SAND2018-8280C

Profiling and Debugging Support for the Kokkos Programming Model

S.D. Hammond[1], C.R. Trott[1], D. Ibanez[1], and D. Sunderland[1]

Center for Computing Research
Sandia National Laboratories
Albuquerque, New Mexico, 87123
United States of America
sdhammo@sandia.gov

Abstract. Supercomputing hardware is undergoing a period of significant change. In order to cope with the rapid pace of hardware and, in many cases, programming model innovation, we have developed the Kokkos Programming Model – a C++-based abstraction that permits performance portability across diverse architectures. Our experience has shown that the abstractions developed can significantly frustrate debugging and profiling activities because they break expected code proximity and layout assumptions. In this paper we present the Kokkos Profiling interface, a lightweight, suite of hooks to which debugging and profiling tools can attach to gain deep insights into the execution and data structure behaviors of parallel programs written to the Kokkos interface.

Keywords: Profiling · Performance-Portable · Programming Models

1 Introduction

The future of supercomputing is changing. The past two decades of high-performance computing hardware have been dominated by a period of relative architectural stasis – multiple homogeneous computing nodes interconnected with a high-speed network. Leading contemporary machines and those of the next decade look almost certain to be characterized by a period of rapid architectural change, not least, in part because of a push to achieve machines capable of "Exascale" class computation (defined by the United States Department of Energy to be 50X greater application performance [16]).

Such a significant change in hardware diversity and the much greater use of computational accelerators presents a profound change for the programming models upon which scalable scientific codes are built. While we can expect the use of message-passing (MPI) based distributed memory algorithms to continue, the programming of on-node computation will change dramatically. Unfortunately, our experience to date has shown that each class of hardware often requires different approaches to programming, and in some cases, requires an outright different programming model to be employed. At Sandia National Laboratories, which is one of the largest supercomputing users in the world, and at many other

HPC sites, such a position is untenable. Reprogramming of applications for each class or generation of computing architecture would require an unsustainable level of investment.

To this end, we have developed the Kokkos Programming Model [7],[6], [5] – a C++ meta-template driven approach to programming high-performance applications, that are intended to be portable across diverse computing architectures. Examples of similar approaches also include RAJA [14], Alpaka [26], Intel Thread Building Blocks (TBB) [18] and Thrust [2]. From a single C++ code base it is possible to write one version of a code and have this run efficiently on a wide range of computing architectures including multi-core processors such as Intel's Xeon [13] [11],[17] and IBM's POWER server-class [25][19] processor families, Intel's Knights Landing many-core processor [22][23] and GPUs from NVIDIA and AMD. Behind the scenes, the developer authored kernels are retargeted to architecture-relevant programming models using templating by a C++11 complaint compiler to utilize standard threads, OpenMP [4] threading, OpenMP offload or the CUDA programming model. Such an approach is appealing for multiple reasons: (1) we have shown our ability to rapidly port applications to novel architectures, significantly reducing development costs; (2) by providing commonly requested programming model features we have enabled application developers to reuse well crafted and optimized routines instead of writing their own, further reducing development burden and implementation bugs while also increasing runtime performance, and, (3), we have demonstrated the ability of a code base to achieve extremely agile portability while maintaining performance levels at close to that provided by native programming models. We note that, the use of native programming models will almost always provide the outright highest performance levels, but often by limiting the portability of the code to single hardware solution. Instead, we target performance at roughly 90% of a well written and optimized native implementation but, for which, our solution can provide cross-platform execution.

However, the approach pioneered in the Kokkos programming model is not without its own problems. The extensive use of C++ templating and meta-templating creates, at times, deep levels of abstraction, often implemented through traversal of internal functions and implementation header files. Further, the use of features developed within the Kokkos model exacerbate the complexity of the underlying implementation code. The anecdotal effect has been that codes developed using Kokkos have been more challenging to debug and profile than those which utilize a purely native programming model where tool support is often well tuned by the implementing vendor. A good example is the use of OpenMP on Intel platforms which is cleanly understood using the VTune Amplifier XE profiler but which struggles to disambiguate parallel regions when OpenMP is used from within the Kokkos runtime. The challenge has come in part because debugging and profiling tools are unable to efficiently handle the deepening hierarchy of calls, functions and header files implemented using Kokkos, and, further, many tools have utilized expected behavior of programs, particularly in the case of OpenMP, that parallel directive markings are directly adjacent to users code.

In the case of Kokkos, this assumption is violated as lambda functions or C++ functors are given to Kokkos to map into parallel kernels.

Such observations come at potentially the worst time for developers – charged with making their code cross platform *and* performant, their need for insightful debugging and profiling tools is perhaps at its maximum as our code teams begin to understand execution behavior on the next-generation of HPC compute nodes. Our observation has been that many leading tools have failed to provide the insights needed. Failures have included: misattribution of execution timing, misrepresentation of parallel behavior on execution timelines, and, at times, sufficiently complex performance outputs that the developers are unable to wade through the results to diagnose performance/correctness issues.

This paper is laid out as follows: Section 2 provides a description of the Kokkos Profiling interface, including the types of events for which the system provides support. In Section 3, we describe several tools that have been implemented using the interface and describe how these can support application developers. We conclude the paper in Section 4.

2 Kokkos Profiling Tools

In this paper, we present our latest work in the development of a Kokkos profiling interface (which we often refer to as *KokkosP*) layer which is intended to provide a suite of hooks that profiling and debugging tools are able to connect to. Once attached to the predefined hooks, profiling and debugging tools are able to be called at defined events during the execution of Kokkos-based applications. This work has been developed and refined over a period of four years and has been robustly tested in large-scale production applications developed at Sandia National Laboratories. The contributions of this work include:

- **Provision of a Cross-Platform Suite of Programming Hooks** that are consistent across underlying execution hardware and baseline programming model. This provides a uniform developer experience across HPC node types and systems, enabling them to perform the same analysis activities regardless of target platform;
- **Ability to Analyze Parallel Computation and Kokkos Data Structure Allocation/Lifetime** - our hooks cover two important aspects provided by the Kokkos programming model: (1) its ability to target performance-portable execution of computational kernels using several parallel patterns (all of which have associated profiling hooks), and, (2), the ability to create multi-dimensional arrays (Kokkos "Views") and have these tracked by profiling tools from creation, to copying to/from acceleration devices and eventual destruction. In so doing, we are able to exploit deep knowledge about the Kokkos programming model handling of users direct requests, and, in the cases of the latest POWER9/Volta machines, the ability of the hardware to perform some operations, such as data migration from the host to the GPU devices, automatically.

– **Compatibility with Leading Vendor Solutions** - we provide "connectors" which are our terminology for small layers that perform translation from a Kokkos event stream into events that can be consumed by leading vendor tools such as Intel's VTune Amplifier XE profiler, NVIDIA's NSight tool and Cray CrayPAT suite. While we provide our own basic cross-platform capable tools, which we routinely use ourselves, the ability to harness leading vendor solutions enables us to benefit from their deep hardware knowledge and ensure that Kokkos programs are correctly profiled, analyzed and represented in each tools' user environment.

2.1 Overview of the Kokkos Profiling Interface

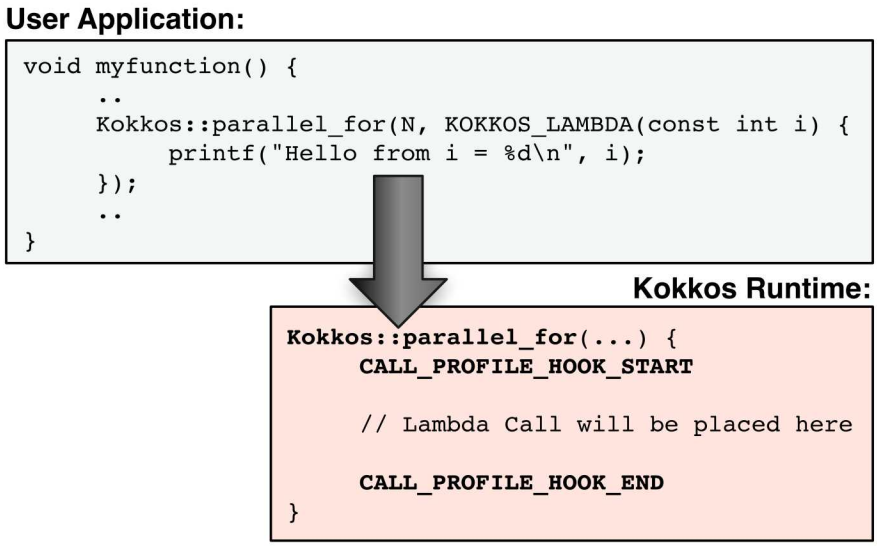

Fig. 1. Compilation of the Kokkos Profiling Hooks

Kokkos profiling hooks are implemented as a collection of function pointers which can be called when specific events of interest occur within the Kokkos runtime. Figure 1 gives a high-level overview of how the function pointer hooks would be placed into compiled code so that they would be called immediately prior and after a parallel-for loop. Profiling hooks are registered against each particular event handling function pointer through calls to the Linux `dlopen` interface which is utilized to load a specific profiling tool and identify the location of function calls to map into the runtime. In the event that there is no mapping from a Kokkos profiling hook to a function call within the loaded profiling tool,

or that no tool is loaded for analysis, the function call is marked as empty and ignored during execution.

Unlike many profiling tools in this class, many of the Kokkos profiling hooks are automatically built into the Kokkos programming model during compilation and require no specific user/developer coding, although an extensive API is also provided where developers can provide *additional* definition of sections of their application. In general, users can consider Kokkos profiling hooks to be annotations hidden within the programming model that are compiling into the runtime by default. Through careful design we have been able to insert our hooks such that when performance tools are not running, there is no measurable overhead from the presence of the profiling hooks to running applications. However, when needed, the hooks can be connected to dynamically (through dynamic loading of a performance tool) without requiring recompilation of the application – typically an expensive operation for applications with millions of lines of source code. The effect is that applications written to use Kokkos are compiled to have an "always-on" profiling capability that has effectively zero overhead when not in use.

2.2 Event Callbacks from Kokkos

Table 1 presents the types of events which can be subscribed to via the existing implementation of the Kokkos profiling interface. Input parameters are generated by the Kokkos runtime and provided to the currently loaded profiling tool. Output parameters are provided as a pointer which the Kokkos runtime expected the tool to fill in. Three broad classes of events exist: (1) events which relate to parallel execution; (2) events relating to application segments, and, (3), events relating to data structure allocation, deallocation and copying.

Kokkos Events - Execution Dispatch The execution dispatch events shown in Table 1 relate to the three principle kernel classes provided in the Kokkos runtime. At the time of writing, an additional mapping of tools to Kokkos task construction and dispatch is also in development. Since kernels in Kokkos can, in theory, execute asynchronously, each call to enqueue a kernel triggers an event, for which the profiling tool must provide a unique identifier. Kokkos tracks this unique identifier with the enqueued kernel and then returns it when the kernel has completed (a call to the 'end' functions). Tool developers are therefore expected to be able to track kernel timing by the unique identifier. Execution dispatch profiling hooks require no additional application code and are enabled by default (*i.e.*, unlike some instrumentation libraries, application code does not need to be scattered with calls to an instrumentation library).

Kokkos Events - Application Code Segments Our experience with developing high-performance parallel kernels for the Trilinos solver library [12] has shown that providing additional information around each kernel can aid in the task of analyzing complex full-application behavior. This is typically made more

Event Type	Input Parameters	Output Parameters
Execution Dispatch		
Begin Parallel For	Kernel Name, Device ID	Unique Kernel ID
End Parallel For	Kernel ID	None
Begin Parallel Reduce	Kernel Name, Device ID	Unique Kernel ID
End Parallel Reduce	Kernel ID	None
Begin Parallel Scan	Kernel Name, Device ID	Unique Kernel ID
End Parallel Scan	Kernel ID	None
Application Code Segments		
Push Region	Region Name	None
Pop Region	None	None
Create Section	Region Name	Unique Section ID
Start Section	Section ID	None
Stop Section	Section ID	None
Destroy Section	Section ID	None
Data Structures		
Data Allocation	Data Space, Label, Start Address, Size	None
Data De-Allocation	Data Space, Label, Start Address, Size	None
Deep Copy Start	Origin Data Space, Origin Label, Origin Start Address Dest. Data Space, Dest. Label, Dest. Start Address, Copy Length	None
Deep Copy End	None	None

Table 1. Event Types which can be subscribed to using the Kokkos Profiling/Debugging Interface

complex in larger applications because multiple calls paths may eventually execute the same kernel and the sheer number of kernels running in an application can be difficult to locate. To this end, Kokkos provides two methods for providing a nesting/description of application structure to profiling tools: (1) code regions, and, (2) code sections.

A code region is a lightweight mechanism for describing code structure that behaves like a stack. A region is pushed onto the current structure stack and then popped once completed. The overhead associated with maintaining the stack is extremely low and so regions can be aggressively used without introducing significant extra runtime when using profiling tools. Historically, push/pop mechanisms have been well supported by vendor tools such as Intel's VTune and NVIDIA's NSight profiling tools.

Code sections have extra overhead but greater flexibility. Each section is created and assigned a unique identifier. The section can then be repeatedly started and stopped prior to its final deletion. This approach lends itself to artifacts such as user-defined timers which can execute concurrently and may not be retired by the programmer using a first-in-first out approach (that would better map to a code region). The free-form nature of sections allows them

to overlap and to potentially be passed around via calls to libraries etc if the programmer has interest in doing so. The overheads of using code sections relate to the number of concurrent sections in an application as the section identifier must be looked up by a profiling tool prior to any operation.

Both code regions and code sections require additional code to be added to the application, however, in the case of Trilinos, we have been able to hide a number of these calls in existing code structures such as timers provided by the core Teuchos utility framework [1].

Kokkos Events - Data Allocation Data allocation events from the Kokkos runtime are triggered through the creation of structures like Kokkos multi-dimensional arrays ('Views'), through creation of parallel containers such as multi-threaded maps and vectors or through direct calls to the Kokkos `malloc` function. Since Kokkos is responsible for mapping allocations to memory spaces which can be on the host or any number of devices, a data space identifier is provided to the tools for creation and de-allocation. Such an approach is critical for systems which do not implement shared virtual addressing since a pointer address may not be unique in the system (and therefore cannot uniquely identify an allocation). A user defined label (string) is also provided to the event so that a programmer friendly name mapping to the allocations can be presented in user tools such as debuggers or profilers. These allocation labels are also utilized by Kokkos when expensive range-checking is compiled into the code to identify which data structures are being used to perform illegal accesses.

Data copying is typically an expensive but critical component of optimized host-device style execution (such as a host processor and a GPU). Since deep copies often need to be carefully profiled – in our experience they can result in significant performance penalties if not performed correctly – we provide an event immediately prior to the start of the copying operation and an event to notify the profiling/debugging tool that the copy has completed. We have used these events extensively to assess the performance of data movement traffic over the NVLINK high-performance CPU-to-GPU links [10][25] found on a number of recently installed pre-exascale platforms.

Use of data allocation events requires no additional code by the application programmer as the runtime is able to use existing allocation/deallocation and copy routines to generate the events of interest.

3 Tools for Profiling Kokkos Applications

In order to guide the design of the profiling hooks in Kokkos and provide basic functionality to developers, a small suite of basic tools have been created. This suite runs on all of the systems supported by Kokkos and additionally acts as an exemplar for the developers of larger tool suites such as TAU [21], OpenSpeed-Shop [20], Score-P [15] and HPCToolkit [24] amongst others.

3.1 Kernel Profiling

The most frequently used tool in the suite is a simple kernel timer which intercepts all Kokkos kernel dispatch and application structure hooks. This allows it to time each kernel running in an application as well as any application sections or regions. The output provided to the developer is a mapping of the kernel name (provided by the developer or, if none is provided, the C++ mangled name), total time taken in the kernel, the number of calls made to the kernel and then a division of these to produce an average timing. Each MPI rank in a parallel application is profiled independently and the output written to a per-process file for post execution analysis.

3.2 Parallel Time Stack Profiling

A more sophisticated variant of the kernel profiling is an MPI-aware parallel time-stack profiler. In this tool application regions and kernels are tracked in a similar manner to conventional call-tree profiling (*i.e.* nesting is preserved). Application sections are also tracked but because overlapping is permitted, nesting is not preserved. The output provided to the developer at the end of the run is given as a nested breakdown of execution time with inclusive and exclusive timing. Because MPI is used, we are also able to provide a measure of imbalance/variation across MPI ranks, giving insight into any specific rank variance that an application may experience.

3.3 Memory Event/Heap Profiling

The Memory Event profiling tool allows users to track their Kokkos data structure allocations on a per-device basis. Every data allocation and deallocation is tracked, providing a Kokkos heap-usage-by-time profiling capability. We track each device in the Kokkos application independently with a separate heap. This allows application developers to inspect how much of resources such as GPU memory as being used during execution and at what times the greatest pressure is being placed on them.

4 Conclusions

The diversity of computer architectures is growing, not least because of a greater push for compute-dense accelerators to hit Exascale-class performance levels, but also, because of a much greater variety of workload drivers in the broader computing ecosystem. The challenge of re-programming large-scale, complex applications for each architecture has been addressed, at least in some parts of the HPC community, by the adoption of C++ abstraction frameworks such as Kokkos, RAJA, Agency and Thrust. While these frameworks have been largely successful at providing the portability, and mostly the performance, required, their complex construction, often using aggressive templating in C++, has frustrated the

use of existing profiling and debugging tools. In a number of cases, such tools have either given incorrectly attributed results or provided such complex outputs that they have become practically unusable for all but the most expert and persistant of users.

Anecdotal evidence from our use of Kokkos in the practical setting of Sandia's large-scale engineering and scientific computing workload has been that capable profiling tools in particular, are a prerequisite to handling the complex task of porting codes to next-generation architectures. Therefore, a solution to address the gap between portable C++-based abstractions, and the profiling tools offered by vendors is required.

In this paper we have provided a brief overview and discussion of KokkosP - a suite of hooks that profiling and debugging tools are able to register against to receive an event stream from an executing application written to use Kokkos for on-node parallelism. Event classes including parallel kernel dispatch, data allocation/de-allocation and varying forms of application code structure (to aid in more readable output). The hooks are intended to be compiled in to every application using Kokkos and operate with virtually zero overhead when not in active use (*i.e.*, zero overhead when any particular event type is not registered to be received by a profiling tool). Our experience with using the profiling interface over the past fours years of Kokkos development is that this is largely met and that we are able to compile the hooks in routinely. Such an approach is appealing because if any particular run of an application experiences performance issues then profiling tools can be attached to that instance of the binary – no recompilation for profiling or debugging is necessary to utilize the event stream.

Our future work will utilize the Kokkos profiling hooks to provide further information to application programmers. We are working across the DOE's Exascale Computing Project to partner with tool developers looking to utilize the Kokkos profiling hooks (such as profiling tools who want to natively profile Kokkos kernels). In addition, we expect to utilize our system to provide continuous performance monitoring through introspection of small benchmarking runs during overnight testing. The collection of data from the hundreds of runs performed each night will be able to provide timely feedback to our developer community informing them of kernels/software packages which are gradually slowing down over time or which precise kernel is experiencing performance regressions in their most recent repository changes.

The Kokkos profiling system is an extremely simple, yet practical method of addressing the complexities of profiling complex, performance portable code. Because the hooks are uniform across each architecture platform they met the Kokkos project goals of a unified user experience regardless of specific system. At the time same, they allow significant flexibility, permitting profiling using vendor providing tools on any particular piece of hardware, or a collection of cross-platform tools which will work on any system.

5 Related Work

One of the most similar approaches to ours is the OMPD and OMPT [9], [8] interfaces. These build on work in [3] to add support for OpenMP debugging and profiling interfaces. These interfaces can then be used to provide a similar ability for profiling/debugging tools to connect to telemetry from the OpenMP-based runtimes. In keeping with the philisophy of Kokkos, our aim is to provide a simple, cross platform set of profiling hooks which can be used across programming models, platforms and compilers. OMPD/OMPT attempts to do the same but for a single programming model. These interfaces are now being deployed across OpenMP-complaint runtimes and have support embedded in vendor and open source tools. Similar such extensions are also being investigated for the OpenACC accelerator programming model. We expect through further work, OMPT may also be nested within KokkosP regions to provide refined profiling information.

6 Tool Availability

Kokkos Profiling hooks are built directly into the standard Kokkos runtime and require no user configuration/settings if the default build parameters are used. The profiling can be easily disabled in the event of compilation difficulties. We recommend that the profiling hooks are left on so that profiling/debugging is made easier once Kokkos is installed.

The basic Kokkos tools suite is available from the project Github repository at: https://github.com/kokkos/kokkos-tools. Bug reports and feature requests can be made using the standard Github pages.

Acknowledgements

Sandia National Laboratories is a multimission laboratory managed and operated by National Technology and Engineering Solutions of Sandia, LLC., a wholly owned subsidiary of Honeywell International, Inc., for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-NA0003525.

References

1. Bartlett, R.A.: Teuchos C++ Memory Management Classes, Idioms, and Related Topics, the Complete Reference: a Comprehensive Strategy for Safe and Efficient Memory Management in C++ for High Performance Computing. Tech. Rep. SAND2010-2234, Sandia National Laboratories (2010)
2. Bell, N., Hoberock, J.: Thrust: A Productivity-Oriented Library for CUDA. In: GPU computing gems Jade edition, pp. 359–371. Elsevier (2011)

3. Cownie, J., DelSignore, J., de Supinski, B.R., Warren, K.: DMPL: an OpenMP DLL Debugging Interface. In: International Workshop on OpenMP Applications and Tools. pp. 137–146. Springer (2003)
4. Dagum, L., Menon, R.: OpenMP: an Industry Standard API for Shared-Memory Programming. IEEE Computational Science and Engineering **5**(1), 46–55 (1998)
5. Edwards, H.C., Sunderland, D., Porter, V., Amsler, C., Mish, S.: Manycore Performance-Portability: Kokkos Multidimensional Array Library. Scientific Programming **20**(2), 89–114 (2012)
6. Edwards, H.C., Trott, C.R.: Kokkos: Enabling Performance Portability Across Manycore Architectures. In: Extreme Scaling Workshop (XSW), 2013. pp. 18–24. IEEE (2013)
7. Edwards, H.C., Trott, C.R., Sunderland, D.: Kokkos: Enabling Manycore Performance Portability through Polymorphic Memory Access Patterns. Journal of Parallel and Distributed Computing **74**(12), 3202–3216 (2014)
8. Eichenberger, A., Mellor-Crummey, J., Schulz, M., Copty, N., DelSignore, J., Dietrich, R., Liu, X., Loh, E., Lorenz, D.: OMPT and OMPD: OpenMP Tools Application Programming Interfaces for Performance Analysis and Debugging. In: International Workshop on OpenMP (IWOMP 2013) (2013)
9. Eichenberger, A.E., Mellor-Crummey, J., Schulz, M., Wong, M., Copty, N., Dietrich, R., Liu, X., Loh, E., Lorenz, D.: OMPT: An OpenMP Tools Application Programming Interface for Performance Analysis. In: International Workshop on OpenMP. pp. 171–185. Springer (2013)
10. Foley, D., Danskin, J.: Ultra-Performance Pascal GPU and NVLink Interconnect. IEEE Micro **37**(2), 7–17 (2017)
11. Hammarlund, P., Martinez, A.J., Bajwa, A.A., Hill, D.L., Hallnor, E., Jiang, H., Dixon, M., Derr, M., Hunsaker, M., Kumar, R., et al.: Haswell: The Fourth-Generation Intel Core Processor. IEEE Micro **34**(2), 6–20 (2014)
12. Heroux, M.A., Bartlett, R.A., Howle, V.E., Hoekstra, R.J., Hu, J.J., Kolda, T.G., Lehoucq, R.B., Long, K.R., Pawlowski, R.P., Phipps, E.T., et al.: An Overview of the Trilinos Project. ACM Transactions on Mathematical Software (TOMS) **31**(3), 397–423 (2005)
13. Jain, T., Agrawal, T.: The Haswell Microarchitecture - 4th Generation Processor. International Journal of Computer Science and Information Technologies **4**(3), 477–480 (2013)
14. Killian, W., Scogland, T., Kunen, A., Cavazos, J.: The Design and Implementation of OpenMP 4.5 and OpenACC Backends for the RAJA C++ Performance Portability Layer. In: International Workshop on Accelerator Programming Using Directives. pp. 63–82. Springer (2017)
15. Knüpfer, A., Rössel, C., an Mey, D., Biersdorff, S., Diethelm, K., Eschweiler, D., Geimer, M., Gerndt, M., Lorenz, D., Malony, A., et al.: Score-P: A Joint Performance Measurement Run-time Infrastructure for Periscope, Scalasca, TAU, and Vampir. In: Tools for High Performance Computing 2011, pp. 79–91. Springer (2012)
16. Messina, P.: The U.S. D.O.E. Exascale Computing Project - Goals and Challenges (February 2017) (Feburary 2017)
17. Nalamalpu, A., Kurd, N., Deval, A., Mozak, C., Douglas, J., Khanna, A., Paillet, F., Schrom, G., Phelps, B.: Broadwell: A Family of IA 14nm Processors. In: VLSI Circuits (VLSI Circuits), 2015 Symposium on. pp. C314–C315. IEEE (2015)
18. Pheatt, C.: Intel Threading Building Blocks. Journal of Computing Sciences in Colleges **23**(4), 298–298 (2008)

19. Sadasivam, S.K., Thompto, B.W., Kalla, R., Starke, W.J.: IBM Power9 Processor Architecture. IEEE Micro **37**(2), 40–51 (Mar 2017). https://doi.org/10.1109/MM.2017.40
20. Schulz, M., Galarowicz, J., Maghrak, D., Hachfeld, W., Montoya, D., Cranford, S.: Open—SpeedShop: An Open Source Infrastructure for Parallel Performance Analysis. Scientific Programming **16**(2-3), 105–121 (2008)
21. Shende, S.S., Malony, A.D.: The TAU Parallel Performance System. The International Journal of High Performance Computing Applications **20**(2), 287–311 (2006)
22. Sodani, A.: Knights Landing (KNL): 2nd Generation Intel Xeon Phi Processor. In: Hot Chips 27 Symposium (HCS), 2015 IEEE. pp. 1–24. IEEE (2015)
23. Sodani, A., Gramunt, R., Corbal, J., Kim, H.S., Vinod, K., Chinthamani, S., Hutsell, S., Agarwal, R., Liu, Y.C.: Knights Landing: Second-Generation Intel Xeon Phi Product. IEEE Micro **36**(2), 34–46 (2016)
24. Tallent, N., Mellor-Crummey, J., Adhianto, L., Fagan, M., Krentel, M.: HPCToolkit: Performance Tools for Scientific Computing. In: Journal of Physics: Conference Series. vol. 125, p. 012088. IOP Publishing (2008)
25. Thompto, B.: POWER9: Processor for the Cognitive Era. In: Hot Chips 28 Symposium (HCS), 2016 IEEE. pp. 1–19. IEEE (2016)
26. Zenker, E., Worpitz, B., Widera, R., Huebl, A., Juckeland, G., Knüpfer, A., Nagel, W.E., Bussmann, M.: Alpaka–An Abstraction Library for Parallel Kernel Acceleration. In: Parallel and Distributed Processing Symposium Workshops, 2016 IEEE International. pp. 631–640. IEEE (2016)