SCIPHI
Score-P and Cube extensions for Intel Phi

Marc Schlütter, Christian Feld, Pavel Saviankou, Michael Knobloch, Marc-André Hermanns, Bernd Mohr

Abstract The Intel® Xeon Phi™ Knights Landing processors offers unique features with regards to memory hierarchy and vectorization capabilities. To improve tool support within these two areas, we present extensions to the Score-P measurement infrastructure and the Cube report explorer. With the Knights Landing edition, Intel introduced a new memory architecture, utilizing two types of memory, MCDRAM and DDR4 SDRAM. To assist the user in the decision where to place data structures, we introduce a MCDRAM candidate metric to the Cube report explorer. In addition we track all MCDRAM allocations through the hbwmalloc interface, providing memory metrics like leaked memory or the high-water mark on a per-region basis, as already known for the ubiquitous malloc/free. A Score-P metric plugin that records memory statistics via numastat on a per process level enables a timeline analysis using the Vampir toolset. To get the best performance out of Intel® Xeon Phi™, the large vector processing units need to be utilized effectively. The ratio between computation and data access and the vector processing unit (VPU) intensity are introduced as metrics to identify vectorization candidates on a per-region basis. The Portable Hardware Locality (hwloc) [4] library allows us to visualize the distribution of the KNL-specific performance metrics within the Cube report explorer, taking the hardware topology consisting of processor tiles and cores into account.

M. Schlütter · C. Feld · P. Saviankou · M. Knobloch
Jülich Supercomputing Centre, Forschungszentrum Jülich GmbH, Jülich, Germany
e-mail: m.schluetter@fz-juelich.de; c.feld@fz-juelich.de; p.saviankou@fz-juelich.de; m.knobloch@fz-juelich.de

M.-A. Hermanns · B. Mohr
JARA-HPC, Jülich Supercomputing Centre, Forschungszentrum Jülich GmbH, Jülich, Germany
e-mail: m.a.hermanns@fz-juelich.de; b.mohr@fz-juelich.de

1 Introduction

Many-core architectures like the Intel® Xeon Phi™ provide opportunities and challenges for intra-node optimization of applications. The Intel® Xeon Phi™ Knights Landing (KNL) comes with a unique memory hierarchy and a 512-bit-wise vector processing unit. To gain the the full benefits of the new features, the user needs to understand how effectively an application makes use of the underlying hardware capabilities. The objective of the SCIPHI (Score-P and Cube extensions for Intel Phi) project has been to incorporate this knowledge in the Score-P and Cube tools and provide it to their users.

Figure 1 shows a schematic image of the KNL chip. It highlights some areas of interest for the SCIPHI project, specifically the memory topology and tiled hardware layout.

A KNL chip consists of 38 uniform tiles [16], of which at most 36 are enabled. Each tile comes with two Silvermont cores – supporting up to four hyperthreads – and two Advanced Vector Extensions, known as AVX-512 [7] vector processing units (VPU). The high number of VPUs and the lower clock frequency of the Silvermont cores, compared to recent Xeon CPUs, makes vectorization not just an opportunity, it becomes a necessity for efficient node usage.

Besides the AVX-512 vector units, KNL is special with regards to memory. It comes with two types of memory, conventional DDR4 SDRAM providing high capacity and MCDRAM (High-Bandwidth Memory) providing high bandwidth. There are eight MCDRAM modules integrated on package, providing a total of 16 GB high bandwidth memory. These devices come with their own memory controller, providing a total bandwidth of more than 450 GB/s (Stream Triad [16]). The DDR4 SDRAM memory, on the other hand, is connected via two controllers, serving three channels each. The maximum capacity is 384 GB and the bandwidth can reach up to 90 GB/s.

The memory can be configured at boot time in one of three modes. In *cache mode*, MCDRAM serves as a cache for DDR4 SDRAM. In *flat mode*, MCDRAM is treated as standard memory in the same address space as DDR4 SDRAM; the *memkind* library[1] – a heap manager built on top of *jemalloc*[2] – allows for MCDRAM heap allocations via the *hbwmalloc* API. *Hybrid mode* is a combination of cache and flat mode, where a portion of MCDRAM serves as cache while the remainder can be use a standard memory. Cache friendly applications are likely to benefit from *cache* mode. There are applications though that might benefit from explicit MCDRAM memory management in *flat* mode.

In this paper we present extensions to the scalable performance measurement infrastructure for parallel codes *Score-P* [11] and the performance report explorer *Cube* [17] with regard to the special memory and vectorization features of the Intel® Xeon Phi™ Knights Landing processor. In particular, we implement means to track memory allocations and deallocations for both, DDR4 SDRAM and MCDRAM

[1] https://github.com/memkind

[2] http://jemalloc.net/

memory. In addition, a MCDRAM candidate metric on a per region basis is introduced. With respect to vectorization, we present VPU-related metrics that point the user to code regions that would benefit from vectorization. In addition we utilize the *hwloc* [4] library to visualize the distribution of the KNL-specific performance metrics, taking the hardware topology into account.

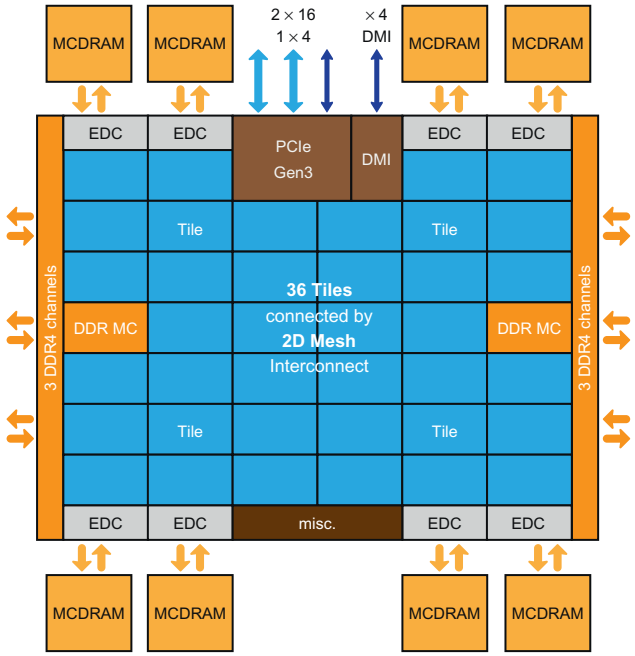


Fig. 1: Intel® Xeon Phi™ Knights Landing architecture [16]

The paper is structured as follows. Section 2 contains a survey of related work. Sections 3 and 4 focus on the two main topics, memory analysis and vectorization support. Section 5 presents the changes and requirements for the Score-P measurement work-flow that result from the previous sections. In Section 6 we present the Score-P hardware topology visualization in Cube using the example of the KNL architecture. We close with a conclusion and an outlook of future extensions in Section 7.

2 Related Work

Investigating memory usage, performance analysis tools, such as Score-P [11], TAU [19], and Vampirtrace [9], follow their call-path oriented approach in assigning measurement data to code regions. Jurenz et al. [8] highlight different methods

for querying memory data and how they are used in Vampirtrace. In contrast, tools, such as Intel VTune [1] or HPCToolkit [3], provide a data-centric perspective. For example, Liu et al. [12] present an extension to the HPCToolkit, using instruction-based sampling to record memory usage through relevant events, pinpointing to an effective memory address.

Vectorization and memory related metrics use hardware counters as basis for calculation, accessed either directly or through third party software. PAPI [5] provides such access to hardware counters, in the KNL case also to the node level uncore counters, which are required for the analysis of memory accesses and bandwidth. As an alternative, the LIKWID tools [20] allow similar use of counter information on the KNL, including access to shared counters. Wylie et al. [22] highlights that depending on the type measuring multiple hardware counters at the same time can be an issue and proposes a solution through multiple manual measurements. Reinders et al. [16] suggest metrics and thresholds for the KNL architecture, which should serve as guidelines for the user during optimization.

Scalasca version 1.x [14], still using its internal measurement system, provided topologies for Cube, while such support in Score-P is scheduled for a future release.

3 Memory Analysis

The availability of high bandwidth MCDRAM on the Intel® Xeon Phi™ Knights Landing provides unique opportunities as well as challenges for the application developer. The potential increase in bandwidth when using MCDRAM is counterbalanced by the reduced capacity available. With only 16 GB MCDRAM, compared to the maximum of 384 GB DDR4 SDRAM, the use of MCDRAM has to be managed carefully. If the KNL is booted in *cache mode*, the developer has no direct access to the MCDRAM, but the system uses this up to 16 GB as an additional, transparent cache for DDR4 SDRAM data. However, if the node is booted in *flat mode* – and to a degree in *hybrid mode* – the developer is responsible to explicitly manage allocations from MCDRAM. In these modes, at least one additional NUMA node is present, depending on the selected cluster mode. Allocations and deallocations can be managed in several ways. One way is to utilize the *NUMA Control utility* (numactl). This method requires no code changes but the allocations must completely fit into MCDRAM, as all memory is allocated from this NUMA node, including the data segments and stack. Another method without the need of code changes is the use of *autohbw*, which comes with the memkind package. This library allocates memory chunks of a certain size range transparently from MCDRAM. Yet another way to handle allocations into MCDRAM is provided by the *hbwmalloc* API that also comes with the *memkind* library. This API offers replacements for glibc's malloc routines in C and C++ and the FASTMEM directive for Intel Fortran. This method provides the maximum amount of control but comes at the cost of mandatory code changes.

Given these three methods to explicitly manage MCDRAM and the limited amount of high bandwidth memory, the developer is faced with the question of *what* and *how much*, if not all, to allocate from MCDRAM. By providing relevant information about memory usage, tools can support the user in the decision-making process.

3.1 DDR4 SDRAM and MCDRAM usage

If the working set fits into the 16 GB of MCDRAM, the easiest way to utilize the high bandwidth memory is to use numactl. The command `numactl -H` gives us the available NUMA nodes and with `numactl -m 1 ./application` we start a program that allocates all memory from NUMA node 1, which maps to MCDRAM in flat/quadrant mode.

If the working set is larger than the available 16 GB, shifting all memory allocations to MCDRAM will fail. To determine the actual memory requirements of an application, one can observe fine-grained allocations by tracking calls to `malloc`, `free`, and similar functions[3]. A more course-grained approach is to monitor the output of `numactl`, `numastat`, or `getrusage`. The open-source measurement infrastructure Score-P [11] already provides the functionality to observe the fine-grained allocations by wrapping the necessary library calls. This allows to determine allocations and deallocations on a per-region and per-thread basis. It also points the developer to leaked memory, i.e., it shows allocations that haven't been deallocated. Last but not least it keeps track of the used memory's *high-water mark*, i.e., the maximum amount of memory allocated at a time and points the developer to the code region where this maximum was reached. These recorded memory metrics can be analyzed in CUBE's visualization of call-path profiles [17] and be timeline-visualized by the Vampir trace analyzer [10], and potentially any other tools that support the open Score-P output formats *cubex* and *OTF2* [6].

The memory allocation tracking works only for conventional dynamic memory allocations, i.e., allocations from DDR4 SDRAM, as `malloc` and friends don't allocate from MCDRAM. By observing these conventional allocations one can estimate the memory requirements of an application and determine if the entire working set would fit into the 16 GB MCDRAM. The recorded allocations also provide the entire set of candidates that might benefit from being moved to MCDRAM.

To also observe allocations, find leaks, and the high-water mark when explicitly working with MCDRAM, it seemed natural to extend Score-P to intercept the *hbwmalloc* API. In flat mode, this API allocates from MCDRAM and can be used as a drop-in replacement for the usual `malloc/free` set of functions. To track the hbwmalloc allocations, we added the following functions to the set of wrapped library functions, this way providing the same analysis opportunities as for con-

[3] `malloc,realloc,calloc,free,memalign,posix_memalign,valloc`

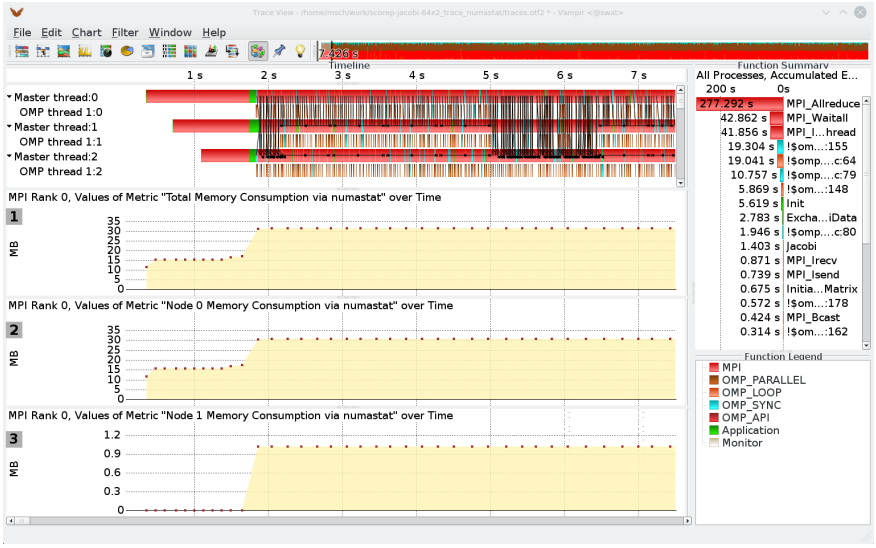
ventional allocations: `hbw_malloc`, `hbw_calloc`, `hbw_realloc`, `hbw_free`, `hbw_posix_memalign`, and `hbw_posix_memalign_psize`.

Tracking allocations and deallocations in such a fine-grained way can induce significant measurement overhead with regards to the Score-P-internal memory requirements to perform the tracking, in particular in cases with excessive numbers of heap memory operations. Furthermore, a high-water mark of memory consumption below 16 GB does not guarantee that the application will fit entirely into MCDRAM. This is due to the fact that memory is allocated on a per-page basis (usually 4K) on first touch. The first touch may happen way later in time than the `malloc` call and freed memory may not get reused. The actual memory consumption might be larger than reported by just tracking memory allocation and deallocation routines.

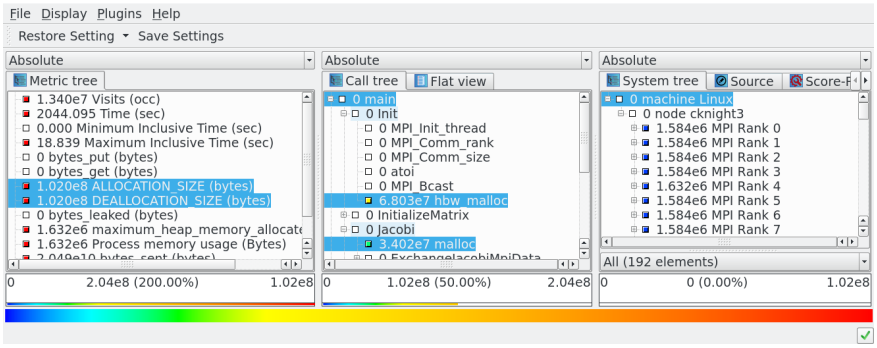
To get a more accurate measure of the used memory on a per-page basis we use the `numastat` command as a complimentary data source. It provides per-NUMA-node memory statistics for processes and the operating system. This allows not only for monitoring conventional memory allocations, but also for MCDRAM allocations in arbitrary cluster modes. Depending on the cluster mode – quadrant, SNC-2, or SNC-4, selected at boot time – a KNL chip in flat memory mode reports two, four, or eight NUMA nodes per quadrant, equally distributed among DDR4 SDRAM and MCDRAM. To monitor the evolution in time of these NUMA nodes with regards to memory, we implemented a Score-P metric plugin. Score-P metric plugins are (third-party) shared libraries that are linked to the measurement core in order to provide any kind of additional metrics. They are modeled after VampirTrace's [9] Counter Plugins [18]. Our numastat plugin is executd asynchronously, i.e., it is triggered at regular intervals within an extra thread, analyses the numastat output for the current process (`numastat -p <PID>`) and feeds the per-NUMA-node memory statistics into a Score-P timeline.

Figure 2 shows the visualization of such a timeline, generated by a simple Jacobi solver, using the Vampir tool set. For this demonstration some memory requests have been explicitly allocated from MCDRAM, using the hbwmalloc API. We can observe the DDR4 SDRAM and MCDRAM allocations in Vampir via an additional metric timelines alongside the master timeline. In our example, the numastat plugin records three additional timelines, the memory usage for each of the two NUMA nodes and the total memory usage. Here "Node 0"(2) corresponds to DDR4 SDRAM and "Node 1"(3) to MCDRAM.

The polling frequency of the numastat plugin can be chosen via an environment variable. If we are solely interested in the memory evolution over time we might accept the overhead introduced by a high polling frequency. But even with a low polling frequency in relation to the application run time the memory usage should be attributable to specifc phases of the application.



(a) Vampir Timeline



(b) Cube Profile

Fig. 2: Memory consumption over time as reported by numastat, separated into DDR4 SDRAM (Node 0), MCDRAM (Node 1), and Total in the Vampir timeline, alongside with a corresponding Cube report with activated hbwmalloc tracking.

3.2 MCDRAM candidates

If the working set does not fit into the 16 GB of MCDRAM it might be beneficial to allocate parts of the working set datastructures into high bandwidth memory using the hbwmalloc API. But such a change in allocation does not always improve the runtime. For the CloaverLeaf [15] hydrodynamics mini-app, e.g., we see runtime reduction due to selective allocations into MCDRAM as compared to pure conventional allocations only for high overall DDR4 SDRAM bandwidth values,

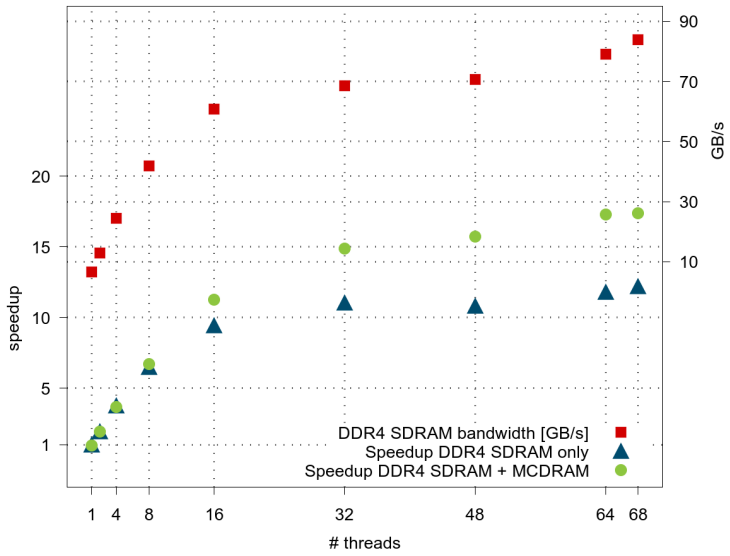


Fig. 3: CloverLeaf3D on a single KNL. Baseline for speedup is DDR4 SDRAM-only (blue) on a single thread. For DDR4 SDRAM + MCDRAM (green), some data structures were manually allocated into MCDRAM. DDR4 SDRAM bandwidth (red) is for the entire application, not individual kernels.

see Figure 3. To create this graph, we manually changed the allocation for bandwidth sensitve datastructures using the `FASTMEM` directive according to the manually found optimum in [16]. We see improvements over the DDR4 SDRAM-only variant for higher thread counts that correspond to higher overall bandwidth. To obtain the bandwidth, we counted the number of read and write accesses to DDR4 SDRAM (`UNC_M_CAS_COUNT.ALL`) with the help of PAPI [5], multiplied by the cache-line size and divided over time. It is important for the process to have exclusive acccess to the KNL-node as obtaining the memory accesses is done via the uncore counters, which provide data for the entire node only, not for individual processes or threads. Hence, this analysis does not work, if the KNL-nodes is shared among jobs or users.

With Score-P, we measure the bandwidth values per code-region outside of OpenMP parallel regions, due the given uncore counter restrictions. Depending on the application, there might be a lot of code regions that show a *high* bandwidth value. To find the most bandwidth sensitive candidates among these regions, we need to sort them by their last-level cache-misses (LLC). This gives us the *MCDRAM candidate metric* per code region, as shown in Figure 4. We derive the MCDRAM candidate metric, i.e., we sort the high bandwith callpaths by their last-level cache misses, in the Cube plugin *KNL advisor* (see also 5.2). As input we use the PAPI-measured access counts for each DDR4 memory channel and the PAPI-

measured LLC counts. We take care of measuring the memory accesses only per-process while running exclusively on a single KNL node.

As Score-P and Cube purely work on code regions, the MCDRAM candidates are also code regions. As a drawback, if a candidate code region accesses several data structures, we cannot point to the most bandwidth sensitive structure. Vtune [1], HPCToolkit [3][12] or ScaAnalyzer [13] might provide more detailed insight.

In addition to this drawback, the above approach is not generally applicable for tools as accessing counters from the uncore requires priviledged access to a machine, either by setting the paranoia flag or by providing a special kernel module. On production machines, this access is, for security reasons, often not granted. This does not only apply to memory accesses, but to all uncore counters.

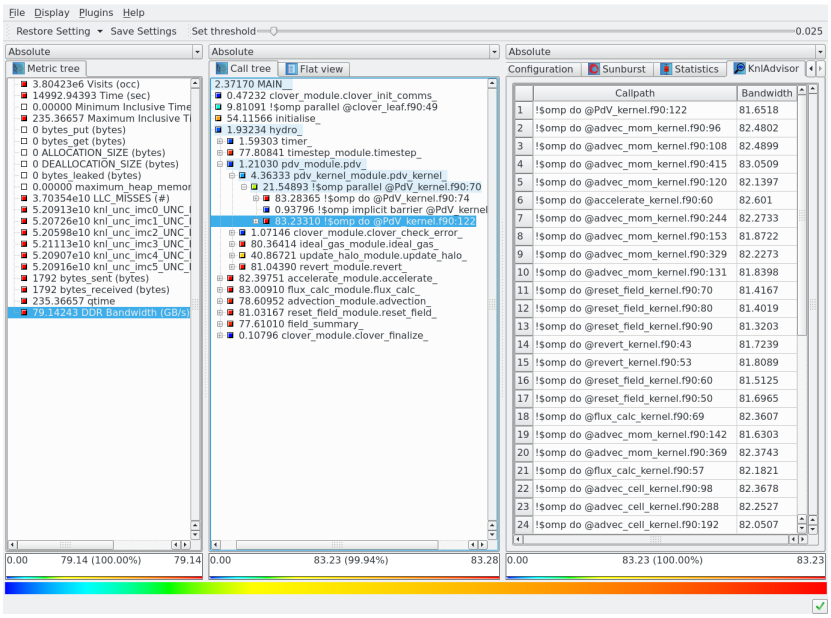


Fig. 4: Candidates for allocations in HBM: Highest bandwidth regions are sorted by last level cache misses.

4 Vectorization Assistance

In this section we focus on user support for vectorization, the second extension area of the SCIPHI project. Specifically, we investigate loops with regard to their degree of vectorization and offer suggestions for optimization candidates. This required hardware counter measurements, obtained in multiple runs, due to the limited num-

ber of available counter registers. In the context of counter measurements this is not unusual for the Score-P work-flow. The suggestion of specific optimization candidates on the other hand is a deviation from the standard Score-P metric semantics. The Score-P metric concept operates on the actual value of a metric (in absolute or relative terms) and analysis sometimes requires implicit information, e.g. if a higher value is worse than a small value. This approach leaves the decision about the relevance of a metric value of a certain call-path to the user. They need to judge the severity of an issue based on the knowledge of the hardware architecture, the source code, the input data, the use case, or even external parameters. Providing a generic set of thresholds, deciding if a metric value is problematic, is a hard problem in general, as too many parameters are involved, some outside the scope of the performance analysis tool. In the case of vectorization assistance we used the cooperation with Intel® to investigate the use of explicit knowledge about the architecture for providing such thresholds in that limited context. In the following we describe the metrics we focused on and the challenges they pose for the Score-P work-flow and analysis.

4.1 Metrics

Before we can generate the candidate lists for vectorization optimization in Section 5.2, we need to define the metrics that form the basis for the selection. We focus on the three metrics, that are listed in Table 1. The first metric calculates the computational density, i.e. the number of operations performed on average for each piece of loaded data. The `L1 compute to data access ratio` can be used to judge how suitable an application is to run on the KNL architecture. Ideally, operations should be vectorized and each datum fetched from L1 cache should be used for multiple operations. Table 1 shows the formula as number of vector operations vs. the number of loads seen by the L1 cache. Similar to this, the `L2 compute to data access ratio` is calculated as the number of vector operations against the loads that initially miss the L1 cache. While the L1 metric is critical in estimating a codes general suitability, the L2 metric is an indicator whether the code is operating efficiently. The thresholds, as listed in table 1, are considered the limits where an investigation into the code section's vectorization would be useful. These limits are based on recommendations of Intel® [16] for the KNL architecture and while these hold true for most applications running on KNL, they are only guidelines and should be applied with care.

An additional metric, the `VPU intensity`, offers a rule of thumb on how well a loop is vectorized, calculating the proportion of vectorized operations on total arithmetic operations. This metric should be applied only to small pieces of code and certain non-arithmetic operations, such as mask manipulation instructions, are counted as vector operations, which can skew this ratio.

Table 1 defines the metrics as ratios of hardware counters provided by the KNL architecture. These can be accessed in Score-P through the PAPI metrics interface

Metric: L1 Compute to data access ratio **Threshold:** <1
`UOPS_RETIRED.PACKED_SIMD` / `MEM_UOPS_RETIRED.ALL_LOADS`

Metric: L2 Compute to data access ratio **Threshold:** $< 100*$ L1 Compute to data access ratio
`UOPS_RETIRED.PACKED_SIMD` / `MEM_UOPS_RETIRED.L1_MISS_LOADS`

Metric: VPU intensity **Threshold:** <0.5
`UOPS_RETIRED.PACKED_SIMD` /
(`UOPS_RETIRED.PACKED_SIMD` + `UOPS_RETIRED.SCALAR_SIMD`)

Table 1: Vectorization metrics and their thresholds

and can measured at a call-path level on each thread. To calculate all derived metrics, multiple native hardware counters have to be recorded. Since the KNL architecture provides only two general purpose counters per thread, multiple measurements have to be used to obtain the full set of counters required. To facilitate a consistent user experience, the Score-P / Scalasca workflow has been extended to automate multiple runs with varying settings, which we describe in the following section.

5 Measurement Work-Flow

The analysis workflow for users of Scalasca comprises (1) instrumentation, (2) measurement, and (3) result examination. The analysis options presented in the previous sections require an adaption of this workflow, as multiple measurement runs need to be conducted before the results can be examined. Additionally, the results of the individual measurements need to be merged before examination to provide a holistic view across all of the different measurements. Furthermore, Cube needs to compute possible optimization candidates based on the unified results in an additional analysis step. To retain usability, we adapted the Scalasca toolset to automate the necessary measurements and post-processing steps.

5.1 Multi Run

The SCIPHI workflow needs multiple measurement runs, as the required hardware counters cannot be obtained by a single measurement. It is a known limitation of hardware performance counters that only specific combinations of counters can be combined in a single measurement [22]. The counters required for the analyses in SCIPHI need to be obtained in multiple measurements. So while the changes to the measurement work-flow are driven by the specific use case of SCIPHI, the generic implementation of the workflow adaptation also benefits other hardware counter measurements. Further benefiting scenarios are the quantification of variations in measurement, or the verification of statistical stability of results.

We implemented this adapted workflow as part of the Scalasca convenience command `scan`. As `scan` already automates the run of multiple execution steps—measurement and subsequent trace analysis—it is the natural target for also orchestrating multiple profile measurements. The basic mechanism of different execution settings per measurement is using environment variables. In principle, this allows arbitrary changes to the execution environment of each of the different runs, however, for the specific case of the KNL analysis the parameters have to be chosen carefully as the ability to merge the single run results depends on the similarity of the application runs. The new mode of multiple runs is controlled by input parameters of `scan`. The user has to specify the number of runs and the sets of variables for each run. Because the variables parameter can have a wide range of complexity, two ways of specification are offered: (1) a string combining all setting and (2) a configuration file.

Using a single string to specify the individual runtime parameters is convenient only for small parameter sets. Parameters within the string can be separated by two types of separators: `%` as run separator and `|` as key-value pair separator. An example for multiple key-value pairs is given below:

```
SCOREP_METRIC_PAPI=PAPI_TOT_CYC|SCOREP_TOTAL_MEMORY=33M
```

Such strings can be concatenated with an additional `%` between settings for each run, where the *i*-th sub-string indicates the environment settings for the *i*-th run. Empty sub-strings, indicated by `%%`, specify runs without special run configuration. If the configuration string contains less sub-strings as there are runs configured, `scan` uses as many of the sub-strings for its runs as available and assumes any missing sub-string at the end to be empty.

Using a configuration file is more practical when using a large number of variables or runs. A valid configuration file is a text file with one variable definition (key-value pair) per line and lines starting with a `%` as run separators. Analog to shell style comments lines starting with # as well as empty lines are ignored.

In either case per-run variables can only be used if they are not already set to a value in the current environment enabling the use of global and local variables and a strict separation of both. With these changes `scan` creates a single experiment directory with sub-directories containing the numbered results of the individual steps.

5.2 Cube tools

Given successful measurements, further analysis is performed by Cube. In recent releases, the Cube GUI has been extended by a flexible plugin API [17], which allows the easy addition of new capabilities through feature-specific plugins. In the context of this work, Cube needed to be extended in two aspects. First, the management of multiple measurement results per run and their combination into a single unified result. Second, the analysis generating optimization suggestions based on the criteria

presented in the Section 4. Given their different nature, these aspects are supported through two separate plugins: *Spotter* and *KNL Advisor*.

The Spotter plugin manages the creation of a single Cube profile from the multiple measurements found in the measurement archive. It scans a given directory and merges any found Cube profile to the joint profile one after the other. Any new metric found in a single profile is added to the joint Cube profile. This way, all metrics existing in any of the profiles will be present in the joint Cube report containing all partial counter recordings. During the merge process metrics existing in all profiles like `time` are replaced by the last instance in the merge chain. Therefore, a measurement run without additional counters can be used to provide low overhead time information as suggested in Section 5.1. With the complete set of base metrics Cube can calculate the derived metrics described in Section 4.1. As counters from possibly different measurement steps are combined, the user must consider the results with care and place them in context of the application's deterministic and repeatable behavior.

The KNL Advisor processes the joint profile generated by the Spotter plugin and applies the thresholds defined in Section 4.1. Based on these thresholds it generates a list of possible candidates for optimization referring to code location and triggering metric for each incident. As mentioned before, automatically applying thresholds bears the risk of creating false or irrelevant information. As most of the metrics generally make sense only for small code regions with focus on loop optimization, the calculation of the metrics and their candidates is restricted to loops and their children in the call tree. This still allows the user to focus on the most relevant parts from a vectorization point of view and their respective sub-trees while reducing the clutter noticeably. Loops are special code regions marked during instrumentation and recorded during measurement. Score-P allows for measurement of two different loop constructs: (1) OpenMP loops, which as automatically detected during instrumentation, and (2) user-defined loops, which are manually instrumented using Score-P's user-instrumentation API. To narrow down the list of candidates even further, they are classified for relevancy. The metrics used as the basis for the analyses presented in this work are all relative values. These ratios may indicate a high relative impact while their absolute impact on the overall runtime may be negligible. Therefore, relative *and* absolute impact needs to be taken into account. To honor user knowledge about the code and allow for application dependencies no specific, absolute cut-off threshold has been chosen here. Instead, a percentage slider based on a code region's runtime in relation to the total runtime determines the list of current suggestions, which is updated when the percentage is changed. This allows the user to apply different levels of detail to inspect and choose possible candidates for optimization.

Figure 5 shows an example of the KNL advisor plugin in use. On the right side the panel lists the suggestions for the current percentage setting, chosen from the slider in the toolbar above, as a list of call site and triggering metric. This list contains suggestions concerning the three vectorization metrics presented in Section 4.1. The left and middle pane show relevant metric and call-path information, respectively, for the currently selected incident. Additionally, the call tree also highlights every

other incident of the same metric in green. When selecting a different incident the call tree and the metric selection will update their selections to the relevant view.

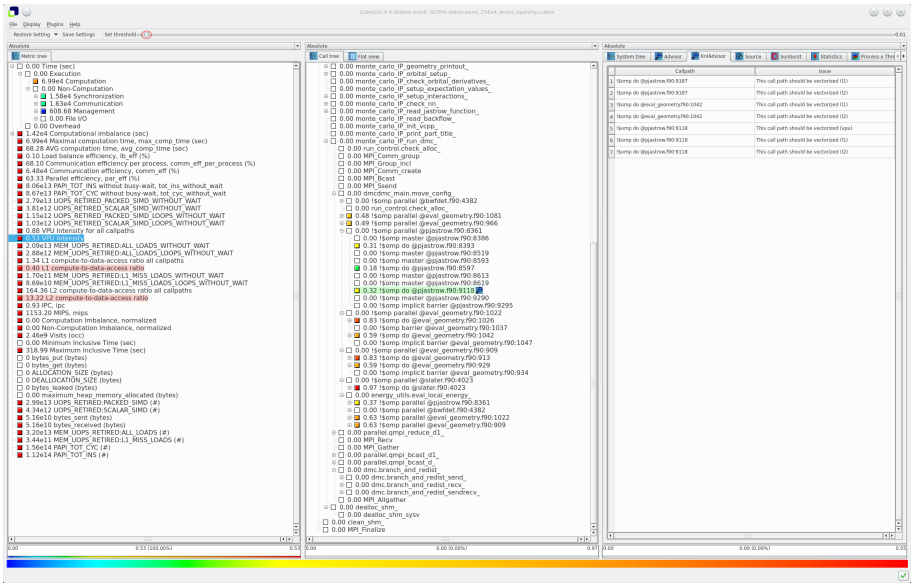


Fig. 5: KNL Advisor plugin

Generating optimization candidates and presenting them to the user in this way provides a intuitive starting point for the performance analysis. However, the user should always keep in mind that the suggestions given by the KNL advisor are only guidelines for optimization.

6 Topology Visualization

When examining the behavior of applications at large scale, it is important that the user understands how the individual processes and threads are distributed across the machine. A visual representation of the execution topology facilitates such understanding through an intuitive access to information about the execution context. Furthermore, in the case of many-core architectures, such as Intel KNL, the distribution on a single many-core node is of particular interest to gain the best performance for an application. The Score-P measurement infrastructure encodes this information as Cartesian topologies and saves it as part of the measurement profile to be visually explored using the Cube report browser.

Score-P Cartesian topologies map processes and threads to coordinates in multiple dimensional regular grids. While Score-P can record Cartesian topologies of ar-

bitrary dimensions, they are folded onto three dimensions for visualization in Cube. Score-P records meta information about the dimensions of the underlying topology and coordinates for each location, however, it is currently limited to CPU threads and processes excluding accelerators.

Currently, Score-P supports the generation of Cartesian topologies from four sources: (1) MPI Cartesian communicators, (2) proprietary query interfaces, (3) processes-by-threads matrices, and (4) user-defined topologies. Through the interception of the `MPI_Cart_create` call, Score-P automatically uses the information passed to the call to generate all necessary topology information and no further user interaction is needed. Each call `MPI_Cart_create` will create a separate topology. For platforms that provide an interface to supply coordinate information, such as the IBM Blue Gene or the Futjitsu K Computer, Score-P creates a hardware topology, mapping processes and threads with relation to the given dimensional information. For hybrid applications, Score-P automatically generates a two-dimensional topology with all processes in one dimension and their respective threads in the other dimension. The relationships visualized are similar to those shown in the system tree widget. However, the data is presented in a much more concise fashion, allowing a better overview of larger configurations. Finally, Score-P provides a user API to manually define Cartesian topologies. This allows users to record topology information as needed and enables the generation of application-level topologies that do not directly map to any other method above.

As topologies are regarded static through-out the execution of the application, topologies require pinning of threads to cores. Since the coordinate mapping is only done once per location Score-P will produce erroneous results if a thread migrates during the run-time. To keep a consistent mapping between threads and hardware locations, over-subscription isn't supported and application threads should be evenly spread between hardware threads.

An application run without explicit use of topologies in either MPI or user instrumentation will show the "Process x Threads" topology and, if supported, a hardware topology. Users can enable and disabled each source of topology information through the use of environment variables. This can become useful in cases where hardware topologies are problematic or for example when MPI Cartesian topologies are created in a loop iteration creating too many for practical use in the Cube result. As the coordinate information is only acquired once per location the run-time overhead is limited, however these meta data will increase the size of the measurement results depending on the number of locations. Therefore, depending on scale and memory requirements it can be helpful to reduce the memory overhead caused by topologies.

As part of the project, we investigated options to provide platform topology support for the KNL architecture. Since, the KNL architecture doesn't supply a specific interface to inquire this information directly during measurement, a more generic approach has been implemented in Score-P. Using the third-party *hwloc* library [4], Score-P now provides topology information for generic Linux systems and the KNL architecture in particular. Hwloc gathers node-level information about the core distribution and the memory hierarchy. It is an Open-MPI sub-project, mostly devel-

ht

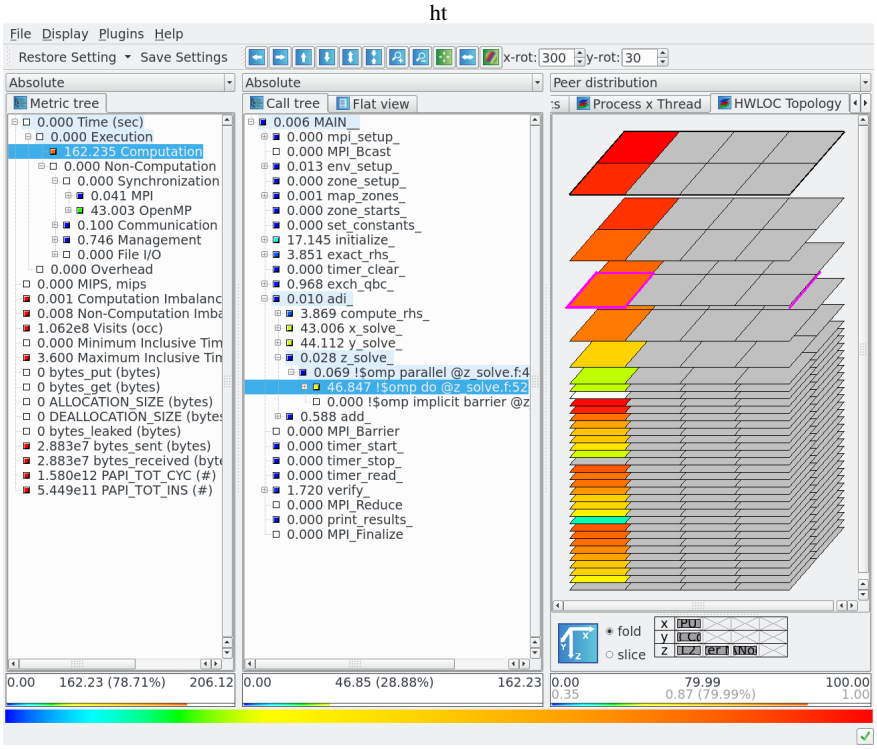


Fig. 6: Topology on a single node with one thread per core. Using one of four hardware threads per core and with two cores for each of the 36 tiles in z-dimension.

oped by the TADaaM team at Inria. As it provides only node-level information and no further information about the network structure is available, Score-P maps all nodes onto a single dimension. A further limit of hwloc is that some levels of the collected hierarchy, e.g., the L2 cache level of the KNL tiles, have only a logical numbering, which precludes any implicit assumption about a direct mapping to the respective hardware element. This limits the recording of direct neighborhood relations of the 2D tile map of the KNL chip, as shown in the schematic of Figure 1. Therefore, Score-P maps it to a single dimension instead.

In general, topology data is stored in both Cube profiles as well as OTF2 trace definitions. However, in the context of this project, we focused on the user benefits of visualization in Cube, as the core of the extensions for SCIPHI are based on profile data.

Figure 6 shows an example of a hardware topology for KNL obtained via hwloc during a measurement of the NAS Parallel Benchmark[21]. The Cube topology viewer plugin is available in the third pane of the Cube display, as an alternative visualization option to system tree, boxplot and others. Every topology created dur-

ing run-time is represented by an additional tab in this pane. With the help of the detachment mechanism for tabs, Cube allows the inspection of multiple topologies simultaneously. Most of the plugin space is used for a three dimensional display of the selected topology. The view can be manipulated in zoom and orientation either through mouse interaction or the toolbar. The user can select locations and query process and thread information on the selected coordinate. At the bottom of the display is a control interface that allows dimensions to be mapped to the three dimensions of the display, which becomes a necessity once you have more than three dimensions defined in your topology. There are two options to reduce the number of dimensions: folding and slicing. With folding the user can choose which input dimensions should be folded into one output dimension. In the case of slicing, three dimensions are selected to be shown completely and for the remaining dimensions single elements are chosen. Figure 6 shows a single node example, where the cluster dimension doesn't provide additional information and can be safely merged with the tile level of the KNL architecture.

That leads to an arrangement, where one (x,y) layer represents the two cores each with four hardware threads of a tile while the z-dimension shows the 36 tiles of a KNL node. Unused coordinates within topology are grayed out, as can be seen in this example where only one thread per core has been used.

To highlight the effect of different dimensions and their layout on the topology visualization, Figure 7 shows the detached view of a second example of using topologies on KNL. This second example shows a topology representation of an application measurement run on multiple nodes. The application is a Monte Carlo simulation called Casino, executed on 16 nodes of the CINECA Marconi cluster [2]. As the measurement now spans multiple nodes, the off-node dimension cannot be folded into the node layer without obscuring the node-level information. Figure 7 therefore keeps this dimension separate in the z dimension, while retaining the node-level dimensions mapped to the x and y dimension. That way one x,y plane represents one KNL cluster node with a line along the x dimension containing one tile, showing the hardware threads grouped by core.

As these two simple examples show, there is not a singular best way to arrange the dimensions in three dimension. Its usefulness depends on run-time configurations and user focus. Specifically in the case of the KNL architecture, the number of relevant dimensions also depends on the chosen cluster and memory modes as they influence the number of NUMA and sub-NUMA nodes. Furthermore, the examples demonstrate that topologies are a powerful tool to visualize run-time distribution of performance metrics, across large-scale measurements while having all locations visible without needing to scroll through lists of locations. This enhances the user ability to recognize patterns based on process placement and severity.

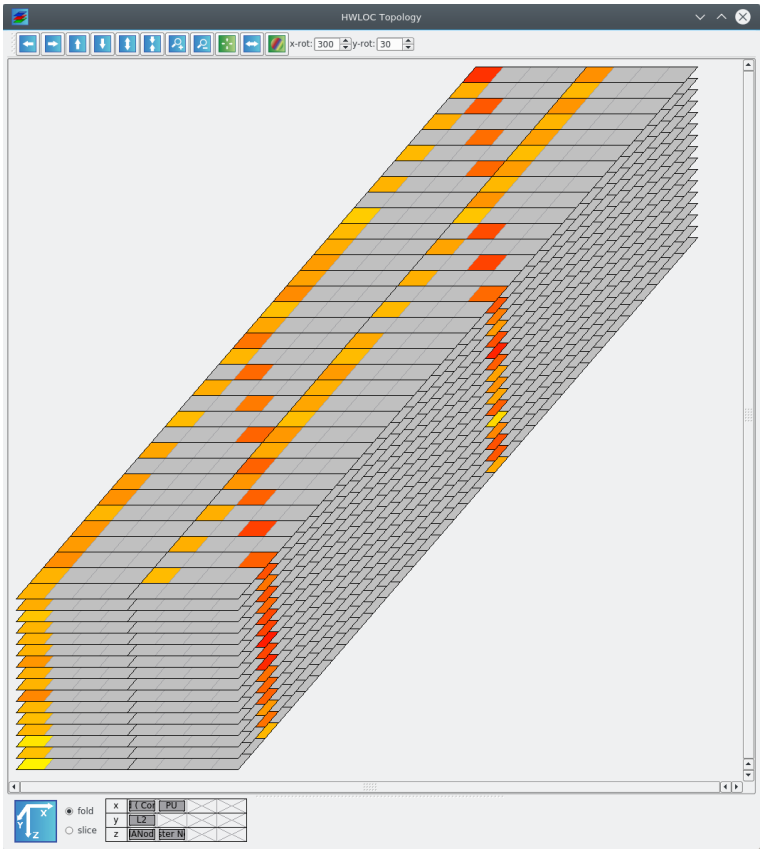


Fig. 7: View of the topology in detached state for an application run with a set of cluster nodes. On the x axis the two cores of a tile are arrayed in sequence with each of their four hardware threads grouped together. In combination with the 36 tiles in y direction each x,y plane represents a node. The 16 levels in z direction show the 16 used nodes for this run.

7 Conclusion

In this work we presented a set of extensions for Score-P that originated from a cooperation with Intel® and were therefore focused on the KNL architecture. With objective of improving the user experience and providing options for more in depth intra-node analysis, we focused on the important topics of memory hierarchy and vectorization. For the memory hierarchy, the explicit use of the two types of available memory, DDR4 SDRAM and MCDRAM, were of particular interest. The extensions highlight possibilities for tracking allocations and actual usage as a guideline to steer the developer to an efficient use of the fast MCDRAM. The work on

the vectorization assistance using knowledge about the specific target architecture presents a deviation from the standard Score-P metric definition. Instead of reporting just the severity of a metric, thresholds are provided as deciding factor to suggest optimization candidates.

The architecture dependent use case created possibilities for future work in broadening the focus of the presented extensions. The adaptations for multiple runs have a wide range of possible use cases, however for generic and integrated use in the standard work flow the limiting ramifications have to be addressed, in particular ensuring or at least checking the similarities between runs. Also, for a broader use of metrics from different measurements the scaling of potential different timings have to be considered, possible through the use of reference counters. Streamlining the work-flow might also contain an automatic way of source to source loop instrumentation to avoid the manual step of interaction.

8 Acknowledgements

We would like to express our thanks to Intel Corporation, who supported this work by the Intel Gift Grant “SCIPHI – Score-P and Cube extensions for Intel PHI”.

References

1. Intel® VTune™ amplifier. https://software.intel.com/en-us/intel-vtune-amplifier-xe.
2. Marconi, new CINECA tier-0 system. http://www.hpc.cineca.it/hardware/marconi.
3. L. Adhianto, S. Banerjee, M. Fagan, M. Krentel, G. Marin, J. Mellor-Crummey, and N. R. Tallent. Hpctoolkit: Tools for performance analysis of optimized parallel programs http://hpctoolkit.org. *Concurr. Comput. : Pract. Exper.*, 22(6):685–701, April 2010.
4. François Broquedis, Jérôme Clet-Ortega, Stéphanie Moreaud, Nathalie Furmento, Brice Goglin, Guillaume Mercier, Samuel Thibault, and Raymond Namyst. hwloc: A generic framework for managing hardware affinities in hpc applications. In IEEE, editor, *PDP 2010 - The 18th Euromicro International Conference on Parallel, Distributed and Network-Based Computing*, Pisa, Italy, February 2010.
5. S. Browne, J. Dongarra, N. Garner, G. Ho, and P. Mucci. A portable programming interface for performance evaluation on modern processors. *Int. J. High Perform. Comput. Appl.*, 14(3):189–204, August 2000.
6. Dominic Eschweiler, Michael Wagner, Markus Geimer, Andreas Knüpfer, Wolfgang E. Nagel, and Felix Wolf. Open Trace Format 2 - The next generation of scalable trace formats and support libraries. In *Proc. of the Intl. Conference on Parallel Computing (ParCo), Ghent, Belgium, August 30 – September 2 2011*, volume 22 of *Advances in Parallel Computing*, pages 481–490. IOS Press, 2012.
7. Intel Corporation. Intel architecture instruction set extensions programming reference. https://software.intel.com/isa-extensions.
8. Matthias Jurenz, Ronny Brendel, Andreas Knüpfer, Matthias Müller, and Wolfgang E. Nagel. *Memory Allocation Tracing with VampirTrace*, pages 839–846. Springer Berlin Heidelberg, Berlin, Heidelberg, 2007.

9. Andreas Knüpfer, Holger Brunst, Jens Doleschal, Matthias Jurenz, Matthias Lieber, Holger Mickler, Matthias S. Müller, and Wolfgang E. Nagel. *The Vampir Performance Analysis Tool-Set*, pages 139–155. Springer Berlin Heidelberg, Berlin, Heidelberg, 2008.
10. Andreas Knüpfer, Holger Brunst, Jens Doleschal, Matthias Jurenz, Matthias Lieber, Holger Mickler, Matthias S. Müller, and Wolfgang E. Nagel. *The Vampir Performance Analysis Tool-Set*, pages 139–155. Springer Berlin Heidelberg, Berlin, Heidelberg, 2008.
11. Andreas Knüpfer, Christian Rössel, Dieter an Mey, Scott Biersdorff, Kai Diethelm, Dominic Eschweiler, Markus Geimer, Michael Gerndt, Daniel Lorenz, Allen D. Malony, Wolfgang E. Nagel, Yury Oleynik, Peter Philippen, Pavel Saviankou, Dirk Schmidl, Sameer S. Shende, Ronny Tschüter, Michael Wagner, Bert Wesarg, and Felix 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.
12. Xu Liu and John Mellor-Crummey. A data-centric profiler for parallel programs. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '13, pages 28:1–28:12, New York, NY, USA, 2013. ACM.
13. Xu Liu and Bo Wu. Scaanalyzer: A tool to identify memory scalability bottlenecks in parallel programs. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '15, pages 47:1–47:12, New York, NY, USA, 2015. ACM.
14. Daniel Lorenz, David Böhme, Bernd Mohr, Alexandre Strube, and Zoltán Szebenyi. Extending Scalasca's analysis features. In Alexey Cheptsov, Steffen Brinkmann, José Gracia, Michael M. Resch, and Wolfgang E. Nagel, editors, *Tools for High Performance Computing 2012*, pages 115–126. Springer Berlin Heidelberg, 2013.
15. A. C. Mallinson, David A. Beckingsale, W. P. Gaudin, J. A. Herdman, J. M. Levesque, and Stephen A. Jarvis. Cloverleaf : preparing hydrodynamics codes for exascale. In *A New Vintage of Computing : CUG2013*. Cray User Group, Inc., 2013.
16. James Reinders, Jim Jeffers, and Avinash Sodani. *Intel Xeon Phi Processor High Performance Programming Knights Landing Edition*. Morgan Kaufmann Publishers Inc., Boston, MA, USA, 2016.
17. Pavel Saviankou, Michael Knobloch, A. Visser, and Bernd Mohr. Cube v4: From performance report explorer to performance analysis tool. *Procedia Computer Science*, 51:1343–1352, June 2015.
18. Robert Schöne, Ronny Tschüter, Thomas Ilsche, and Daniel Hackenberg. *The VampirTrace Plugin Counter Interface: Introduction and Examples*, pages 501–511. Springer Berlin Heidelberg, Berlin, Heidelberg, 2011.
19. Sameer S. Shende and Allen D. Malony. The tau parallel performance system. *Int. J. High Perform. Comput. Appl.*, 20(2):287–311, May 2006.
20. J. Treibig, G. Hager, and G. Wellein. Likwid: A lightweight performance-oriented tool suite for x86 multicore environments. In *Proceedings of PSTI2010, the First International Workshop on Parallel Software Tools and Tool Infrastructures*, San Diego CA, 2010.
21. Rob F. Van der Wijngaart and Haoqiang Jin. NAS Parallel Benchmarks, Multi-Zone versions. Technical Report NAS-03-010, NASA Ames Research Center, Moffett Field, CA, USA, July 2003. `http://www.nas.nasa.gov/Software/NPB/`.
22. Brian J. N. Wylie, Bernd Mohr, and Felix Wolf. Holistic hardware counter performance analysis of parallel programs. In *Proc. of the Conference on Parallel Computing (ParCo), Malaga, Spain*, pages 187–194, September 2005.