Interactive visualization of cross-layer performance anomalies in dynamic task-parallel applications and systems

Andi Drebes, Antoniu Pop, Karine Heydemann, Albert Cohen

Interactive Visualization of Cross-Layer Performance Anomalies in Dynamic Task-Parallel Applications and Systems

Andi Drebes and Antoniu Pop
The University of Manchester
School of Computer Science
Manchester, United Kingdom
Email: *first.last*@manchester.ac.uk

Karine Heydemann
Sorbonne Universités, UPMC Paris 06,
CNRS, UMR 7606, LIP6, France
Paris, France
Email: karine.heydemann@lip6.fr

Albert Cohen
INRIA and DI, École Normale Supérieure
Paris, France
Email: albert.cohen@inria.fr

***Abstract*—This paper studies the interactive visualization and post-mortem analysis of execution traces generated by task-parallel programs. We focus on the detection of performance anomalies inaccessible to state-of-the-art performance analysis techniques, including anomalies deriving from the interaction of multiple levels of software abstractions, anomalies associated with the hardware, and anomalies resulting from interferences between optimizations in the application and run-time system. Building on our practical experience with the performance debugging of representative task-parallel applications and run-time systems for dynamic dependent task graphs, we designed a new tool called Aftermath. This tool enables the visualization of intricate anomalies involving multiple layers and components in the system. It also supports filtering, aggregation and joint visualization of key metrics and performance indicators, such as task duration, run-time state, hardware performance counters and data transfers. The tool also relates this information to the machine's topology. While not specifically designed for non-uniform memory access (NUMA) architectures, Aftermath takes advantage of the explicit memory regions and dependence information in dependent task models to precisely capture long-distance and inter-core effects. Aftermath supports traces of up to several gigabytes, with fast and intuitive navigation and the on-line configuration of new derived metrics. As it has proven invaluable to optimize both run-time environments and applications, we illustrate Aftermath on genuine cases encountered in the OpenStream project.**

I. Introduction

Programming models based on dependent tasks are increasingly presented as one of the most successful approaches to unleashing the processing power of massively parallel general-purpose computing architectures [9], [16], [14], [18], [17], [7]. While these models provide means to expose parallelism, efficient exploitation of the hardware is particularly challenging as the performance of task-parallel programs depends on many aspects, ranging from static code optimizations by the compiler or manual data-layout transformations by the programmer to dynamic optimizations regarding the structure of the task graph, the order of task creation and interactions with the run-time system and the underlying hardware architecture. Identifying performance anomalies and finding their cause requires a detailed understanding of all of these aspects in general and the complex interactions between the software and hardware components involved in the execution in particular. For example, performance bottlenecks arise from the limited parallelism exposed in the application (i.e., inappropriate partitioning, granularity, or sequential parts), from improper load balancing across the machine's cores or from poor locality inducing a drop of sequential task performance. Since main memory in modern parallel systems is usually distributed over multiple memory controllers with non-uniform memory access (NUMA), interactions involving the memory subsystem are of particular interest during performance debugging.

Due to the dynamic behavior of task-parallel applications, very few performance bottlenecks can be practically detected through static analysis. Trace-based analysis, i.e., post-mortem analysis of a trace file with all relevant dynamic events recorded at execution time, is a common technique to overcome these limitations for performance debugging [15], [13], [1]. A visual representation of events and their relationships combined with static information (e.g., the system topology) provides the necessary insight for an accurate analysis, sorting causes and effects and distinguishing application-specific anomalies from inefficiencies of the run-time system. A major difficulty in this process is to correlate low-level information with high-level concepts of the programming and resource model. Many tools for trace-based analysis target distributed applications executing on systems communicating through message passing; as a result, they do not natively support performance analysis of task-parallel applications and run-time systems. In addition, most tools do not reflect NUMA in their resource models.

We present Aftermath, a tool for interactive, off-line visualization, filtering and analysis of execution traces of task-parallel applications and run-time systems with explicit support for NUMA. Aftermath has been used extensively within the OpenStream project [17], a task-parallel, data-flow programming model implemented as an extension to OpenMP, adding syntax to express task-level data-flow dependences. Arbitrary dependence patterns can be used to exploit task, pipeline and data parallelism. Aftermath has provided deep insight into interactions between the application, the run-time, the operating

system and the hardware. Its large set of performance analysis views and tools can be applied to a wide variety of parallel applications and run-time systems and is therefore not limited to a specific framework such as OpenStream. Multiple metrics and indicators can be displayed jointly, accelerating the discovery of significant correlations. For more complex relationships, Aftermath offers powerful filtering mechanisms and is able to match relevant information with the topology of the machine. Its graphical user interface is optimized for responsiveness, enabling rapid exploration of traces and fine-grained control of the degree of detail needed for the analysis.

Our contributions are threefold.

1) *Aftermath is the first performance engineering tool enabling the fine-grained analysis of memory transfers between dependent tasks.*
2) *Aftermath provides native support for NUMA systems, in particular allowing for the analysis and visualization of the locality of task-level memory accesses.*
3) *Aftermath allows the user to immediately identify correlations between program execution characteristics (e.g., slow execution phases, computational load imbalance, insufficient parallelism) and any type of run-time or hardware event (e.g., NUMA locality, performance counter, synchronization, communication, load balancing events).*

The paper follows the a use-case driven structure, engaging into concrete performance debugging scenarios and describing how Aftermath's tools and views support these. The performance anomalies, their visualization and characterization have been selected to illustrate the original problems that can be tackled with Aftermath. Section II provides an overview of the main features of Aftermath. The presentation of Aftermath's capabilities for performance debugging of task-parallel applications based on cases encountered in the OpenStream project is divided into three sections with increasing complexity. Section III presents performance anomalies related to the interaction of software layers, i.e., between the application, the run-time system and the operating system. Section IV focuses on inefficiencies related to NUMA. The discovery of correlations between performance indicators is illustrated in Section V. In Section VI, we provide details about the implementation of Aftermath, including the trace format and optimizations for rendering. Related work is discussed in Section VII, before we conclude in Section VIII.

II. Aftermath in a nutshell

Before presenting concrete use cases of performance analysis and debugging, we start with an overview of Aftermath's main features and graphical user interface.

A. Organization of the user interface

Figure 1 shows the main window of Aftermath during analysis of a trace file with its five primary interface groups.

1. The *timeline* shows the activity of each processor over time. The default mode displays the different states (e.g., executing a task, synchronizing, load balancing) of the worker thread on each core. Additional modes visualize different aspects of program execution and interaction with the hardware, such as memory locality or task duration. The timeline can be overlaid with supplemental information on the evolution of performance counters and specific discrete events (e.g., task creation, communication between workers).

2. A group of *statistical views* presents aggregate quantitative information for a user-selected interval from the timeline (e.g., a histogram showing the distribution of task durations, a text field indicating the average parallelism, a communication matrix indicating which cores and nodes communicate).

3. A set of *filters* allows the user to control the contents of the timeline and the statistical views (e.g., only tasks of a specific type, tasks whose execution duration is in a certain range or tasks that write to certain NUMA nodes).

4. *Detailed textual information* for a selected state and its enclosing task execution (e.g., task and state type, duration, the sources/destinations of data read/written by the task).

5. A menu for customizing *generators* of metrics derived from high-level events or metrics that combine existing statistical counters (e.g., average task duration, number of bytes exchanged between specific NUMA nodes, ratio of hardware counters, etc.), overlaid on the timeline.

B. Timeline modes

Located in the center of the user interface, the timeline component can be specialized to highlight specific aspects of the trace by selecting one of the five main modes:

1. The default *state mode* shows which states each worker thread traverses over time. These states correspond to the main activities performed by the workers, to execute the application or in the run-time, thus showing how much time is spent in each activity. Common states include task execution, task creation, broadcasts, synchronization or computational load balancing.

2. In *heatmap mode*, the timeline shows only task executions and encodes the relative duration of tasks with different shades of red (darker for longer tasks). For short, we refer to this visual representation as a *heatmap*. The duration of tasks is considered relative either to a user-defined interval or to the shortest and longest task execution currently displayed in the timeline. The tasks displayed can be filtered, e.g., to show only instances of a given task type; shades are configurable.

3. The timeline in *task type mode*, also called *typemap*, associates a different color to every task type (i.e., the work function executed by the task), thus visualizing which type of task each worker executes over time. For example, an application with three work functions, used respectively for tasks performing matrix multiplication, initialization and termination, has three task types, which might be rendered in blue, green and yellow.

4. When in *NUMA mode*, the timeline associates a color to each NUMA node and shows which nodes are targeted by memory accesses performed by the tasks executed by each worker over time. This information is derived from the addresses of memory accesses and information on data placement present in the trace. The graphical representations generated in NUMA mode are called *NUMA read map* or *NUMA write map* for the respective type of memory accesses.

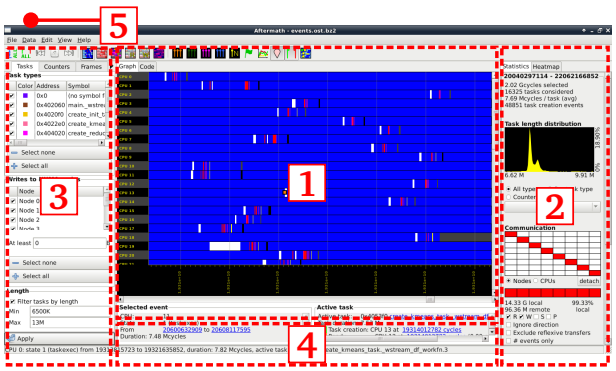


Fig. 1: Main window: timeline (1), filters (2), statistics (3), selected task/event information (4), derived metrics menu (5).

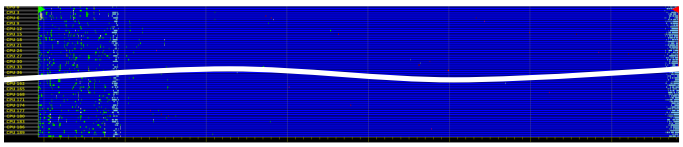


Fig. 2: *Seidel*: Run-time states

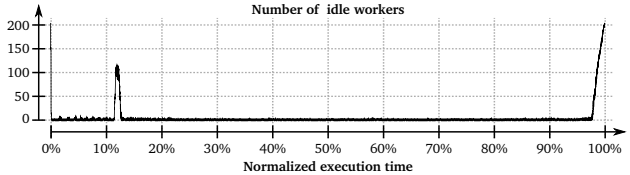


Fig. 3: *Seidel*: Number of idle workers

5. The *NUMA heatmap mode* combines both NUMA modes (read and write accesses) with information about the topology of the machine. The result is a view that indicates the average fraction of remote memory accesses per interval with different shades from blue (mostly local accesses) to pink (mostly remote). This view is especially important when the NUMA read and write maps do not show clear trends or relationships between the accessing and the targeted nodes.

III. Optimizing parallelism

To illustrate how Aftermath accelerates performance analysis and debugging, we present real use cases encountered while developing applications written in OpenStream. We analyze two applications: *seidel*, which implements a 2-dimensional stencil over a matrix of double precision floating point elements, and *k-means*, a data mining application. The test system for *seidel* is an SGI UV2000, composed of Xeon E5-4640 processors, with a total of 192 cores and 756 GiB RAM, distributed over 24 NUMA nodes connected through a Numalink 6 interconnect. The analyses of *k-means* have been conducted on a quad-socket AMD Opteron 6282 SE with a total of 64 cores and 64 GiB RAM, distributed over 8 NUMA nodes connected with HyperTransport 3.0 links.

We first analyze idle phases and the amount of available parallelism in *seidel*. We then illustrate the detection of execution phases and the distribution of long and short running tasks in the same benchmark. The impact of the parametrization on parallelism and run-time overhead is studied for *k-means*.

A. Detecting idle phases and tracking their origins

Figure 2 shows the timeline for *seidel* in state mode, indicating which states each worker traverses over time. Dark blue is associated with task execution and light blue is associated to the idle state, in which a worker engages in work-stealing. As dark blue dominates the graph, the majority of the time is spent on task execution. However, the are two distinct light blue vertical bands, one in the first quarter of the execution and the other at the end, indicating phases where a significant number of workers are idling. This hypothesis, based on a visual inspection of the timeline, is confirmed by analyzing the number of workers simultaneously in the idle state. Aftermath is able to generate a derived counter indicating the evolution of the number of workers that are simultaneously in any given state. By selecting the idle state for this counter, it is thus possible to obtain accurate information on the number of idle workers. To generate the samples for this counter, Aftermath first divides the execution into a user-defined number of intervals. For each interval it then determines for each worker how much time was spent in the specified state. Finally, it calculates the sum for all workers and divides the result by the duration of the interval. Figure 3 shows the average number of workers in the idle state for the trace. The peaks in these plots exceed half the number of cores and thus confirm the presence of the idle phases identified on the timeline.

Idle phases in task-parallel programs are either the result of poor computational load balancing by the scheduler or originate from insufficient available parallelism in the application. We postulate that the work-stealing load balancing strategy is sound, and focus instead on the latter possible cause, which can be validated by analyzing the application's *task graph*.

We define the task graph as a directed, acyclic graph where nodes represent tasks and edges represent inter-task data dependences. Figure 4 shows an example of task graph. The number of tasks at a given *depth* in the graph can serve as a metric to estimate the amount of parallelism available during execution in terms of tasks that are ready for execution. The depth of a task t is defined as the number of edges on the longest path from a task without any input dependence to t. In Figure 4, the longest path from t_0^0 or t_0^1 to t_2^2 has two edges, so the depth of t_2^2 is two. There are two tasks at depth zero (t_0^0 and t_0^1) and at depth one (t_1^0 and t_1^1), three at depth two (t_2^0, t_2^1, and t_2^2) and one task at depth 3 (t_3^0), corresponding to the available parallelism at each step of the computation.

As tasks can have different durations and might be created recursively and dynamically, the parallelism *effectively* available during execution can be lower than what the above metric indicates. The metric thus provides an upper bound for parallelism and is useful to detect bottlenecks that arise from inter-task dependences.

Due to the high number of tasks created in most applications, manual analysis of the task graph is usually infeasible. Therefore, Aftermath is able to reconstruct the task graph from the information present in the trace file and offers tools for

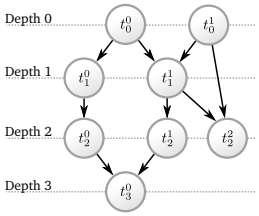


Fig. 4: Example of a task graph

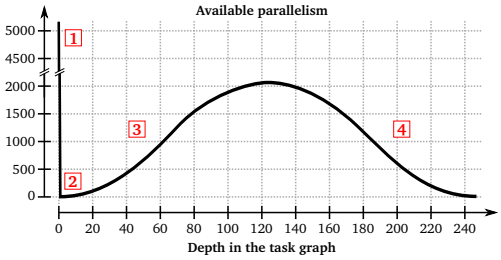


Fig. 5: *Seidel*: Available parallelism

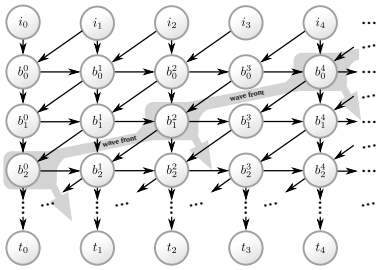


Fig. 6: *Seidel*: Excerpt of the task graph

automated analysis of the generated graph. Reconstruction is based on data dependences between tasks that can be derived from read and write accesses to memory regions shared by the tasks. For example, to reconstruct the task graph in Figure 4, the trace file must contain the write accesses by t_0^0 to memory regions read by t_0^1 and t_1^1, the write accesses of t_1^0 to the region read by t_2^0 and so on. Once the task graph has been reconstructed, Aftermath can be used to determine the depth of each task and to generate a graph showing the available parallelism as a function of the depth.

Figure 5 shows this graph for *seidel* using a $2^{14} \times 2^{14}$ matrix, processed in blocks of $2^8 \times 2^8$ elements. Four phases can be identified: (1) high parallelism with more than 5000 tasks during startup, (2) a sudden drop of parallelism to a single task, (3) increasing parallelism, and (4) declining parallelism. To illustrate the origins of these phases, Figure 6 shows a subset of the task graph of a one-dimensional version of *seidel*, similar to the actual task graph. The initialization tasks i_0 to i_n are ready for execution upon creation and belong to phase (1). The sudden drop in phase (2) results from the direct and transitive dependences of every task in the graph to b_0^0, except the initialization tasks. With the execution of b_0^0, a diagonal wave front is formed. Its size increases until it reaches the maximum at about a depth of 120, corresponding to phase (3). Phase (4) starts with the next step following the tasks on the wave front at its maximum size.

For a detailed analysis of particular tasks, Aftermath is also capable of exporting a subset of the task graph to a file in the DOT format [4]. The contents of this file can be visualized using the GRAPHVIZ package [8].

B. Detecting slow initialization

Following this analysis of the fundamental properties of the implementation of *seidel*, we now illustrate the analysis of anomalies related to dynamic events at execution time of the same benchmark, starting with a consideration of the task duration. A quick method to get an overview on this metric is to use the timeline in heatmap mode, in which different shades of red are used to visualize the task duration.

Figure 7 shows the heatmap for *seidel* with ten shades and the minimum and maximum durations set to 0 cycles and 50 Mcycles, respectively. Four distinct phases can be identified. The first phase in dark red, at the beginning of the execution, is composed of very long running tasks whose duration is close to or exceeding the configured maximum for the heatmap. The second phase corresponds to the phase with low parallelism identified in the previous section. As no task is executed during idle times, neither shade is used and the black and gray colors of the timeline's background become visible. The largest part of the figure is occupied by the third phase with few long running tasks and a majority of short running tasks rendered in white. In the fourth and final phase towards the end, available parallelism drops and the background becomes visible again.

These qualitative observations based on the heatmap can be confirmed by analyzing the derived counter indicating average task duration, shown in Figure 8. The peak of this plot coincides with the first phase with very long running tasks and the large plateau-like part occupies the interval associated to the third phase. Phases with low available parallelism do not appear in the graph: as the number of executing tasks never reaches zero for any interval, nor does the average duration.

The most relevant part for performance is the first phase with very long running tasks. To track the origin of the high average task duration in this phase, we first correlate this part of the trace with the task types by setting the timeline to typemap mode, shown in Figure 9. Pink color in this figure is associated to initialization tasks, while main computation tasks are rendered in ocher. The distinct pattern of pink and ocher indicates that the first phase is dominated by initialization tasks and that the majority of the tasks in the plateau phase are computation tasks. We conclude that the long running tasks identified earlier belong to the initialization.

Initialization tasks in the *seidel* benchmark are the first tasks that write to the memory regions used for data exchanges between tasks and thus trigger physical allocation of the associated pages. As these tasks do not perform any computations, it is thus likely that memory allocation is the cause for low performance. To confirm this assumption it is necessary to (1) verify that physical allocation takes place during initialization and (2) that this process has a significant impact on performance. Figure 10 shows the discrete derivative (difference quotient) of the aggregated system time as well as the discrete derivative for the application's resident size. This information has been collected in a separate trace, with statistics from the `getrusage` function with GNU extension, allowing per-worker measurements. A derived, aggregating counter created with Aftermath converts per-worker data into global statistics. The main reason for collecting `getrusage` statistics in a separate trace is our observation that concurrent calls to this function generate significant overhead. This also influences the termination of the initialization phase, which occurs earlier in

Fig. 7: *Seidel*: Timeline in heatmap mode

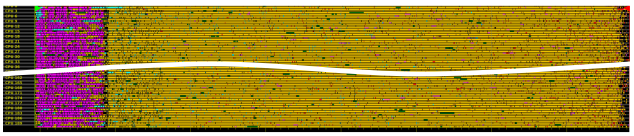


Fig. 9: *Seidel*: Timeline in typemap mode

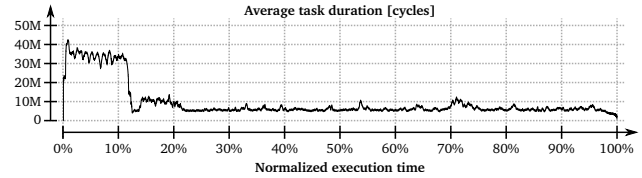


Fig. 8: *Seidel*: Average task duration

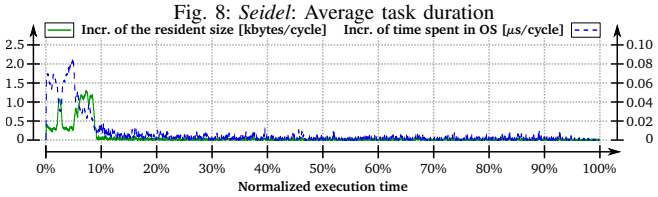


Fig. 10: *Seidel*: Increase of the system time / resident size

Figure 10 than in the earlier graphs. However, the figure shows that the memory footprint and the time spent in the operating system increase almost exclusively during initialization. This finally confirms our hypothesis about interactions between initialization tasks and the operating system.

C. Adjusting task granularity

The inefficiencies detected in the analyses of *seidel* result from characteristics of its implementation. In the following analysis, we focus on the impact of parameters chosen at execution time on available parallelism and run-time overhead for *k-means*, a data-mining benchmark that partitions a set of n multidimensional points into k clusters using a naive implementation of the K-means clustering algorithm.

Figure 11 shows a subset of the application's task graph for two iterations of the algorithm with. The set of points to be clustered is first divided into m blocks (8 blocks in the example). The size of these blocks as well as the number of blocks remains constant and does not depend on the relationship between points and clusters. In each iteration i, each block j is treated by a task k_i^j that calculates the distance of each point to the k cluster centers and associates the point to the nearest cluster center. At the end of each iteration, the application updates the cluster centers by calculating their barycenters based on the set of associated points. This is done by a reduction in a tree-like fashion by the tasks $r_{i,q}^s$. The root $r_{i,q}^0$ of this tree finally detects whether the algorithm has terminated, i.e., whether the number of points that have been associated to a new cluster is below a user-defined threshold. If the algorithm has not terminated, another iteration is necessary and the updated cluster centers are propagated to the tasks k_{i+1}^j of the next iteration by the tree of tasks formed by $p_{i,q}^s$.

As the block size determines the number of tasks, the amount of work per task and the memory footprint of each task, it must be chosen carefully. For huge block sizes, the available parallelism is low and only a subset of the cores can contribute to the computation, while tiny block sizes generate significant task management overhead. In the following experiments with *k-means*, we use a set of $4096 \cdot 10^4$ points, with 10 dimensions, to be grouped in 11 clusters.

Figure 12 shows the wall clock execution time as a function of the block size, ranging from 2500 points to 1.28 million points per block. Each bar represents the average for 50 executions and error bars indicate standard deviation. As expected, execution time is higher for very large or small blocks, with a minimum for a block size of 10^4 points.

The actual cause for this behavior can be verified by analyzing the activity of the workers executing the tasks. Figure 13 shows Aftermath's timeline in state mode for each block size. For a block size of 1.28 million points, the number of blocks is 32, which is far below the number of cores. This causes most of the workers to idle, showing a pattern with predominant light blue parts on the timeline. Decreasing the block size to 640 thousand points, as shown in Figure 13b, increases the number of blocks to 64, which provides enough tasks to keep each core busy during each iteration. However, differences in the tasks' execution time cause some workers to finish earlier than others. This leads to a characteristic alternating pattern of task execution and idle phases. Although this anomaly persists when further reducing the block size, its impact on execution time becomes lower. For block sizes smaller than 20000 points the pattern becomes imperceptible and the impact becomes negligible. However, excessive reduction of the block size to less than 5000 points leads to high task management overhead towards the end of the execution, resulting in idle phases at termination as shown in Figure 13j.

IV. Optimizing memory accesses

Let us now turn to NUMA-related performance anomalies. We investigate two traces of *seidel* obtained with two different configurations of the OpenStream run-time. The *non-optimized* configuration uses random work-stealing for computational load balancing and does take NUMA into account, neither for scheduling nor data placement. The *optimized* configuration exploits NUMA-specific information within the scheduler and memory allocator.

Aftermath offers three timeline modes for NUMA, shown in Figure 14, to intuitively visualize the locality of memory accesses with respect to NUMA. The first mode, in Figures 14a and 14b, shows the origin of the predominant amount of data *read* by a task. Every NUMA node is automatically assigned a different color and every task is represented on the timeline using the color corresponding to the NUMA node which contains the largest fraction of the data read by the task. Note

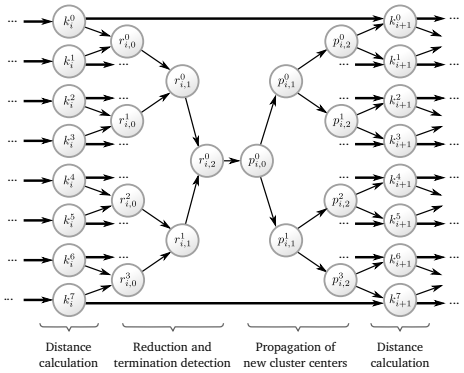


Fig. 11: Excerpt of the task graph for *k-means*

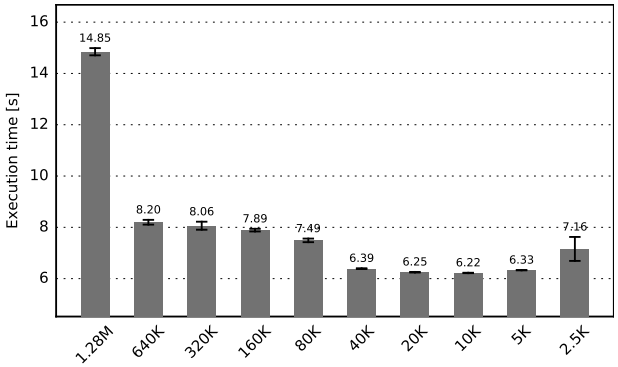
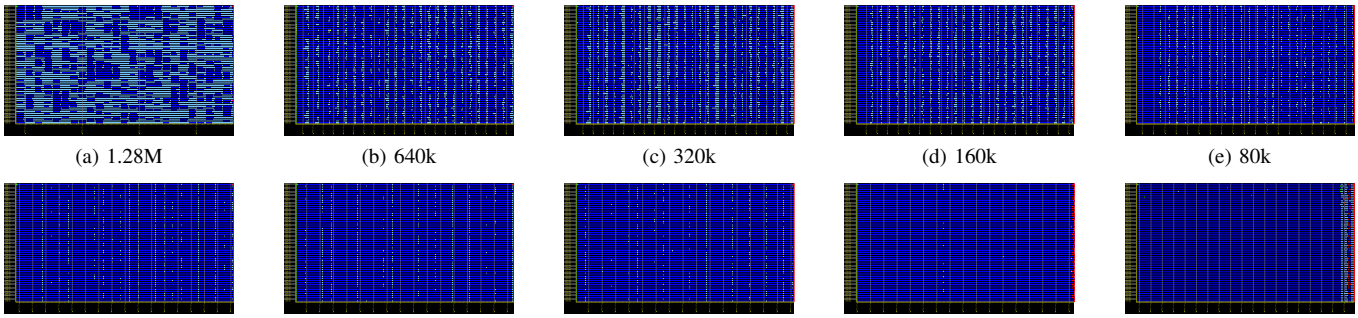


Fig. 12: *K-means*: execution time as a function of block size

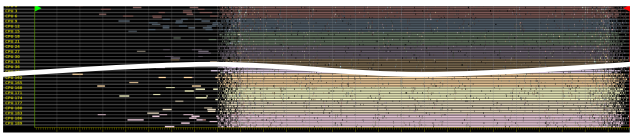


(a) 1.28M (b) 640k (c) 320k (d) 160k (e) 80k

(f) 40k (g) 20k (h) 10k (i) 5k (j) 2.5k

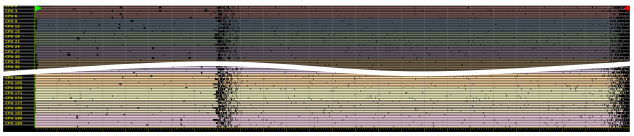
Fig. 13: *k-means*: Timeline in state mode for a block size of 2500 points up to 1.28 million points

(a) NUMA nodes targeted by read accesses, run-time not optimized

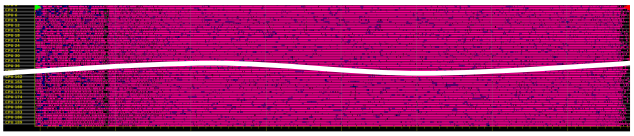


(b) NUMA nodes targeted by read accesses, optimized run-time

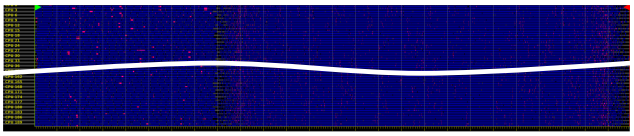
(c) NUMA nodes targeted by write accesses, run-time not optimized



(d) NUMA nodes targeted by write accesses, optimized run-time



(e) NUMA heatmap mode, run-time not optimized



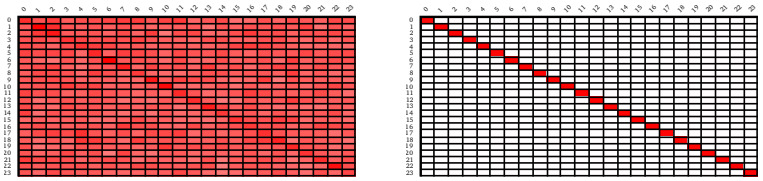
(f) NUMA heatmap mode, optimized run-time

Fig. 14: Seidel: locality of memory accesses

that the time scales for both traces have been normalized to the execution time: 7.91 Gcycles for the non-optimized and 2.59 Gcycles (3× speedup) for the optimized version. Due to the normalization, and a higher optimization impact on main computation tasks than on initialization tasks, the fraction of the execution time dedicated to initialization is visibly greater for the optimized version.

Figure 14a shows the timeline for the non-optimized execution of the application; the absence of any apparent pattern of colors in the timeline is characteristic of poor locality since tasks executing on a given node read data from all remote NUMA nodes. In contrast, Figure 14b shows the timeline of the optimized execution, where a distinctive pattern can be observed: nearly all tasks executing on a same NUMA node (adjacent cores on the timeline) have the same color. This band pattern means that a single NUMA node contains most of the data read by all of the tasks executed on a given node. Similarly, Figures 14c and 14d show the locality of write memory accesses, with the same intuitive visual confirmation of poor (14c) and good (14d) locality.

Finally, Figures 14e and 14f show the NUMA heatmap, an aggregated timeline mode where tasks are displayed with



(a) Not optimized (b) Optimized

Fig. 15: Communication incidence matrix for Seidel. Non-Optimized execution (left) and optimized execution (right).

shades ranging from blue (good locality) to pink (poor locality). Once again, this visualization allows to instantly identify the non-optimized (14e), and optimized (14f) executions.

An application-wide summary of memory locality and communication is provided as an incidence matrix, shown in Figure 15. This matrix represents the overall proportion of communication between each pair of NUMA nodes as shades of red (deeper for higher). Figure 15a shows this matrix for a non-optimized execution: deep red across the matrix means that each node communicates with every node in similar proportions, generating a high amount of memory traffic. The optimized execution, in Figure 15b, shows a very sharp diagonal which can be immediately interpreted as indicative of near-optimal locality. Indeed, there is no discernable red color present outside of the diagonal, which means that most of the data is accessed locally, within each node.

Aftermath also provides means to analyze data locality with respect to caches. Qualitative analysis is possible by visualizing hardware performance counter data for cache misses of each CPU on top of the time line. By letting Aftermath attribute counter data to tasks (e.g., calculate the number of cache misses for each task), it is possible to analyze cache locality quantitatively in built-in histograms or by exporting these values to a file for statistical analysis with external tools.

V. Correlating performance indicators

The previous sections described new analyses of performance anomalies enabled by Aftermath by means of specific trace information and original views examining a single metric at a time. However, other performance anomalies can only be discovered by correlating multiple metrics. In this section, we detail the analysis of such an anomaly: an inefficiency related to branch mispredictions, encountered while debugging the performance of *k-means*.

Figure 16 shows the task duration histogram for the computation tasks, having filtered out all auxiliary tasks, such as reduction and propagation tasks. Although computational tasks have similar workloads, their execution time is not uniform, as indicated by the peaks in the histogram. In addition, there is no clear relationship between task duration and machine topology: each core executes long and short running tasks during the entire execution, as shown in the timeline in heatmap mode in Figure 17. We investigated the code of the affected task type and decided on a set of relevant hardware performance counters to be analyzed. The work function associated to long running tasks consists of a loop nest calculating the distance of a block's points to the cluster centers. As this involves access to a large amount of data and frequent conditional updates, we focus on hardware counters for memory and cache accesses as well as branch predictions. Very low cache miss rates quickly rule out hypotheses involving memory accesses, but a significant amount of branch mispredictions occur during task execution.

Figure 18 shows the discrete derivative (difference quotient) of the branch misprediction count rendered on top of a small subset of the heatmap of Figure 17. As the hardware counters for each core used to record the misprediction count have been sampled immediately before and immediately after task execution, the graph interpolates with a constant value corresponding to the average misprediction rate for each task. The interval represented by the vertical axis has automatically been adjusted to the minimum and the maximum number of branch mispredictions per cycle and corresponds to the interval $[0; 0.009215]$. The combination of the graph and the task duration heatmap instantly reveals a correlation: long running tasks with a darker shade of red have a higher branch misprediction rate than short running tasks with a lighter shade.

While visualization helps identifying such correlations on a subset of the trace, manual verification of the correlation on large parts of the trace is impractical. Aftermath allows to automate this process, exporting performance data to a file which can then be processed by an external application. In particular, Aftermath is able to determine the increase of a monotonically increasing counter for each task and to write this information to a file. For the analysis of branch mispredictions this means that Aftermath can determine the number of mispredictions *per task* along with the task duration. Fine-grained control over the contents of the file is given by the filter mechanisms, which also apply to the exported data. This functionality is essential to filter out outliers and to limit the subsequent analysis to certain types of tasks. The actual test of the correlation between the duration and the performance counter can be carried out using a statistics package, such as the SciPy [2] package for Python, used in the analysis below.

A metric for the correlation of two indicators is the coefficient of determination of a linear regression. Figure 19 shows the duration of the main computation tasks in the benchmark as a function of the rate of branch mispredictions. Outliers with an execution time below $1\,\mathrm{Mcycles}$ have been filtered out before exporting the data. The shape of the point cloud in the center visually suggests a linear relationship between the branch misprediction rate and the duration. The dashed straight regression line has been determined using the least squares method to minimize residuals. The coefficient of determination

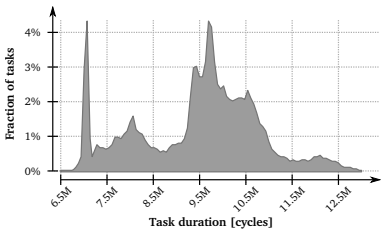


Fig. 16: Distribution of the main computation tasks duration in *k-means*

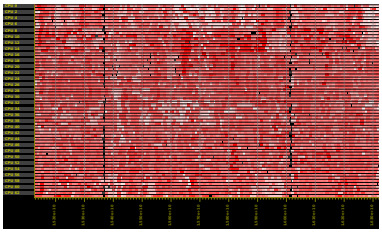


Fig. 17: The timeline in heatmap mode covering several iterations

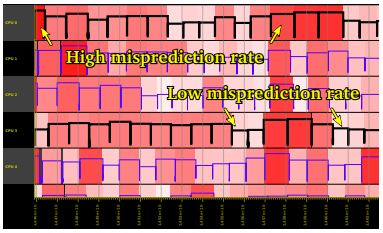


Fig. 18: Zoom with branch misprediction rate

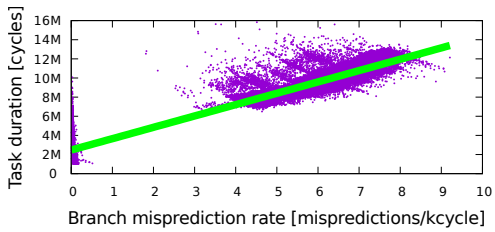


Fig. 19: Task duration as a function of the number of branch mispredictions per thousand cycles in *k-means*

of 0.83 provides statistical evidence for a correlation.

This result implies that the frequent conditional updates in the loop nest have a large impact on performance. It is possible to transform the condition, making the cluster update unconditional, and hoisting the check outside of the time-critical loop. This reduces the mean task duration of the main computational tasks without outliers from 9.76 Mcycles to 7.73 Mcycles and the standard deviation from 1.18 Mcycles to 335 kcycles .

VI. Implementation and performance issues

Beyond the initial focus on functionality, tool design and implementation quickly hit performance issues. The first challenge is to minimize the interference with the application at trace collection, a well-known problem in observing parallel execution behavior. Depending on the programming model, and therefore on the information natively available in the run-time system, generating traces can introduce variable amounts of overhead. In the information-rich environment of the OpenStream run-time, the overall impact on execution time of tracing is generally below the noise threshold, with minor variations within similar confidence intervals, and therefore does not alter application behavior to a significant degree. The second is to optimize the efficiency of the analysis of the trace data to provide a fluid and responsive interface for human interaction and ultimately to scale up to processing very large traces.(10 GiB and above).

A. Trace format

Aftermath traces are organized as streams of data structures. They contain events (i.e., state changes, hardware counters, communication events or discrete events), topological information about the machine, descriptions of hardware performance counters or information about the location of memory regions with respect to NUMA. Structures can appear in any order: as long as event timestamps remain ordered for each worker, events from different cores can be freely interleaved. This avoids adding the overhead for sorting events during trace generation. A total order per core is sufficient to limit overhead when a trace file is loaded.

Aftermath uses a native trace format, currently specialized for the OpenStream run-time. It may analyze trace files that omit certain events, e.g., all OpenStream-specific events. If a trace file only contains beginning and end markers for task execution, but does not include information about memory accesses, Aftermath cannot provide data locality information but may still be used for analyses based on task duration; it may also visualize data from hardware performance counters. The purpose of this incremental approach is twofold: (1) it does not limit Aftermath to any specific run-time system, but preserves its capabilities for targeted analyses when the information is available; and (2) trace collection overhead and trace size can be reduced by omitting data not necessary to the analyses requested by the user.

Finally, the format was also designed to minimize redundancies. Information not explicitly available in the trace file, but needed for rendering or generating statistics, is added to the internal representation when the trace is loaded or on-demand during rendering. For example, the NUMA placement of a given memory region is stored only once, regardless of the number of accesses; when localizing memory accesses, the memory addresses are used to look up the corresponding memory region and find its location. Trace data is stored in a binary format, to reduce its size and to avoid long parsing delays when a trace is opened, and compressed with standard GNU/Linux tools, such as GZIP, BZIP2 or XZ. Aftermath can directly open compressed traces, calling a decompression tool and reading uncompressed data from an unnamed pipe.

B. Optimizations for rendering

Aftermath provides a responsive interface (based on GTK+ [5] and the Cairo Graphics Library [3]), avoiding delays that might interrupt the user's work-flow. It supports arbitrary zooming and scrolling along the timeline through an intuitive interface. Filters directly affect the information displayed, as well as the statistical views for the selected portion of the trace, providing immediate visual feedback. Rendering has been carefully optimized as discussed below. Note that the following principles apply to all timeline modes, but we illustrate them on the state mode and performance counters for simplicity.

a) Every pixel of an overlay (timeline, performance counter, discrete events) is drawn only once: Each horizontal pixel of the timeline represents an interval of the trace. The duration of the interval and therefore the amount of information represented by

the pixel depend on the zoom level. Figure 20 shows two zoom levels A and B. In Zoom A, every pixel represents an interval short enough to display each state. However, when the interval covers multiple state changes, as in Zoom B, a naive approach would render sequentially each state, which is neither efficient nor accurate. Instead, Aftermath determines the predominant state covered by a pixel interval and only renders the associated color once.

As performance counter samples have two dimensions, rendering optimizations affect both the horizontal and vertical direction. Figure 21a shows an example of performance counter samples with linear interpolation. Instead of drawing a line for each pair of adjacent samples, Aftermath determines the minimum and maximum values $v_{\min}$ and $v_{\max}$ for each pixel on the horizontal axis, determines the associated pixels $p_{\min}$ and $p_{\max}$ on the vertical axis, and draws a line between them as shown in Figure 21b. Depending on the zoom level, this leads to a significantly lower number of drawing operations as shown in Figures 21(b) to (d).

b) Aggregation of rendering operations: If the color for adjacent pixels in the timeline is identical, such as in zoom B in Figure 20, Aftermath reduces the number of calls to rendering functions by aggregating these pixels and by issuing a single call drawing a covering rectangle.

c) The use of indexes: Aftermath uses simple, yet efficient data structures for its in-memory representation of traces. Each core uses one array per type of event (state changes, discrete events, performance counter data, etc.) sorted by the timestamp. This allows to determine the array slice containing the relevant events for any interval through a fast binary search.

For each performance counter and each core, Aftermath also builds an n-ary search tree that allows to quickly determine the minimum and maximum value of the counter for any interval on any core. This accelerates the rendering of performance counters described above as it avoids scanning all performance counter values within the interval covered at the resolution of a single pixel. To reduce the memory footprint of these additional search trees, Aftermath uses a default arity of 100 for all search trees, resulting in fewer nodes. This effectively limits the overhead to 5% of the actual performance counter data.

C. Symbol tables and annotations

To enhance user experience when optimizing parallel applications, Aftermath relates the visual elements of the interface to the source code of the application. This information is extracted from the application's binary using the NM command-line tool. When the user selects a task in the timeline, Aftermath retrieves the address of the associated work-function, looks up the corresponding entry in the debug symbols and displays the name of the function in the detailed text view. Clicking on this name starts an editor that opens the corresponding source file and jumps to the function.

To further support the user in the development cycle, in particular in collaborative environments, Aftermath can be used to record user-defined annotations. Since trace analysis can be time-consuming and can involve more than one person, annotations can be saved independently from the trace file and loaded for further analysis at a later time.

VII. Related work

Visualization and analysis of trace files are common techniques, critical for performance analysis and debugging in high performance computing, for which many tools have been developed. We concentrate our survey on the performance debugging capabilities of 8 representative tools.

PARAVER [15] provides powerful interactive filtering mechanisms for multiple graph types and independent views on trace data. Earlier versions of OpenStream included support for trace files in PARAVER's native format. However, PARAVER focuses on computation resources rather than memory and task communication patterns, which are essential to the characterization of performance anomalies on many-core NUMA architectures.

PARAPROF [6], a profile visualization tool of TAU [19], is a retargetable framework for writing trace analysis applications rather than a single tool for a specific type of trace files or performance analysis. It provides a set of extensible components for data sources, data management, analysis and visualization that can be used as a basis for new tools, but does not provide ready-to-use solutions for task-based performance analysis.

PERFEXPLORER [10] is an interactive data mining application for performance analysis, based on TAU and PARAPROF. It offers statistical tools to study correlations based on linear regression, similar to the analysis in Section V. Clustering can be used to group threads with similar characteristics and to relate performance indicators to the topology. However, since PARAPROF is essentially a generic framework for performance data mining; the existing components and those of PERFEXPLORER have little overlap with the specialized ones required for task-parallel applications. As a result, building Aftermath within these frameworks would have been close to the cost of developing Aftermath from scratch.

VAMPIR [13] is a well-known commercial tool that has been used in high performance computing for almost two decades. It provides a rich user interface for interactive exploration and analysis of huge traces and has a highly elaborated filter interface. Multiple connected views with different granularity from cluster level to function calls are supported. But unlike Aftermath, the tool is optimized for the analysis of massively parallel applications based on message passing. Neither NUMA resources nor tasks are modeled.

VITE [1] analyzes parallel programs traces while focusing on fast rendering. However, the tool lacks support for NUMA topologies and customizable analysis filters.

DAGVIZ [12] is a tool that is able to visualize the computation DAGs and parallelism profiles of task-parallel applications. Applications to be analyzed with DAGVIZ must be rewritten using a generic model for task parallelism proposed by the authors. Furthermore, DAGVIZ neither models memory accesses nor data dependences.

JEDULE [11] analyzes task schedules in parallel applications. It offers zoomable Gantt charts, capable of displaying infor-

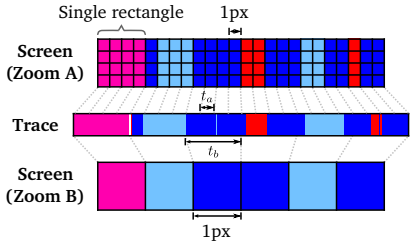
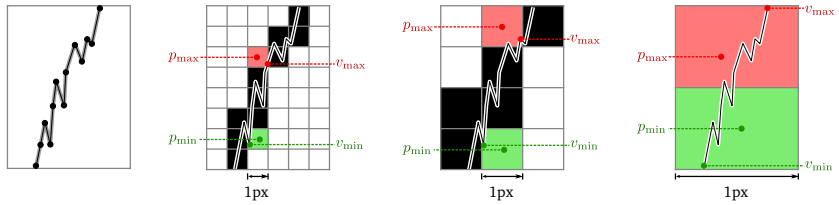


Fig. 20: Timeline at different zoom levels



(a) Actual samples (b) Zoom A (c) Zoom B (d) Zoom C

Fig. 21: Rendering of a performance counter at different zoom levels

mation similar to Aftermath's timeline in state or type mode. However, the scheduling-centric approach only covers the order and execution location of tasks, but does not allow for NUMA-specific analyses.

Weyers et al. [20] proposed a trace-based tool for NUMA analysis that relates interconnect bandwidth usage for an interactively selected interval by the user to the machine topology. The visualization consists in a set of plots showing the evolution of the bandwidth usage of the interconnect between each pair of nodes, organized in the grid of a communication coincidence matrix. The background color of each plot indicates the average bandwidth for the pair of nodes for the selected interval. While these visualizations are well-suited to investigate NUMA-related performance anomalies in general, bandwidth usage cannot be broken down to tasks as the tool does not focus on task parallelism.

VIII. Conclusion

We presented Aftermath, the first performance engineering tool enabling the fine-grained analysis of memory transfers between dynamically created dependent tasks. The new capabilities of Aftermath are particularly well suited to the optimization of locality and concurrency on NUMA systems, involving advanced visualization and analyses. In particular, Aftermath allows the user to immediately identify correlations between program execution characteristics (e.g., slower execution phases, computational load imbalance, insufficient parallelism due to dependences or task creation overhead) and any type of run-time or hardware event (e.g., NUMA locality, hardware performance counters, synchronization, load balancing events, communication).

Aftermath has been used extensively within the OpenStream project and helped resolving a multitude of performance anomalies. We presented a subset of these cases, selected to demonstrate Aftermath's strengths and capabilities.

Aftermath is currently being ported to other dependent tasking models, starting with OpenMP 4.0. We also work on the out-of-core processing of large traces, and proposing semi-automatic statistical methods to quickly focus the search for interesting anomalies.

Acknowledgments

Our work was supported by the grants EU FET-HPC ExaNoDe H2020-671578, UK EPSRC EP/M004880/1, French Nano2017 DEMA. A. Pop is funded by a Royal Academy of Engineering Research Fellowship.

References

[1] http://vite.gforge.inria.fr/. Acc. 09/2015.
[2] http://www.scipy.org/. Acc. 10/2015.
[3] Cairo graphics. http://www.cairographics.org/. Acc. 09/2015.
[4] The DOT language. http://graphviz.org/doc/info/lang.html. Acc. 09/2015.
[5] The GTK+ project. http://www.gtk.org/. Acc. 09/2015.
[6] Robert Bell, Allen D Malony, and Sameer Shende. Paraprof: A portable, extensible, and scalable tool for parallel performance profile analysis. In *Euro-Par 2003 Par. Processing*, pages 17–26. Springer, 2003.
[7] George Bosilca, Aurelien Bouteiller, Anthony Danalis, Thomas Hérault, Pierre Lemarinier, and Jack Dongarra. DAGuE: A generic distributed DAG engine for high performance computing. *Parallel Computing*, 38(1-2):37–51, 2012.
[8] Emden R. Gansner and Stephen C. North. An open graph visualization system and its applications to software engineering. *SOFTWARE - PRACTICE AND EXPERIENCE*, 30(11):1203–1233, 2000.
[9] Thierry Gautier, Xavier Besseron, and Laurent Pigeon. KAAPI: A thread scheduling runtime system for data flow computations on cluster of multiprocessors. In *Proceedings of the 2007 international workshop on Parallel symbolic computation*, pages 15–23. ACM, 2007.
[10] Kevin A. Huck and Allen D. Malony. PerfExplorer: A performance data mining framework for large-scale parallel computing. In *Proceedings of the 2005 ACM/IEEE Conference on Supercomputing*, SC '05, pages 41–, Washington, DC, USA, 2005. IEEE Computer Society.
[11] Sascha Hunold, Ralf Hoffmann, and Frederic Suter. Jedule: A tool for visualizing schedules of parallel applications. In *Proc. of the 2010 39th International Conference on Parallel Processing Workshops*, ICPPW '10, pages 169–178, Washington, DC, USA, 2010. IEEE Computer Society.
[12] An Huynh, Douglas Thain, Miquel Pericas, and Kenjiro Taura. DAGViz: A DAG visualization tool for analzying task parallel program traces. In *Workshop on Visual Perf. Analysis at ACM Supercomputing (SC)*, 2015.
[13] Matthias S. Müller, Andreas Knüpfer, Matthias Jurenz, Matthias Lieber, Holger Brunst, Hartmut Mix, and Wolfgang E. Nagel. Developing scalable applications with Vampir, VampirServer and VampirTrace. In *Proc. of ParCo '07*, volume 15 of *Advances in Parallel Computing*, pages 637–644. IOS Press, 2008.
[14] OpenMP Architecture Review Board. *OpenMP Application Program Interface version 4.0*, July 2013.
[15] Vincent Pillet, Vincent Pillet, Jesús Labarta, Toni Cortes, and Sergi Girona. PARAVER: A tool to visualize and analyze parallel code. Technical report, In WoTUG-18, 1995.
[16] Judit Planas, Rosa M. Badia, Eduard Ayguadé, and Jesús Labarta. Hierarchical task-based programming with StarSs. *Intl. J. on High Performance Computing Architecture*, 23(3):284–299, 2009.
[17] Antoniu Pop and Albert Cohen. OpenStream: Expressiveness and data-flow compilation of OpenMP streaming programs. *ACM Trans. on Architecture and Code Optimization*, 9(4):53:1–53:25, January 2013.
[18] Polyvios Pratikakis, Hans Vandierendonck, Spyros Lyberis, and Dimitrios S. Nikolopoulos. A programming model for deterministic task parallelism. In *Proc. of ACM SIGPLAN Workshop on Memory Syst. Perf. and Correctness*, MSPC '11, pages 7–12, New York, NY, USA, 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] Benjamin Weyers, Christian Terboven, Dirk Schmidl, Joachim Herber, Torsten W. Kuhlen, Matthias S. Müller, and Bernd Hentschel. Visualization of memory access behavior on hierarchical NUMA architectures. In *Proceedings of the First Workshop on Visual Performance Analysis*, VPA '14, pages 42–49, Piscataway, NJ, USA, 2014. IEEE Press.