Profiling Hybrid HMPP Applications with Score-P on Heterogeneous Hardware

Marc SCHLÜTTER [a] Peter PHILIPPEN [a] Laurent MORIN [b] Markus GEIMER [a]
Bernd MOHR [a]

[a] *Jülich Supercomputing Centre, Germany*
[b] *CAPS Entreprise, France*

Abstract.

In heterogeneous environments with multi-core systems and accelerators, programming and optimizing large parallel applications turns into a time-intensive and hardware-dependent challenge. To assist application developers in this process, a number of tools and high-level compilers have been developed. Directive-based programming models such as HMPP and OpenACC provide abstractions over low-level GPU programming models, such as CUDA or OpenCL. The compilers developed by CAPS automatically transform the pragma-annotated application code into low-level code, thereby allowing the parallelization and optimization for a given accelerator hardware. To analyze the performance of parallel applications, multiple partners in Germany and the US jointly develop the community measurement infrastructure Score-P. Score-P gathers performance execution profiles, which can be presented and analyzed within the CUBE result browser, and collects detailed event traces to be processed by post-mortem analysis tools such as Scalasca and Vampir.

In this paper we present the integration and combined use of Score-P and the CAPS compilers as one approach to efficiently parallelize and optimize codes. Specifically, we describe the PHMPP profiling interface, it's implementation in Score-P, and the presentation of preliminary results in CUBE.

Keywords. accelerator, CUBE, GPU, GPGPU, HMPP, OpenACC, optimization, performance, PHMPP, profiling, Score-P, tools, tracing

Introduction

The domain of high-performance computing is targeted by scientific applications in many fields, like geological analysis, bio-medical research, financial model evaluation or automotive crash simulation. All these applications require massive computational power and for decades their improvement has relied on the speedup of simple sequential processing. However, over the last ten years the computing speed of these sequential applications has significantly slowed down while the transistor density is still doubling every eighteen months as Moore's law predicts.

From the potential of increasing numbers of transistors, a more radical change in processor architecture has occurred. The design of chips with a multitude of parallel cores has nearly replaced increasing the speed of single cores. The first multi-core architectures were built using a shared-memory model and this trend has led to the design of

massively parallel heterogeneous architectures. In this context, the segment of graphics processing units (*GPUs*) has evolved from pure visualization to general purpose GPUs (*GPGPUs*) which are now widely used to accelerate scientific applications. For example, the CUDA [1] programming model can be used to accelerate applications requiring a massive number of floating-point operations with NVIDIA GPUs. Also, directive-based programming models such as HMPP and OpenACC provide abstractions over low-level programming models, thereby simplifying application development.

One of the biggest challenges developers of large parallel applications are facing is to efficiently use all available resources concurrently, especially when targeting heterogeneous hardware. This is aggravated by the fact that the use of high-level programming models hides many hardware-related performance aspects. Therefore, it is important to measure and identify how the runtime schedules the work load to evaluate a parallelization based on such high-level models. This motivates the need for performance-analysis tools providing specific support for heteogeneous architectures.

In this paper we present our work on integrating the CAPS compilers for the HMPP and OpenACC high-level GPU programming models with the community measurement infrastructure Score-P. To this end the profiling interface PHMPP has been developed and implemented. It allows recording events with different granularity, e.g., user application, CAPS runtime, CUDA runtime, and accelerator hardware. To our knowledge the integration of high- and low-level GPU support with OpenMP and MPI for profiling and tracing of applications, with a single combined measurement infrastructure, is unique. Combined with providing output in open data formats that can be used by multiple tools for further analysis, this presents our main contribution.

1. Related Work

Accelerator Programming Models We distinguish between high- and low-level programming models for accelerators. The low-level models CUDA [1] and OpenCL [2] both provide interfaces for performance measurement tools. As high-level accelerator programming model, OmpSs [3] developed by the Barcelona Supercomputing Center (BSC) pursues abstracting the accelerator level, similar to HMPP. The Portland Group (PGI) compiler suite [4] provides a directive set for offloading code regions to hardware accelerators. A consortium consisting of NVIDIA, CAPS, PGI and CRAY have proposed OpenACC [5], a directive-based programming model for accelerators. The OpenMP 4.0 specification [6] also contains directive support for accelerators, but at the time of our work no implementation was available yet. Neither the PGI accelerator directives nor OpenACC or OpenMP 4.0 have direct support for external profiling and tracing tools.

Measuring Performance on Accelerators Many hardware vendors provide tools for profiling applications, which are tailored to their product line. For example, NVIDIA provides the NVIDIA Visual Profiler as part of the CUDA Toolkit [7]. With VTune, Intel offers a tool that supports the Intel Xeon Phi coprocessor and OpenCL [8]. The Paraver tool set, developed at BSC, provides a time-line viewer to visualize event traces for CPU threads and accelerators programmed using either OmpSs, CUDA or OpenCL [9,10]. The traces are collected with the Extrae instrumentation package. VampirTrace, the original measurement system of the Vampir trace visualizer developed at TU Dresden, also has been extended to support the PHMPP interface [11], as well as OpenCL, CUPTI and

CUDA hardware counters. However, none of these tools provide a solution for combined profiling and tracing as well as open data formats.

2. CAPS Many-Core Compilers

The evolution from single-core to multi-core processors has raised a barrier for application developers, especially in regard to adopting these new architectures for legacy codes. To exploit the potential of heterogeneous and massively parallel architectures, most applications must be either rewritten or, even worse, redesigned. To reduce the cost of porting applications, the *HMPP* programming model proposes a way to express the remote and parallel execution of kernels on accelerators in a brief and simple manner. The purpose of the *CAPS compiler* is to provide a complete workbench, which greatly simplifies the compilation of programs for heterogeneous architectures.

Heterogeneous and Parallel Programming Using Directives With the progress of parallel architectures, scientific applications have been extended to express parallelism within their algorithms. For shared memory architectures, the OpenMP programming model enables this inside legacy C or Fortran codes with little effort. It defines a set of directives which can be placed around code regions to supply the compiler with information on how this region can be parallelized. Using this principle for heterogeneous and massively parallel architectures, CAPS entreprise proposed the *HMPP* programming model which enables offloading of computational kernels to accelerators [12]. It is based on the concept of "*Codelets*": small functions, which define the computation that should be remotely executed. Initially only CUDA architectures were targeted, but it has since been extended to other architectures with support for the OpenCL language [2]. Also, HMPP has been proposed as an open standard named *OpenHMPP*.

Compilation for Many-Core Heterogeneous Architectures The CAPS compiler is a source-to-source compilation framework taking care of all the necessary steps for the generation of hybrid and parallel applications. First, the code that should be offloaded to an accelerator is identified. The compiler then generates the codelet parallel code and compiles it for the targeted architecture(s). Next, the application is modified to use the offloaded code via the HMPP runtime and compiled with the original compilation chain. The runtime system is then responsible for dynamically allocating the accelerator, transferring the necessary data, and initiating the remote execution of the codelet code.

Optimizing HMPP Applications The optimization of the code executed on the accelerator affects only the codelet implementation. The code should be composed of one or several loop nests that will be the basis for the parallel execution on the device. HMPP provides a dedicated set of directives offering a wide choice of loop transformations (e.g., unrolling, blocking) and a way to control important aspects of the parallel execution like reductions or block sizes of a data parallel grid. Furthermore, HMPP also provides a set of directives to tune device management, memory allocations, and the number of memory transfers. These optimizations are highly portable across different accelerator models.

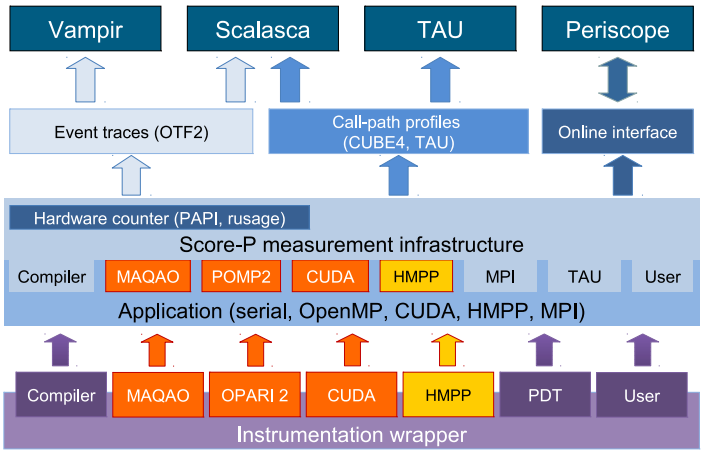

Figure 1. Architecture of the Score-P instrumentation and measurement system and integration with supported analysis tools. In the scope of the H4H project, special focus was put on the integration of MAQAO, advanced OpenMP support with OPARI2 and the POMP2 interface, and GPU support by integration of CUDA and HMPP.

3. The Score-P Measurement System

The first implementation of an application is often not optimal. Many factors that influence performance only become apparent after executing the program on the targeted hardware and scale, while performing detailed performance measurements. A number of tools exist to assist developers in analyzing their software to detect a wide range of performance problems. Methods for collecting relevant data about an application range from static analysis of the source code or the executable, over various runtime sampling approaches, to detailed profiling and tracing of all relevant sections of the program during its execution.

The community measurement system Score-P [13] focuses on instrumentation-based profiling and tracing to collect detailed information about an application's runtime behavior. It is the result of a collaboration between the developers of the performance analysis tools Periscope [14], Scalasca [15], TAU [16], and Vampir [17]. In this section, we will briefly describe the basic architecture of Score-P and demonstrate how the results of measurements are presented to the user.

Architecture Figure 1 shows the basic architecture of Score-P. The basis for measuring the runtime behavior of an application is the instrumentation of all regions of the code a user is interested in. This can be achieved in a number of different ways. One method is to insert function calls into the program, e.g., at the beginning and end of functions. Another possibility is to make use of library interposition, as with PMPI, the profiling interface of MPI.

Following a modular design, the different instrumentation techniques are loosely coupled with their corresponding adapters, which are part of the Score-P measurement libraries that are linked to the instrumented application. Their purpose is to implement the function calls that were inserted at instrumentation time. During measurement, the provided information is processed to generate generalized events that can be handled by the measurement core. Here, timestamps are taken and, depending on the settings, the

information is stored in form of profiles and/or traces in a unified and compressed way. The main enhancements implemented during the EU ITEA2 project H4H [18] (Hybrid Programming for Heterogeneous Architectures) are highlighted in Figure 1, with special focus put on the integration of HMPP.

Processing and Visualization of Results The profiles generated by Score-P measurements can be visualized using the separately distributed profile browser CUBE. It can be used to examine the different collected metrics (time, number of visits, bytes transferred, etc.) in respect to the dynamic call-tree of the application as collected during runtime. Furthermore, the metrics are also mapped to the different processes (MPI) and threads (OpenMP), see Figure 4.

In contrast to a profile which provides only aggregated information for the whole run, event traces provide very detailed information about the dynamic behavior of the application. To explore the event traces produced by Score-P, the visual trace analyzer Vampir can be used. It allows to interactively inspect the collected data and to focus on individual instances of execution bottlenecks to identify their causes.

Since traces can easily become large and unwieldy, it is often necessary to combine both approaches. While profiling can already pinpoint interesting sections of a code on a high level, selective tracing of such regions then provides the input data for an in-depth investigation. Therefore, Score-P supports both modes of operation, easily selectable using environment variables.

4. Integration

To enable measurement of applications compiled for heterogeneous architectures with the CAPS compilers, a new interface for HMPP was designed: the PHMPP profiling interface. This interface as well as the necessary extensions of Score-P are described in more detail below.

4.1. PHMPP Profiling Interface

The CAPS compiler suite can be used to compile and execute HMPP applications to leverage the computational performance of massively parallel accelerators. Programmed with HMPP, the code can be very simple but does not always provide the expected performance without a bit of tuning, i.e., optimizing various aspects of the execution process of the code: the device and memory allocations, memory transfers, and kernel executions. Using the directive set, the user can quickly optimize the remote execution of the code and improve the performance of computational kernels. To do so, the origin of the lack of performance first has to be determined. However, until recently, profiling of an application had to rely on the combination of a regular profiling tool for the host part of the code and a target-specific profiling tool for the kernels executed on the device. This operation requires expertise on the accelerator and a basic understanding of the target code generated by the CAPS compiler. On the other hand, the CUDA support available in Score-P 1.1 allowed measuring performance on NVIDIA GPUs, but did not take high-level interfaces such as HMPP into account. In this context, the *PHMPP* interface [19] proposes a unified *Application Programming Interface* (*API*) to monitor events related to the remote execution of a kernel with the HMPP programming model.

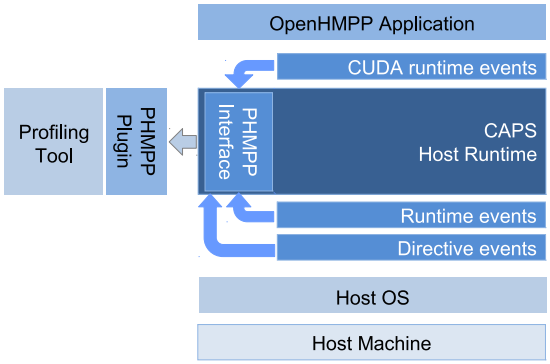

Figure 2. Overview of the PHMPP profiling interface.

A tool that makes use of the PHMPP interface may register callback functions, for certain types of events, with the HMPP runtime. These functions then get called with the objective to monitor all types of events that could be related to the HMPP operations in the application. These events can have multiple sources:

- The usage of an HMPP directive: device allocation, memory transfers, etc.
- A generic operation performed by the CAPS runtime on the host: implicit host memory allocation, implicit or explicit memory transfers, kernel loading, etc.
- A target-specific operation performed by the CAPS runtime using the target driver: context or stream creation in CUDA, queue management in OpenCL, etc.
- Target-specific operations performed by the kernel code generated at compile time: memory initializations and transfers, CUDA grid execution, etc.

All these operations focus on various aspects of the HMPP execution that differ in granularity, in target dependency, and in usability. They require different levels of expertise and are not all pertinent at the same time. To manage this complexity, the PHMPP interface has been designed in two layers: the backbone API and the semantic extensions.

The first layer, the backbone API, is in charge of the interface between the CAPS runtime and external profiling tools. It represents the main part of the PHMPP interface and is not intended to change over time. The API defines the syntax and the communication protocol between tools, but only specifies a simplified abstraction for five basic event types that might occur during the execution. In particular, start/stop events for accelerator allocation, device memory allocation, kernel execution, data transfers and system-related activities are defined. These events will encapsulate all other, more specific events.

The second layer – the semantic extensions – is composed of the specification of a set of different groups of events, each describing a particular aspect of the HMPP execution. Each group – or semantic extension – can focus on either a particular target or a particular granularity of events. Semantic extensions can easily be updated with the evolution of targets or programming languages, or new extensions can be introduced for new targets or runtimes. Currently, user-side directive calls for HMPP and OpenACC, runtime events of CUDA and OpenCL, as well as events from the host HMPP runtime are covered.

Figure 2 illustrates the different runtimes and components used by an HMPP application. Events are generated from the CAPS runtimes but in the future could also be

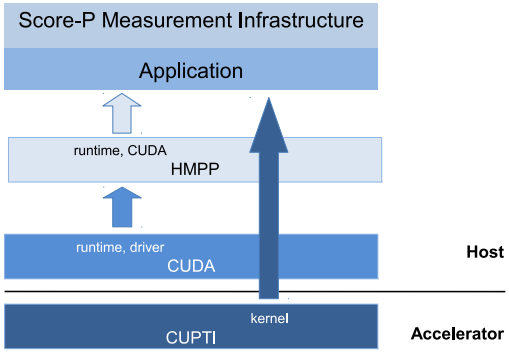

Figure 3. Flow of information between the HMPP and CUDA runtimes, and the Score-P measurement system.

triggered from the code generated for the kernel. The PHMPP backbone is available in the CAPS host runtime and makes the link with one or multiple profiling tools. Once the connection is made, events generated from different sources of the CAPS compiler workbench are propagated to the PHMPP backbone, and further sent to the profiling tool.

4.2. Score-P Software Extensions

Automatic Instrumentation of HMPP Applications Usually, the CAPS compiler driver is prefixed to the actual (host) compile command and implicitly decomposes the compilation into separate steps for host-side code and accelerator kernels. This scheme conflicts with Score-P's instrumenter which needs to process (i.e., instrument) the host-side code. Therefore, the interaction between Score-P and HMPP has been adapted: the Score-P instrumenter transparently invokes the CAPS compiler driver using low-level commands to perform its steps explicitly, thereby allowing Score-P to instrument the generated host code while leaving GPU kernels untouched. Since HMPP applications are automatically recognized, the typical Score-P instrumentation workflow can be retained, providing a consistent user experience.

The Score-P PHMPP Adapter On the Score-P side, an adapter was implemented to provide all the necessary functionality to interact with the PHMPP interface. It is supplied to the HMPP runtime as a plugin and is responsible for initializing the communication by registering the required callback functions, which in turn prepare the data provided via the PHMPP interface into a form suitable for the Score-P measurement core. Due to the abstraction provided by PHMPP, it is not necessary to distinguish between HMPP and OpenACC constructs at this level.

Three components are responsible for generating information: the usual Score-P adapters (compiler, MPI, OpenMP, etc.), the PHMPP runtime, and the hardware-dependent, low-level accelerator runtime (such as CUPTI). The interaction between these components is illustrated in Figure 3. Events from different sources are gathered and combined into one cohesive measurement by Score-P. All relevant host-side information related to operations driving the CUDA runtime is intercepted by the HMPP runtime and enriched with information about the corresponding directives in the developer's source code. This data then gets passed to the Score-P adapter via the PHMPP interface. On the other hand, device-level data, i.e., timing information about kernel executions and data

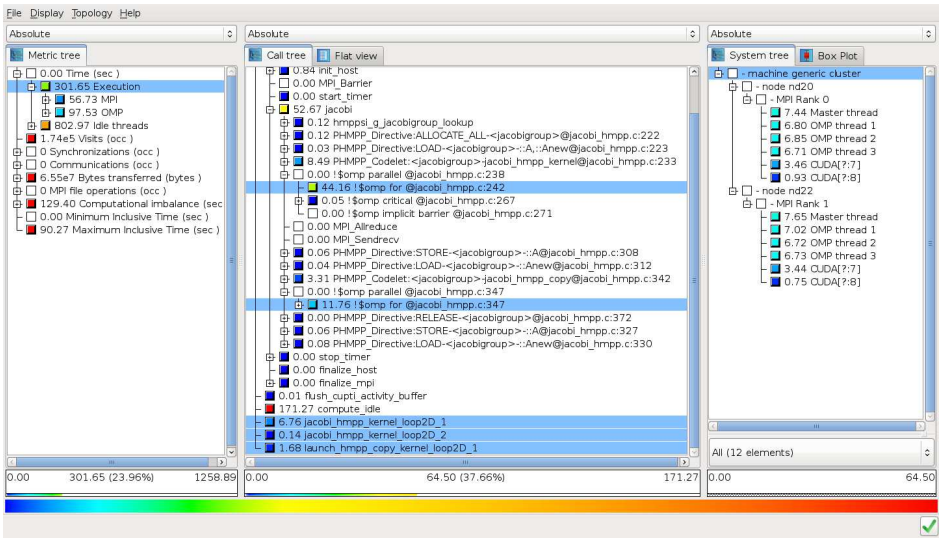

Figure 4. Hybrid MPI, OpenMP and HMPP measurement of a Jacobi benchmark.

transfers, is collected using the low-level CUPTI interface and directly processed by the Score-P measurement system.

Especially the latter presents the difficulty that events originating on the accelerator are registered asynchronously. At the time of their transfer to the host (during host/GPU synchronization points), the causal relation between the kernel executions and the HMPP source-code directives cannot be directly made anymore. Even the context IDs provided by CUPTI are of limited value, as the HMPP runtime internally creates multiple CUDA contexts and does not expose this information yet. However, we plan to address this limitation as part of our future work.

5. Exemplary Measurements and Analysis

In this section, we highlight the benefits of using the enhanced version of Score-P for performance measurements of HMPP applications. These measurements have been carried out on the H4H project platform Nova2, an Intel Xeon-based HPC cluster provided by Bull, equipped with Nvidia Tesla GPUs.

As a proof-of-concept, we performed measurements with a hybrid implementation of the Jacobi algorithm using MPI, OpenMP and HMPP in combination. Thereby we demonstrate how the difficulties in measuring performance on heterogeneous hardware can be addressed. Special focus is put on the interaction of the different hardware components. In this case the Jacobi algorithm distributes the data over three dimensions. The first level is the inter-process work distribution using MPI. On each process the data columns are distributed between the second and the third level, i.e., OpenMP threads and HMPP kernels, respectively. Load balancing within the node can be controlled by an input parameter defining the percentage of work offloaded to the GPU. This test case is using HMPP and OpenMP side by side, but it would also be possible to start HMPP kernels out of an OpenMP parallel region, e.g., to leverage multiple accelerators.

Figure 4 shows how results are presented in the CUBE profile browser and how the interaction between the different components is visualized. Reading the result from left to right, the user can select the type of metric or issue he wants to analyze in the first column. Choosing, e.g., the pure execution time, the second column shows how this time is distributed over the calltree. When selecting a subset of entries, the third column on the right side shows the distribution of this selection over the system tree, which in this hybrid case contains MPI ranks, OpenMP CPU threads and CUDA streams, the underlying layer for the HMPP kernels. As HMPP uses separate CUDA streams for different types of kernels, each process shows two streams: one for the Jacobi kernel and one for the copy kernel. Due to the asynchronous nature of the execution, the calltree does not just have one single root node, following the course of the main program. Instead it also shows nodes for the execution times of each kernel, which originate in the CUPTI interface. Since the kernel execution times from the CUPTI interface are not inherently linked to call sites, the identification has to be done by matching names, including kernel splits by loop level. With the help of kernel executions and OpenMP loops, the application developer can get an impression of the distribution of the time spent and check if the load is well-balanced between CPU threads and GPU streams, to see if the given hardware is used efficiently. Figure 4 shows that even in a relatively straightforward test case, finding the balance between components with notably different characteristics is a task worth of investigation. Given that the OpenMP for-loop follows the synchronous model, we see here a mixture of synchronous and asynchronous behavior.

6. Conclusions and Future Work

In this paper, we have shown that the automatic instrumentation of hybrid MPI, OpenMP and HMPP applications allows users to gain an overview on load balancing and hardware utilization with respect to the different hardware components. A proof-of-concept, using a hybrid Jacobi algorithm, has been presented. We were able to show that the integration of HMPP and Score-P provides valuable information and semantic links between the different parts of the runtime system. The demonstrated method is a new and unique approach.

The asynchronous execution mode of kernels and tasks in general presents open challenges for call-path profiling in terms of statistical loss of information for different execution behaviors and increased overhead due to task granularity. For accelerators in particular, deeper call-paths will likely have to be considered and visualized in the future.

While there are still open issues with the analysis of HMPP kernels, and GPU kernels in general, the challenges of heterogeneous hardware and the necessity of scalability justify the use of runtime systems like HMPP, aiding the application developer in streamlining his parallelization efforts by abstracting the underlying hardware. For Score-P, the most direct goal will be to enhance the link between the different information sources, e.g., by extending PHMPP with new semantic extensions providing the relation between high- and low-level operations.

Acknowledgements

This work has been supported by the EU ITEA2 project H4H, the German Ministry of Education and Research, the French Ministry for Economy, Industry, and Employment, CAPS Entreprise and Forschungszentrum Jülich GmbH.

References

[1] NVIDIA. CUDA C Programming Guide. http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html. Accessed: 2013-Jul-07.

[2] The Khronos Group. The open standard for parallel programming of heterogeneous systems. http://www.khronos.org/opencl/, 2010. Accessed: 2013-Jul-07.

[3] J. Bueno, J. Planas, A. Duran, R.M. Badia, X. Martorell, E. Ayguade, and J. Labarta. Productive programming of GPU clusters with OmpSs. In *Parallel Distributed Processing Symposium (IPDPS), 2012 IEEE 26th International*, pages 557–568, 2012.

[4] The Portland Group. PGI accelerator programming model. http://www.pgroup.com/resources/accel.htm, 2010. Accessed: 2013-Jul-07.

[5] CAPS CRAY NVIDIA PGI. The OpenACC application programming interface. http://www.openacc.org/sites/default/files/OpenACC\%202\%200.pdf, July 2013. Accessed: 2013-Jul-07.

[6] OpenMP Architecture Review Board. The OpenMP API specification for parallel programming, version 4.0. http://openmp.org/wp/openmp-specifications/, 2013. Accessed: 2013-Jul-26.

[7] NVIDIA. Profiler user's guide. http://docs.nvidia.com/cuda/profiler-users-guide/index.html. Accessed: 2013-Jul-02.

[8] Intel. Intel® VTune™Amplifier 2013. http://software.intel.com/sites/products/documentation/doclib/stdxe/2013/amplifierxe/lin/lin_ug/index.htm. Accessed: 2013-Jul-02.

[9] J. Labarta, J. Gimenez, E. Martinez, P. Gonzalez, H. Servat, G. Llort, and Xavier Aguilar. Scalability of tracing and visualization tools. In *Parallel Computing: Current & Future Issues of High-End Computing*, number 33 in John von Neumann Institute for Computing Series, pages 869–876. Central Institute for Applied Mathematics, 2005.

[10] J. Caubet, J. Gimenez, J. Labarta, L. De Rose, and J. S. Vetter. A dynamic tracing mechanism for performance analysis of OpenMP applications. In *Proceedings of the International Workshop on OpenMP Applications and Tools: OpenMP Shared Memory Parallel Programming*, WOMPAT '01, pages 53–67, London, UK, UK, 2001. Springer-Verlag.

[11] Technische Universität Dresden. VampirTrace 5.14.3 with extended accelerator support. http://www.tu-dresden.de/die_tu_dresden/zentrale_einrichtungen/zih/forschung/software_werkzeuge_zur_unterstuetzung_von_programmierung_und_optimierung/vampirtrace/accelerator/dateien/VampirTraceManual/VampirTrace%205.14.3-gpu1-user-manual.pdf. Accessed: 2013-Jul-22.

[12] Romain Dolbeau, Stéphane Bihan, and François Bodin. HMPP: A hybrid multi-core parallel programming environment. In *Workshop on General Purpose Processing on Graphics Processing Units (GPGPU 2007)*, 2007.

[13] A. Knüpfer, C. Rössel, D. an Mey, S. Biersdorff, K. Diethelm, D. Eschweiler, M. Geimer, M. Gerndt, D. Lorenz, A. D. Malony, W. E. Nagel, Y. Oleynik, P. Philippen, P. Saviankou, D. Schmidl, Sameer S. Shende, R. Tschüter, M. Wagner, B. Wesarg, and F. Wolf. Score-P – A joint performance measurement run-time infrastructure for Periscope, Scalasca, TAU, and Vampir. In *Proc. of 5th Parallel Tools Workshop, 2011, Dresden, Germany*, pages 79–91. Springer Berlin Heidelberg, September 2012.

[14] S. Benedict, V. Petkov, and M. Gerndt. Periscope: An online-based distributed performance analysis tool. In *Tools for High Performance Computing 2009*, pages 1–16. Springer Berlin Heidelberg, 2010. 10.1007/978-3-642-11261-4_1.

[15] M. Geimer, F. Wolf, B. J. N. Wylie, E. Ábrahám, D. Becker, and B. Mohr. The Scalasca performance toolset architecture. *Concurr. Comput. : Pract. Exper.*, 22(6):702–719, April 2010.

[16] S. Shende and A. D. Malony. The TAU parallel performance system. *Int. J. High Perform. Comput. Appl.*, 20(2):287–311, May 2006.

[17] A. Knüpfer, H. Brunst, J. Doleschal, M. Jurenz, M. Lieber, H. Mickler, M. S. Müller, and W. E. Nagel. The Vampir performance analysis tool-set. In Michael Resch, Rainer Keller, Valentin Himmler, Bettina Krammer, and Alexander Schulz, editors, *Tools for High Performance Computing*, pages 139–155. Springer Berlin Heidelberg, 2008. 10.1007/978-3-540-68564-7_9.

[18] H4H project web page. http://www.h4h-itea2.org/. Accessed: 2013-Jul-08.

[19] CAPS entreprise, Rennes. *H4H - HMPP Profiling Event Specification*, version 3.2.0 edition, 2012.