Language-Centric Performance Analysis of OpenMP Programs with Aftermath

Andi Drebes, Jean-Baptiste Bréjon, Antoniu Pop, Karine Heydemann, Albert Cohen

Language-Centric Performance Analysis of OpenMP Programs with Aftermath

Andi Drebes[1], Jean-Baptiste Bréjon[3], Antoniu Pop[1], Karine Heydemann[2], and Albert Cohen[3,4]

[1] The University of Manchester, School of Computer Science, Manchester, UK
[2] Sorbonne Universités, UPMC Paris 06, CNRS, UMR 7606, LIP6, Paris, France
[3] INRIA, Paris, France
[4] École Normale Supérieure, Paris, France

Abstract. We present a new set of tools for the language-centric performance analysis and debugging of OpenMP programs that allows programmers to relate dynamic information from parallel execution to OpenMP constructs. Users can visualize execution traces, examine aggregate metrics on parallel loops and tasks, such as load imbalance or synchronization overhead, and obtain detailed information on specific events, such as the partitioning of a loop's iteration space, its distribution to workers according to the scheduling policy and fine-grain synchronization. Our work is based on the Aftermath performance analysis tool and a ready-to-use, instrumented version of the LLVM/CLANG OpenMP run-time with negligible overhead for tracing. By analyzing the performance of the *MG* application of the NPB suite, we show that language-centric performance analysis in general and our tools in particular can help improve the performance of large-scale OpenMP applications significantly.

Keywords: OpenMP, Performance Analysis, Tracing

1 Introduction

Optimizing OpenMP programs to exploit modern hardware efficiently is a challenging task. Performance depends on many aspects, such as the amount parallelism exposed by a program, the interaction of the run-time system and the operating system, the locality of memory accesses, and optimizations for sequential performance by the compiler. To identify and eliminate performance bottlenecks, it is crucial for the programmer to understand each of the components involved in the execution as well as their interactions. Such interactions between the parallel program, the hardware and system software is generally out of reach of static analysis. Hence, it is customary to capture *dynamic events* into a trace file at execution time and to perform post-mortem analyses of the trace with appropriate tools [16, 15, 1]. Many existing tools enable the analysis of performance metrics, such as function call profiles, hardware performance counter data and memory allocation. However, only few tools are able to relate performance data to the OpenMP programming and execution model, e.g., to quantify the time spent in barriers, to attribute idle time to parallel regions, or to analyze the load imbalance in parallel loops—information that is essential for choosing the appropriate partitioning schemes for work and data, OpenMP constructs, synchronization and architecture-specific optimizations. Due to the diversity and complexity of the interplay between hardware, run-time programming constructs, and application behavior, it is crucial to rely on both quantitative metrics and detailed information on specific events related to the programming and execution model.

We present a method to analyze the performance of OpenMP applications that suits the needs of synergistic language and hardware analyses, as stated above. Our solution consists of a graphical user interface for performance analysis and visualization, an instrumented OpenMP run-time for trace generation, and a portable library for the generation of trace files. We designed our solution as an extension to Aftermath [8], a tool for trace-based performance analysis, supporting interactive exploration, visualization, filtering and quantitative analysis of programs with dependent tasks. In addition to our extension of the graphical user interface with OpenMP specific tools, we provide Aftermath-OpenMP, an instrumented OpenMP run-time for trace generation and libaftermath-trace, a library that allows programmers to generate trace files that can be analyzed using Aftermath. In contrast to existing tools, we do not only provide a visual representation, aggregate metrics and statistics, but also detailed information for parallel loops, including information on the partitioning of the iteration space among workers. All components are available under free software licenses.

We illustrate the capabilities of our solution by analyzing the performance of *MG* from the NAS Parallel Benchmarks (NPB), executing on a platform with 192 cores and 24 NUMA nodes. We show how this information can be used to determine the cause of bottlenecks and to improve the code, resulting in a speedup of more than 35×. Finally, we demonstrate that the tracing overhead is negligible and thus allows for the collection of precise data for performance analysis.

The paper is organized as follows. Section 2 introduces the components of our solution. In Section 3, we characterize and analyze NPB's *MG* application, then locate and fix two major performance bottlenecks. The overhead induced by trace generation is analyzed in Section 4. Section 5 presents related work on performance analysis, in particular for OpenMP, and we conclude in Section 6.

2 Aftermath: Trace Generation and Analysis

The trace-based analysis of parallel programs requires a component that records dynamic events at execution time into a trace file and a tool for post-mortem analysis of generated traces. As a result, our performance analysis approach needs two separate components: a trace generator called Aftermath-OpenMP—an instrumented OpenMP run-time, and a graphical user interface for trace analysis. We present both components in this section.

2.1 Trace Generation

The Aftermath-OpenMP run-time is based upon LLVM's run-time [3], itself derived from the Intel OpenMP run-time [2]. Our contribution to the run-time consists in adding a lightweight profiling layer to gather information about dynamic events (e.g., execution of parallel loops, barrier synchronization, task creation and execution) through instrumentation of run-time functions. This layer also interfaces the libaftermath-trace library, providing functionality to write all recorded events to a trace file. Libaftermath-trace is a library that provides a common infrastructure for generating trace files compatible with Aftermath and is independent of run-time systems and applications.

To precisely define the tracing capabilities of Aftermath-OpenMP, we introduce the following terms in addition to the terminology of the OpenMP specification [7]: *iteration set*, *iteration period*, and *task period*. We illustrate these terms on the following example with a parallel region composed of two parallel loops and a task.

```
for(int i = 0; i < 2; i++) {
  #pragma omp parallel
  {
    #pragma omp for schedule(static, 10)
    for(int j = 0; j < 100; j++)
      foo();

    #pragma omp for schedule(dynamic, 10)
    for(int k = 0; k < 100; k++)
      bar();

    #pragma omp task
      baz();
  }
}
```

As the parallel region is located in the body of a loop, each parallel loop will be executed twice. Each execution of a parallel loop is referred to as a *loop* and each execution of a task as a *task*. These terms are usually associated with *constructs* in the source code, yet overloading these terms allows for a simple terminology for dynamic analysis and remains unambiguous within the respective context. We use the terms *loop construct* and *task construct* to distinguish constructs and instances when necessary.

Assume that five workers are involved in the execution of the parallel region in the example. The schedule and the chunk size specified for the first loop imply that the first worker executes loop iterations 0 to 9 and 50 to 59, the second worker executes iterations 10 to 19 and 60 to 69, and so on. For a loop with a static schedule, the generated code usually contains only one call to the run-time at the beginning and at the end of the execution of the loop. This means that an instrumented run-time usually cannot distinguish the execution intervals for a worker's chunks. For example, it cannot determine when the first worker has finished executing the chunk $[0; 9]$ and starts executing the chunk $[50; 59]$. We refer to the entire, not necessarily contiguous portion of the iteration space of a loop instance associated to a worker as an *iteration set*. Depending on the schedule, the increment and the chunk size, a worker can have multiple iteration

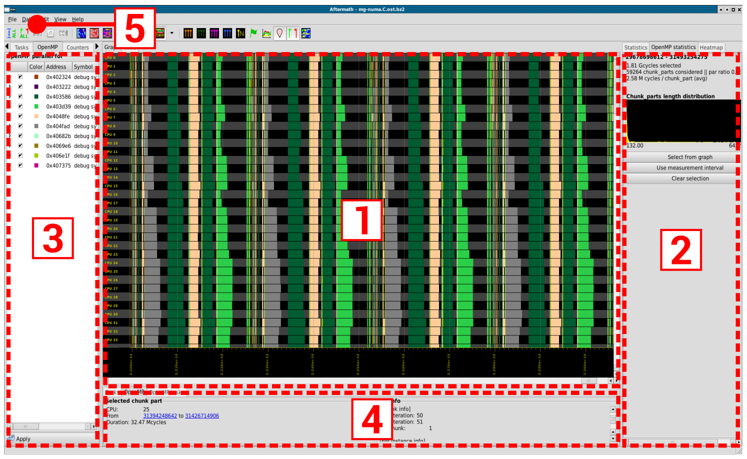


Fig. 1: Main window: timeline (1), filters (2), statistics (3), selected task/event information (4), derived metrics menu (5).

sets, and its iteration sets might comprise one or multiple chunks, each of which might be composed of multiple iterations. Nested parallel regions and barriers can lead to preempting and resuming an iteration set. We refer to a contiguous interval of execution of an iteration set as an *iteration period*. Similar to the loop terminology, we define a *task* as the execution of a task and a *task period* as a contiguous interval of the execution of a task instance.

The Aftermath-OpenMP run-time is capable of tracing each loop and each task executed at least once. For each invocation of a parallel loop or task, the run-time traces loops and tasks, respectively. For loops with a static schedule, for each worker involved in its execution the run-time traces a single iteration set with all information about the set of chunks associated to the worker. For loops with a dynamic or guided schedule, an iteration set is traced each time a worker requests an additional portion of the iteration space. Iteration periods and task periods are recorded according to preemption through events related to barrier synchronization and recursive invocations of parallel loops.

2.2 A Graphical Interface for Trace Analysis

We call Aftermath a *language-centric* visualization and analysis tool, as its graphical user interface exposes hardware and runtime library events at the level of the programming model. The main window is composed of five parts, shown in Figure 1: the *timeline* (1), a panel for *statistics* (2), an interface to configure *filters* (3), a *detailed text view* (4) and a *menu bar* (5) providing access to dialogs.

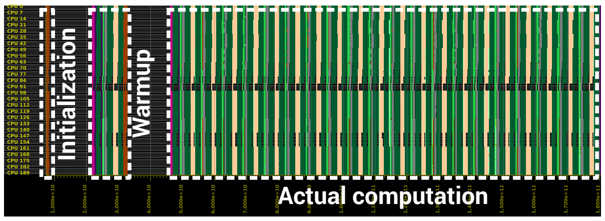
The information displayed on the timeline depends on the activated timeline mode. In the default *state mode*, the timeline shows which run-time states each core has traversed over time (e.g., execution of a single/master construct, waiting on a barrier, execution of a critical region). In *loop construct mode*, Aftermath assigns a different color to each parallel loop construct in the application's source code. This allows the user to obtain a rapid visual overview of which loops have been executed by which processors, when these were executed and how much time their execution has taken. Similarly, in *loop mode*, *iteration set mode*, *iteration period mode*, *task construct mode*, and *task mode*, Aftermath assigns a different color to each loop instance, iteration set, iteration period, task construct and task instance, respectively. The timeline can be overlaid with additional information, e.g., graphs showing the evolution of performance counters recorded for the different cores. In order to explore performance data interactively, the timeline can be zoomed and shifted arbitrarily without noticeable delays, even for large trace files. Intervals without activity (i.e., for which no run-time state was recorded or during which no parallel loop has been executed) are transparent, such that the background of the timeline with an alternating pattern of gray (even cores) and black (odd cores) becomes visible.

The filter interface allows the user to limit the information on the timeline and the statistics panel to specific loop constructs, loop instances, iteration sets, iteration periods, task constructs, task instances, task periods and performance counters. The panel also provides an interface that allows the user to modify the default assignment of colors. All updates are immediately taken into account by the timeline and the statistics panel. The statistics shown in the statistics panel are based on the interval on the timeline selected by the user. For example, the panel shows the duration of the selected interval, how much time has been spent in the different run-time states and a histogram for the distribution of the duration of iteration periods. The text view displays detailed information about a specific item selected from the time line. For a state, it shows its type and duration and for an iteration period, it displays its associated iteration set, loop instance and loop construct, including the number of iterations. The menu bar at the top allows the user to select dialogs to create advanced metrics (e.g., combine two performance counters) or to export data to files (e.g., the contents of the timeline).

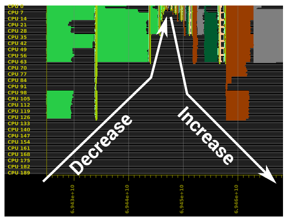
On the implementation side, Aftermath is based on libaftermath-core, the interface to load and analyze trace files in Aftermath's native trace format, GTK [20] for standard graphical user interface components, and the Cairo graphics library [19] for rendering. More information about the algorithms involved for scalable rendering and language-centric visualization can be found in [8].

3 Use case: Optimization of MG

In this section, we show how the user interface presented in the previous section can be used to identify and locate performance bottlenecks in the *MG* application from the NAS Parallel Benchmarks [5], computing the solution of the three-dimensional scalar Poisson equation using a V-cycle multigrid method. In the experiments, we have used the implementation in the C programming language of



(a) Global view with the benchmark's main phases



(b) Between iterations

Fig. 2: Execution phases of *MG*

NPB-2.3 by the Omni Compiler Project [4]. The test system is an SGI UV2000, composed of 24 Intel Xeon E5-4640 CPUs running at 2.4 GHz, with a total of 192 cores and 756 GiB RAM (Hyperthreading disabled). The system runs SUSE Linux Enterprise Server 11 SP3 with kernel 3.0.101-0.46-default. Unless mentioned otherwise, we have set the number of OpenMP workers to 192 with a fixed, round-robin assignment of workers to cores. For a reasonable execution time of several seconds, we have chosen the C input class with $512 \times 512 \times 512$ elements. The benchmark was compiled with LLVM/CLANG, version 3.8.0, and NPB's default compiler flags for optimization.

We first characterize the benchmark's execution phases, then we analyze the execution of parallel loops using Aftermath's timeline, detailed text view and statistics panel in order to locate bottlenecks for performance.

3.1 Identifying Execution Phases

Figure 2a shows Aftermath's timeline in OpenMP loop mode, as presented in Section 2.2. This visual representation indicates three different phases of activity during which the workers execute parallel loops, indicated by the colored sections surrounded by dotted rectangles in the figure. These phases are separated by intervals without loop execution, during which the timeline's background is visible. In the first phase, the benchmark allocates a set of global, multi-dimensional matrices and initializes them in parallel. In the subsequent warm-up phase, a single iteration of the algorithm is performed, resulting in the execution of multiple parallel for loops. The third phase consists of the actual computations of the benchmark. This main phase can be identified on the timeline by the long, repetitive pattern, spanning approximately two thirds of the execution time.

Each iteration of the algorithm in the main phase is characterized by a series of parallel for loops with a decreasing number of iterations, followed by a series of parallel for loops with an increasing number of iterations. This pattern can be spotted by zooming on the timeline between two iterations, as shown in Figure 2b. In the first half of the figure, the height of the colored section decreases, indicating that fewer cores can take part in the execution of loops. In the second

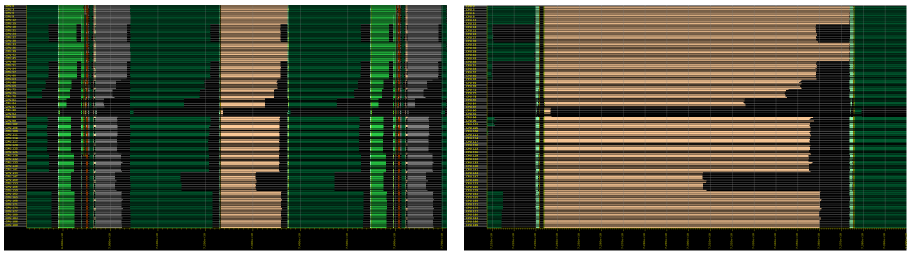

(a) Interval covering two iterations (b) Imbalance for a single loop

Fig. 3: Load imbalance between workers in the main computation phase

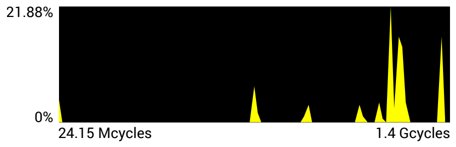


Fig. 4: Histogram of the duration of iteration sets of the selected loop

half, the colored section becomes taller again, which indicates that an increasing number of cores can be used.

3.2 Identifying Load Imbalance Resulting From NUMA

As the application's source code neither specifies the schedule of parallel loops, nor the size of chunks, the default static schedule and chunk size are applied. That is, the iteration space is divided into approximately equal-sized chunks, which are assigned in a round-robin fashion to the workers. The trip count of 512 iterations of most of the loop instances greatly exceeds the number of cores of the test system, such that most of the time all cores can contribute to the computation. The intervals with low parallelism, shown in Figure 2b above, only make up a fraction of the execution time. However, a zoom into the main phase on the timeline also reveals a distinct pattern of imbalance on all of the loops, shown in Figure 3a. To investigate the cause of this imbalance, we arbitrarily pick one of the loops for a detailed analysis.

Figure 3b shows a zoom on one of the loops. The workers with the shortest execution time for this instance are located on cores 88 to 95. As each NUMA node of the machine is composed of eight cores with consecutive core numbers, cores 88 to 95 belong to NUMA node 11. After a click onto one of the associated intervals on the time line, Aftermath provides detailed information about the portion of the iteration space that was executed during this interval. Each of the workers on node 11 executes two iterations of the loop, which takes between 24.15 Mcycles and 29.53 Mcycles. The workers with the longest execution time are located on cores 0 to 15 and 32 to 47 (NUMA nodes 0, 1, 4, and 5). Although these workers only execute one additional iteration in comparison to the worker on cores 88 to 95, their execution time of about 1.4 Gcycles is more than 40 times higher. For other workers, the execution time is between these two extremes. Figure 4 shows the histogram for the duration of iteration sets from the statistics panel for the interval from the beginning of the loop to its end. The distant peaks in this diagram confirm the strong imbalance.

Imbalance is either related to the benchmark itself (e.g., if the amount of work per iteration of a loop varies) or related to unequal access to shared resources due to the topology of the machine (e.g., memory accesses). The parallel loop corresponds to the outermost loop in function *psinv*, with a constant amount of work across iterations, as shown in the listing below.

```c
static void psinv(double ***r, double ***u, int n1, int n2, int n3,
                  double c[4], int k)
{
  int i3, i2, i1;
  double r1[M], r2[M];

  #pragma omp for
  for (i3 = 1; i3 < n3-1; i3++) {
    for (i2 = 1; i2 < n2-1; i2++) {
      for (i1 = 0; i1 < n1; i1++) {
        r1[i1] = r[i3][i2-1][i1] + r[i3][i2+1][i1] +
                 r[i3-1][i2][i1] + r[i3+1][i2][i1];
        r2[i1] = r[i3-1][i2-1][i1] + r[i3-1][i2+1][i1] +
                 r[i3+1][i2-1][i1] + r[i3+1][i2+1][i1];
      }

      for (i1 = 1; i1 < n1-1; i1++) {
        u[i3][i2][i1] = u[i3][i2][i1] +
               c[0] * r[i3][i2][i1] +
               c[1] * (r[i3][i2][i1-1] + r[i3][i2][i1+1] + r1[i1]) +
               c[2] * (r2[i1] + r1[i1-1] + r1[i1+1]);
      }
    }
  }
}
```

This supports the hypothesis that the imbalance is related to the topology of the machine. In fact, Figure 3b also shows that the execution time for workers on the same node is approximately the same. A comparison of the average execution time and the distance[5] of each node to node 11 reveals a correlation between these two metrics. We thus assume that memory accesses are causing the imbalance, with a high fraction of the data placed on the node with the fastest workers, node 11. This suggests a detailed analysis of the memory allocations and initialization of the benchmark.

The initialization routine of *MG* allocates arrays to double precision floating point elements using three and four levels of indirection. The allocation of these hierarchical structures is done stepwise by calling *malloc* from within loopnests:

```c
u = (double ****)malloc((lt+1)*sizeof(double ***));

for (l = lt; l >=1; l--) {
  u[l] = (double ***)malloc(m3[l]*sizeof(double **));

```

[5] As reported by the **numactl** command line tool of LIBNUMA, invoked with the **--hardware** option.

```
6     for (k = 0; k < m3[l]; k++) {
7       u[l][k] = (double **)malloc(m2[l]*sizeof(double *));
8
9       for (j = 0; j < m2[l]; j++)
10        u[l][k][j] = (double *)malloc(m1[l]*sizeof(double));
11    }
12  }
```

The innermost call of *malloc* (Line 10) allocates the space for the actual data. For the input class C, used in the experiments, these allocations have a size that varies between 32 bytes and 4112 bytes. Hence, all of the allocated data regions are smaller than the smallest page size of 4 KiB of the test system, except the largest allocations, which exceed this size by only 16 bytes. Meta data written by the *malloc* function in front of each allocated memory region is likely to be located in the first page of the allocated region. This causes the default first-touch page placement mechanism of the Linux kernel to place the first page of an allocation on the NUMA node associated to the allocating core. Hence, as a side effect of the small allocations of the benchmark, all data pages are placed on a single node. Most of the memory accesses of the main computation phase thus target a single NUMA node, resulting in high memory access latencies due to memory controller contention and remote accesses stressing the machine's interconnect.

To mitigate the early page placement, we have added simple, yet effective custom allocator to the benchmark's source code and replaced the calls to *malloc* in with calls to the new allocator. The revised allocation strategy consists in the allocation of a large, contiguous region of memory on startup and in returning a portion of this region on each invocation of its allocation function. The absence of meta data write accesses delays page placement until the initialization phase of the benchmark. As this initialization is performed in parallel, the first-touch page placement leads to an even distribution across all nodes of the machine, resulting in less contention and thus better performance. With this modification, the execution time is reduced from 48 s to 2.23 s on average for 10 runs.

3.3 Identifying Parallelism Degree Limitations and Imbalance

Figure 5 shows a trace of *MG* after modification of the memory allocation. Although the execution time could be reduced significantly, imbalance between workers is still present. Though, the imbalance pattern has changed, as highlighted by Figure 5b that provides a zoom into one iteration of the algorithm.

The left side of the figure corresponds to the part of the computation where the number of iterations of the parallel loops is below the number of cores. As the execution time of the main loops has decreased, these phases now represent a larger part of the execution. The loops on the right side of the figure have a high iteration count, exceeding the number of workers. In all of the loops, the first workers at the top of the figure spend significantly more time in the loop than the last workers at the bottom. This is due to a mismatch between the trip counts and the number of cores, making it impossible to partition the iteration space evenly across workers. For the dark green and the beige loop

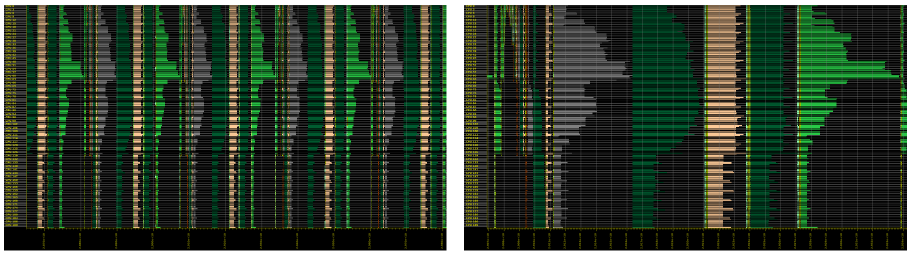

(a) Multiple iterations (b) Different types of imbalance

Fig. 5: Load imbalance with optimized memory allocation

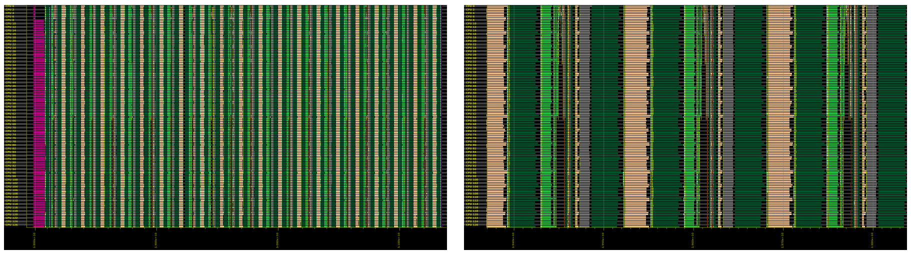

(a) Main computation phase (b) Zoom on two iterations

Fig. 6: Execution using 128 cores

with 512 iterations, the first 128 workers perform three iterations, while the last 64 workers perform only two iterations. Similarly, the first 64 workers executing the gray and light green loops with 256 iterations perform two iterations, while the last 128 workers perform only one iteration. However, for the loops with 256 iterations, the pattern of imbalance is less sharp than for the loops with 512 iterations. This is due to the fact that the memory regions accessed in the additional iterations are placed remotely. Thus, the imbalance pattern resulting from the uneven partitioning is overlaid with an imbalance pattern resulting from the machine topology.

The analysis above implies that an execution with only 128 cores leaves the critical path unchanged with respect to the partitioning of the iteration spaces. Figure 6a shows the main computation phase with this reduced number of cores. The zoom in Figure 6b shows that both the imbalance from the previously unevenly distributed iteration space and the remote memory accesses have disappeared. The latter effect helped reduce the execution time to 1.35 s, which represents a speedup of more than 35× over the initial execution.

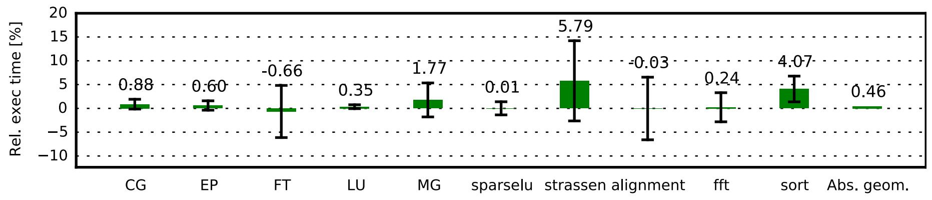


Fig. 7: Relative change of the execution time of tracing wrt. tracing disabled

4 Overhead of Tracing

In this section, we study the overhead induced by the instrumentation of the runtime. In our experiments, we have compared the execution using the unmodified LLVM/CLANG run-time, version 3.8.0 with the execution using our instrumented run-time for a total of 11 applications on the 192-core test system. As the base metric we have used the execution time reported by the benchmarks, i.e., the wall clock execution time, excluding initialization and termination and thus excluding the time needed for disk I/O to write the trace file.

To stress a significant part of the instrumentation code, we have chosen benchmarks from different benchmark suites, covering both benchmarks based on parallel loops and tasks. Loop-based benchmarks are *CG*, *EP*, *FT*, *LU* and *MG* from the C implementation of NPB-2.3 with the C input class, already used in the previous section. For task-based benchmarks, we have chosen *alignment*, *fft*, *floorplan*, *sort*, *sparselu* and *strassen* from the Barcelona OpenMP Task Suite (BOTS, [9]), version 1.1.2. We have used the *omp-tasks* version of the benchmarks where available, otherwise the *for-omp-tasks* version and the largest available inputs.

Some of the benchmarks had to be excluded from the evaluation due to segmentation faults (*BT* of NPB, *nqueens*, *uts* of BOTS), excessive execution time (*SP* of NPB), missing information on the execution time (*IS* of NPB) and unsuccessful verification of the results (*health* of BOTS).

Figure 7 shows the relative change of the execution time in percent when tracing is enabled for a total of 50 executions of each benchmark. These values have been obtained by dividing the execution time of each run with tracing by the average execution time for 50 runs without tracing. Error bars indicate the standard deviation and the values printed above error bars indicate the average change of execution time. The absolute value of the difference is below 6% for all and below 1% for the majority of the benchmarks shown in the graph. We have excluded *floorplan* from the plot, as its tracing overhead of 380% is two orders of magnitude higher than for the other benchmarks. A quick analysis of this benchmark with Aftermath revealed that the huge increase of the execution time is the result of the creation of a large amount of tasks with a very short duration of only a few thousand cycles. However, the geometric mean of the absolute values of the mean differences excluding *floorplan* is below 0.4%, such that the overhead can be considered negligible.

5 Related Work

We compare our approach with the main tools and methodologies for the performance debugging of OpenMP programs. Regarding trace generation, a collaborative API called OMPT is gathering momentum [10]. It is based on the experience of POMP [12] and the Sun/Oracle collector API [13] and is primarily designed for first-party performance tools (i.e., running in address space of the application). OMPT provides limited support for sampling-based performance measurement as well as *blame shifting*, shifting the attribution of costs from symptoms to causes. Depending on the OpenMP implementation, it may be implemented entirely by the compiler, the run-time or both. In its current state, it provides callbacks for thread, parallel region and task begin/end and tracks the state of mutexes, but it does not track loop scheduling, iteration set or iteration period information. This limitation is shared with other tools such as OPARI2 [6], or Extrae [7], the library to generate Paraver trace files. Although the latter has wrappers for major OpenMP libraries the semantical information about OpenMP loops is not available. All the later also lack precise information about OpenMP tasks, such as the spawning tree, array sections and dependences.

On the visualization side, Vampir [15] is a well-known commercial tool 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 an 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. OpenMP is supported at the granularity of parallel regions only. Paraver [16] provides powerful independent views on trace data. However, Paraver focuses on interactive filtering mechanisms for multiple graph types and independent views on trace data. The tool focuses on computation resources rather than loop chunks, task memory access and communication patterns, which are essential to the characterization of performance anomalies of OpenMP programs. On the other hand, Intel's VTune [8] provides finer grained, per-parallel-region analysis of load imbalance, potential performance gain, and enables correlations with hardware counters, but it neither models detailed chunk information, nor nested parallelism.

Unlike the former tools, ParaProf [6] is a retargetable analysis and visualization toolkit, part of TAU [18]. It does not provide ready-to-use solutions for task-based performance analysis. One such solution based on ParaProf is PerfExplorer [11], an interactive data mining application for performance

[6] http://www.vi-hps.org/tools/opari2.html

[7] http://www.bsc.es/computer-sciences/extrae

[8] https://software.intel.com/en-us/articles/profiling-openmp-applications-with-intel-vtune-amplifier-xe

analysis. However, ParaProf's existing components and those of PerfExplorer have little overlap with the specialized ones required for OpenMP loops and tasks applications. As a result, building Aftermath within these frameworks would have been close to the cost of development from scratch.

Finally, Grain Graphs [14] differentiate for the former, chronogram-based frameworks. Its hierarchical graph representation is aimed at OpenMP programmers with little performance debugging experience, and at improving productivity of performance tuning. It facilitates the localization of performance anomalies on the source code, including load imbalance, limited parallelism degree, and synchronization granularity issues. But it suffers from scalability issues on larger traces and—by design—it does not provide a consistent timeline to correlate specific loop chunk or task events. Complementarities between grain graphs and chronogram-based visualization deserve to be investigated in the future.

6 Conclusion and Future Work

We presented a synergistic language and hardware approach to the performance analysis of OpenMP programs. It is designed and implemented as an extension to the Aftermath trace analyzer and the LLVM/clang OpenMP run-time. We contributed an efficient, low-overhead instrumentation of a state-of-the-art OpenMP run-time and a graphical user interface that provides a visual representation of OpenMP constructs, aggregate metrics for statistics, and methods for the detailed inspection of dynamic loop instances, iteration sets, iteration periods, tasks constructs and task instances. We demonstrated that performance analysis at the level of the parallel programming model is essential to characterize and to correct performance bottlenecks on large-scale parallel machines. Application to the *MG* benchmark led to 35× improvement over the baseline OpenMP implementation on a 192-core non-uniform memory access system. We also showed that our solution for trace file generation only causes negligible overhead on the execution time.

We plan to extend the components presented in this paper and the coverage of the programming model. In particular, we will provide support for the tracing of dependent tasks introduced with OpenMP 4, and add tools for the analysis of the task graph, similar to the analyses Aftermath provides for dependent tasks in OpenStream [17, 8].

References

1. http://vite.gforge.inria.fr. Accessed 05/2016.
2. Intel openmp runtime library. https://www.openmprtl.org. Accessed 05/2016.
3. LLVM OpenMP support. http://openmp.llvm.org. Accessed 05/2016.
4. Omni compiler project. http://www.hpcs.cs.tsukuba.ac.jp/omni-compiler/download/download-benchmarks.html. Accessed 05/2016.
5. D. Bailey, E. Barszcz, J. Barton, D. Browning, R. Carter, L. Dagum, R. Fatoohi, S. Fineberg, P. Frederickson, T. Lasinski, R. Schreiber, H. Simon, and V. Venkatakrishnan. The NAS Parallel Benchmarks. Technical report, 1994.

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. OpenMP Architecture Review Board. *OpenMP Application Program Interface Version 4.5*, November 2015.
8. Andi Drebes, Antoniu Pop, Karine Heydemann, and Albert Cohen. Interactive visualization of cross-layer performance anomalies in dynamic task-parallel applications and systems. In *2016 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, April 2016.
9. Alejandro Duran, Xavier Teruel, Roger Ferrer, Xavier Martorell, and Eduard Ayguade. Barcelona openmp tasks suite: A set of benchmarks targeting the exploitation of task parallelism in openmp. In *Proceedings of the 2009 International Conference on Parallel Processing*, ICPP '09, pages 124–131, Washington, DC, USA, 2009. IEEE Computer Society.
10. Alexandre Eichenberger, John Mellor-Crummey, Martin Schulz, Nawal Copty, Jim Cownie, Robert Dietrich, Xu Liu, Eugene Loh, and Daniel Lorenz. OpenMP Technical Report 2 on the OMPT Interface. Technical report, 2014.
11. 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.
12. Marty Itzkowitz, Oleg Mazurov, Nawal Copty, and Yuan Lin. An OpenMP Runtime API for Profiling. http://www.compunity.org/futures/omp-api.html. Accessed 05/2016.
13. G. Jost, O. Mazurov, and D. An Mey. Adding new dimensions to performance analysis through user-defined objects. In *Intl. Conf. on OpenMP shared memory parallel programming, IWOMP'05/IWOMP'06*, pages 255–0266. Springer-Verlag, 2008.
14. Ananya Muddukrishna, Peter A. Jonsson, Artur Podobas, and Mats Brorsson. Grain graphs: Openmp performance analysis made easy. In *Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '16, pages 28:1–28:13, New York, NY, USA, 2016. ACM.
15. 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.
16. 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.
17. Antoniu Pop and Albert Cohen. OpenStream: Expressiveness and Data-flow Compilation of OpenMP Streaming Programs. *ACM Trans. Archit. Code Optim.*, 9(4):53:1–53:25, January 2013.
18. Sameer S. Shende and Allen D. Malony. The tau parallel performance system. *Int. J. High Perform. Comput. Appl.*, 20(2):287–311, May 2006.
19. The Cairo Graphics Team. Cairo graphics. http://www.cairographics.org. Accessed 05/2016.
20. The GTK+ Team. The GTK+ project. http://www.gtk.org. Accessed 05/2016.