RDMA location per target device is added. On this location, we use the same visualization for both explicit and implicit transfers (see Figure 3 (c)). All transfers are marked as RDMA messages from the spawning host thread to the target's RDMA location. Note that for implicit transfers, this also includes the launch overhead for the target region as the two cannot be distinguished using OpenMP means.

6 Experiments on Intel Xeon Phi

We use an implementation of the sparse Conjugate-Gradient (CG) method to evaluate our measurement approach. The algorithm consists of a matrix-vector multiplication and some vector operations which have been offloaded to the target device. Additionally, a *target data* construct was added to keep all vectors and the matrix on the target until the computation has finished.

Figure 4 compares the visualization of the performance data with the same kernel parallelized using OpenMP on an Intel Xeon Phi coprocessor and OpenACC on a NVIDIA C2050 Fermi GPU. In both cases the target is shown as a separate location in the timeline view where offloaded kernels are illustrated as activities on the target location. This makes it easy for programmers already familiar with OpenACC and Vampir

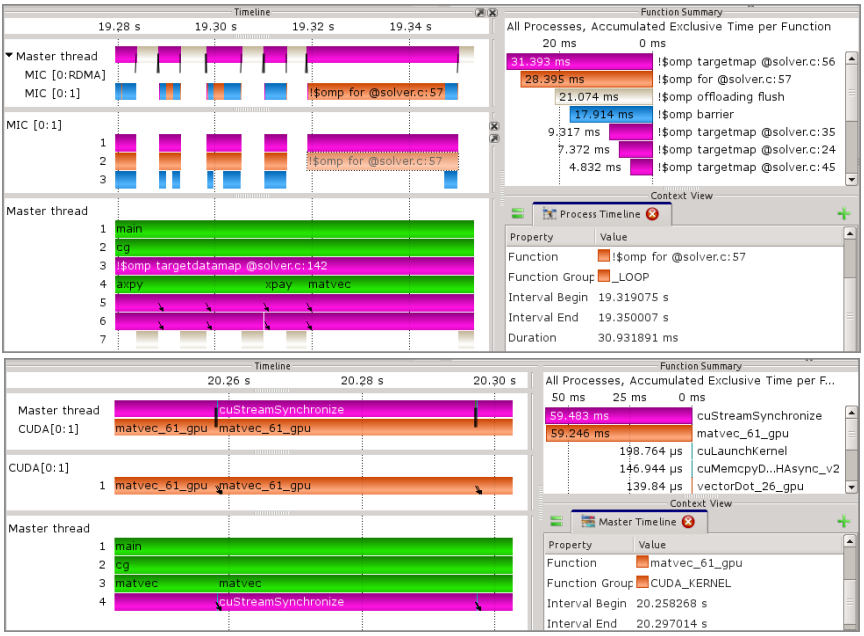

Fig. 4. Performance results for the CG method visualized in Vampir for OpenMP 4.0 (top screenshot) and OpenACC (bottom screenshot). Both show a timeline view (top-left), call stack views for the target/device (middle-left) and the host (bottom-left), a function summary (top-right) and the context view (bottom-right).

to navigate through the OpenMP target device activities. An advantage of *libmpti* is that more detailed information is collected for the target device. As shown in the process timelines in figure 4 (top), the call stack presents information on different OpenMP regions, e.g. synchronization in barriers on the target and their nesting.

To investigate the measurement overhead on our test system, which is equipped with two Intel Knights Corner devices with B0 stepping, 61 cores at 1090 MHz and 8 GB GDDR5 at 5.5 GT/s, we compare the instrumented with the original code version. We repeated each test 50 times and used 120 threads on the coprocessor. The average time for the computational part of the solver was 5.21 sec in the original case and 5.82 sec including performance measurement. The overhead for the measurements was about 12 %, which mainly stems from flushing the target device buffer at a synchronous point in the program execution. It depends only on a fixed offloading latency and the number of target device records to be transferred.

Intel allows to gather basic performance information for the offloaded regions by setting the environment variable *OFFLOAD_REPORT*. This results in a text report with information about all regions. For our CG solver the report contained 600 entries, 598 compute regions and one entry for the enter and exit of the data region. In the trace we also observed 598 compute regions on the target device and the data region was shown as a separate region in the call stack of the master process. The accumulated time of all regions in the measured trace was 5.82 sec which exactly matches the average compute time for the kernel in the instrumented case.

7 Conclusion and Future Work

This work presents a portable approach to obtain performance relevant information on programs utilizing the new OpenMP 4.0 target directives. For performance measurement on the target device we rely on OMPT. OPARI2 is used to instrument the *target* constructs and to add source-code correlation. We show where instrumentation hooks have to be added to measure explicit and implicit data transfers between host and target device as well as the runtime and the overhead for the execution of a *target* construct.

We developed the measurement library *libmpti* to record the execution of OpenMP constructs on Intel MIC target devices. For another target device, *libmpti* needs to be replaced with a platform-specific implementation. approach and allow a visual analysis of the performance data we extended the popular measurement infrastructure Score-P. In a use case we compared the obtained information with an OpenACC version of the same CG kernel. To extend our implementation, we plan to add instrumentation of the *target update* directive and record respective data transfers.

Acknowledgements. Parts of this work were funded by the German Federal Ministry of Research and Education (BMBF) under Grant Numbers 01IH11006(LMAC) and 01IH13008(ELP).

References

1. Mellor-Crummey, J., et al.: OMPT support branch of the open source Intel OpenMP runtime library (December 2013), `http://intel-openmp-rtl.googlecode.com/svn/branches/ompt-support`
2. Eichenberger, A., Mellor-Crummey, J., Schulz, M., Copty, N., Cownie, J., Dietrich, R., Liu, X., Loh, E., Lorenz, D.: OpenMP Technical Report 2 on the OMPT Interface (March 2014)
3. Geimer, M., Wolf, F., Wylie, B.J.N., Erika Abraham, D.B., Mohr, B.: The Scalasca performance toolset architecture. Concurrency and Computation: Practice and Experience 22(6), 702–719 (2010)
4. Knüpfer, A., Brunst, H., Doleschal, J., Jurenz, M., Lieber, M., Mickler, H., Müller, M.S., Nagel, W.E.: The Vampir Performance Analysis Tool-Set. In: Resch, M., Keller, R., Himmler, V., Krammer, B., Schulz, A. (eds.) "Tools for High Performance Computing", Proceedings of the 2nd International Workshop on Parallel Tools for High Performance Computing. Springer, Stuttgart (2008)
5. Liu, X., Mellor-Crummey, J., Fagan, M.: A new approach for performance analysis of OpenMP programs. In: Proceedings of the 27th International ACM Conference on International Conference on Supercomputing, pp. 69–80. ACM (2013)
6. Mey, D., Biersdorf, S., Bischof, C., Diethelm, K., Eschweiler, D., Gerndt, M., Knüpfer, A., Lorenz, D., Malony, A., Nagel, W.E., Oleynik, Y., Rössel, C., Saviankou, P., Schmidl, D., Shende, S., Wagner, M., Wesarg, B., Wolf, F.: Score-P: A Unified Performance Measurement System for Petascale Applications. In: Bischof, C., Hegering, H.G., Nagel, W.E., Wittum, G. (eds.) Competence in High Performance Computing 2010, pp. 85–97. Springer (2012)
7. Mohr, B., Malony, A.D., Shende, S., Wolf, F.: Design and Prototype of a Performance Tool Interface for OpenMP. The Journal of Supercomputing 23(1), 105–128 (2002)

8. NVIDIA: CUDA Toolkit Documentation — CUPTI (July 2013), http://docs.nvidia.com/cuda/cupti/index.html
9. OpenMP Architecture Review Board: OpenMP application program interface version 4.0 (July 2013), http://www.openmp.org/mp-documents/OpenMP4.0.0.pdf
10. Wylie, B.J., Frings, W.: Scalasca support for MPI+OpenMP parallel applications on large-scale HPC systems based on Intel Xeon Phi. In: Proceedings of the Conference on Extreme Science and Engineering Discovery Environment: Gateway to Discovery, p. 37. ACM (2013)