Visual Comparison of Trace Files in Vampir

Matthias Weber[1], Ronny Brendel[2], Michael Wagner[1,3], Robert Dietrich[1], Ronny Tschüter[1], and Holger Brunst[1]

[1] Technische Universität Dresden, Germany
{matthias.weber, robert.dietrich, ronny.tschueter, holger.brunst}@tu-dresden.de
[2] Oak Ridge National Laboratory, USA
brendelr@ornl.gov
[3] Barcelona Supercomputing Center, Spain
michael.wagner@bsc.es

Abstract. Comparing data is a key activity of performance analysis. It is required to relate performance results before and after optimizations, while porting to new hardware, and when using new programming models and libraries. While comparing profiles is straightforward, relating detailed trace data remains challenging.
This work introduces the Comparison View. This new view extends the trace visualizer Vampir to enable comparative visual performance analysis. It displays multiple traces in one synchronized view and adds a range of alignment techniques to aid visual inspection. We demonstrate the Comparison View's value in three real-world performance analysis scenarios.

Keywords: alignment, comparison, performance analysis, tracing, visualization

1 Introduction

HPC application developers need to leverage the potential performance of modern HPC computing systems. Increasingly complex hardware configurations as well as software systems make achieving this goal ever more challenging. Performance analysis tools aid developers in obtaining better scalability, tracking down bottlenecks, and in general enable detailed understanding of the performance characteristics of applications.

Performance tools observe metrics about and behavior of a running application and its underlying system. After an execution they present the obtained performance data in text or through visualizations. Investigating application performance typically involves comparing data between multiple application runs, for example with varying number of processing elements, varying inputs, and varying hardware configurations.

Performance data comes in two principal flavors: profiles and traces. Profiles consist of aggregated performance data. For instance a *flat profile* summarizes the number invocations and time spent for each function of an application. Another common form is the *call path profile*. It works similarly, but aggregates information for a function call stack configuration, rather than a function and disregarding its calling context. Comparing two profiles can be achieved by subtracting the values of one profile from the other for each equal function (or call path).

However, the ability of profiles to reveal performance problems is limited. Many performance issues are dynamic in nature, and hard to detect in an aggregated statistic. For example to see if load balance worsens over time, or a performance flaw occurs only occasionally (and is thus averaged out), more detailed performance data is needed.

Traces retain the chronological order of events, where profiles do not. This more fine-grained performance data enables investigating dynamic behavior of applications. However, displaying more data, makes investigating traces more difficult. Comparing two traces is even more challenging, because two runs of the same application can be very different. Possible reasons for this include timing differences, reordered functions, partially removed stacks (due to e.g. function inlining) and changes in dynamic application behavior.

In this work we introduce techniques to improve visual trace comparison. Our contributions are:

- Vampir's [2] *Comparison View*, which displays traces and statistics side by side. It allows manual alignment of multiple traces on one common timescale to simplify visual comparison.
- A heuristic to automatically align traces.
- A case study demonstrating the effectiveness of the *Comparison View* for performance analysis of real-world applications.

All introduced techniques are available starting with the current Vampir and VampirServer releases.

This work is organized as follows: Section 2 enumerates related work. Section 3 presents our techniques and implementation. Section 4 shows how the Comparison View aids performance understanding and improvement for three real-world scientific applications. The last section summarizes our contribution and highlights future development directions.

2 Related Work

Comparing performance data is a central activity in performance analysis. Consequently, a wide range of research has been performed on this topic. Solutions range from profile comparison techniques to advanced trace compression and analysis schemes. The developed techniques can serve different purposes. For example clustering algorithms can be used to compress data, automatically categorize it, or aid visualization.

Schulz and de Supinski's [14] present a tool, based on GNU gprof, for comparing profiles between application runs. Song et al. [15] introduce an algebraic framework for comparing profile-based performance data.

PerfExplorer [7] provides a framework for performance data mining. It allows comparing runtime, relative speedup, and efficiency across different sets of profiles.

Weber et al. developed techniques for structural comparison of performance data. They introduced a hierarchical alignment algorithm [17] that compares the structure of two traces. Building on that, they introduced alignment-based comparison metrics [20] to highlight structural and temporal differences between application runs. In subsequent work they developed a structural clustering algorithm [18] that scalably classifies processing elements into groups of similar behavior.

In the context of the performance analysis tool Paraver [13], Llort et al. [11] use object tracking techniques to automatically detect changing performance characteristics between application runs. Knüpfer et al. [9] propose the *Compressed Complete Call Graph* structure for trace data. The technique identifies similarities inside of and between processes to store trace data more efficiently. This compressed storage is able to speed up various analysis operations, as well as reveal repeating patterns in program execution.

Mohror and Karavanic [12] divide applications into segments. Based on the similarity of these segments they compress the traces while retaining as much crucial performance information as possible. Gamblin et al. [4] use an adapted k-means clustering algorithm to scalably compress trace data.

Trace viewers like Vampir [2] and Intel Trace Analyzer [8] provide visual analysis of trace data. Intel Trace Analyzer offers visual comparison of two traces, but lacks the ability to align them. To compare trace it stretches both into the same time frame. This leads to a rather unnatural representation of the trace, making visual comparison challenging. Vampir supports folding timelines [19] to save space and get an aggregate view of multi-processing element activity (e.g. CUDA streams, or OpenMP threads) where appropriate. Edge bundling [1] is a promising technique for aiding visual analysis and comparison of large-scale communication traces.

3 Methodology

This section introduces a new *Comparison View* and its features for visual trace file comparison. The Comparison View integrates all common performance charts of Vampir and adds additional comparison functionality. To enable effective comparison of multiple trace files we couple and synchronize the zoom of the traces. In Vampir the user can *zoom* into regions of a trace to investigate areas in more detail. To compare areas of interest between traces, displayed trace regions need to be freely shiftable in time. This allows for arbitrary alignments of the trace files, and thus, enables visual comparison of user selected areas side by side.

The number of compared trace files is not limited by Vampir. However, Vampir needs to load the complete data of all trace files into main memory. Thus, the amount of available main memory becomes the limiting factor. In order to compare large trace files exceeding the size of typically available memory on workstations, users can employ VampirServer for the comparison. VampirServer runs in parallel on an HPC machine. This component allows to harness the distributed memory of a cluster for the comparison.

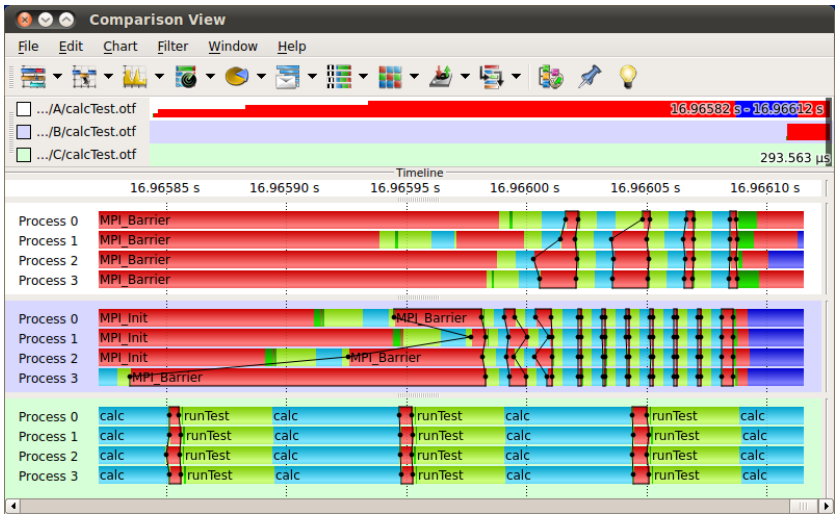

Fig. 1. The Comparison View showing three traces

Figure 1 shows the new *Comparison View*. We use three example trace files to introduce its comparison functionality. The example traces show one test application performing ten iterations of calculations. Each trace, respectively, represents an execution of this application on a different machine.

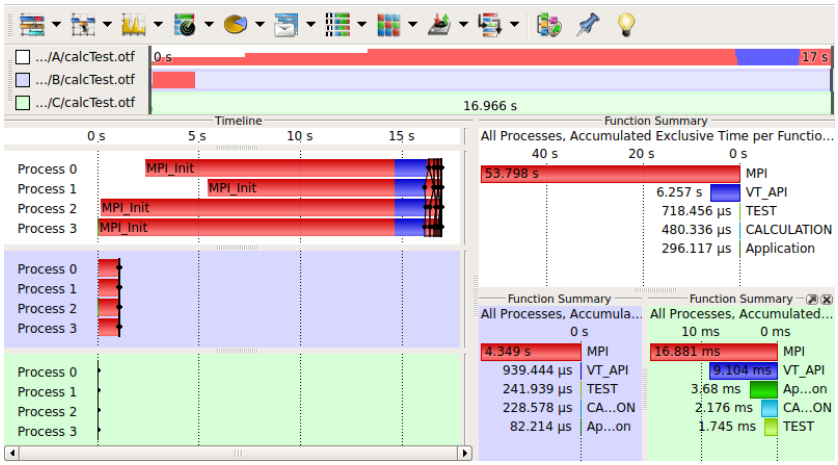

Fig. 2. Open Comparison View

3.1 Comparing Application Characteristics using Charts

As indicated by the *Navigation Toolbar*, at the top in Figure 2, all three trace files are included in the single Comparison View instance. The Navigation Toolbar gives an overview of all open traces and provides an easy access for manipulating the selected zoom area. The Comparison View provides timeline and statistic charts (common charts of Vampir) for the comparison of performance metrics. Colors in the charts represent different function calls, e.g., MPI calls are shown in red, computations are shown in cyan (in this example). The Comparison View opens one chart instance for each loaded trace file, i.e., one click on the Master Timeline icon opens three Master Timeline charts. In order to distinguish charts between traces, we assign a dedicated background color to all charts belonging to one trace.

As shown in Figure 2, trace A exhibits the largest duration time. The duration of trace C is so short that it is barely visible. Zooming into the compute iterations of trace C (left side in Figure 2 at $0s$) would make them visible and allow an detailed inspection. However, due to the coupled zoom, zooming into the area around $0s$ would also zoom into the `MPI_Init` phase in trace A and B. To visually compare the compute iterations between all three traces, they need to be aligned side by side. This necessitates an alignment method for the traces to facilitate a meaningful visual comparison of related areas. We present the available options for trace alignment in the following.

3.2 Aligning Traces Manually

The Comparison View allows to shift individual trace files in time. This enables comparison of areas that did not occur at the same time. In our example the

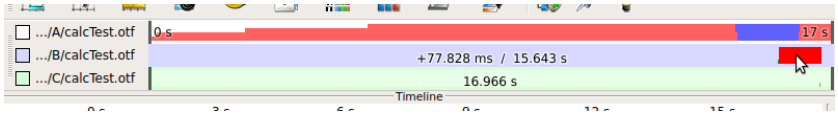

Fig. 3. Coarse alignment of traces using the Navigation Toolbar

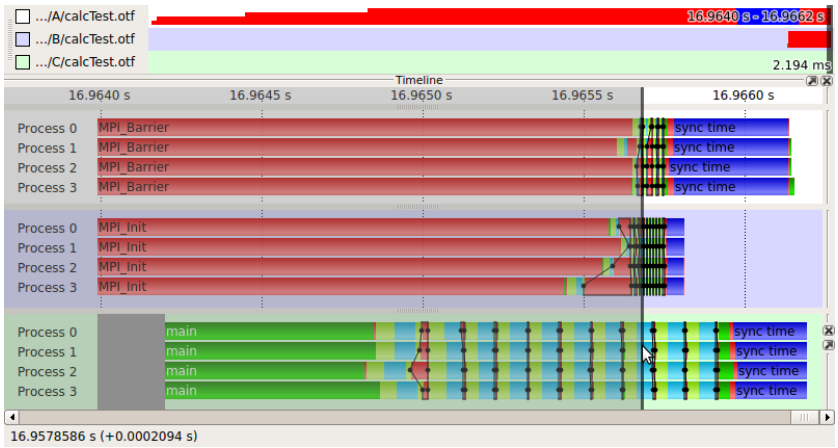

Fig. 4. Fine-grain alignment of traces directly in the Master Timeline

compute iterations need to be aligned prior to visual comparison because the initialization of the application took different times on the three machines.

We provide several ways to shift trace files in time. One option is to directly set the time offset of an individual trace using a context menu. The easiest way to achieve a coarse alignment is to directly drag and drop (using the mouse) the trace in the Navigation Toolbar. Figure 3 shows the compute iterations of all example trace files coarsely aligned.

After the coarse shifting the Master Timeline allows a finer alignment directly inside the chart. Therefore, the user can zoom into an area of interest and directly align the traces by dragging with the mouse. Figure 4 depicts the process of dragging trace C to the compute iterations of trace A and B.

As shown in Figure 4, although the initialization of trace A took the longest, this machine was the fastest in computing the calculations.

3.3 Aligning Traces Automatically Using Predefined Markers

Markers in traces point to particular places of interest in the trace data. These markers are useful for navigation in trace files. For trace file comparison markers are interesting due to their potential to quickly locate places in large trace data sets. They allow to quickly find the same location in multiple trace files.

The *Marker View* in Vampir provides a combined access to all markers contained in the open trace files. Clicking a marker in the Marker View highlights the

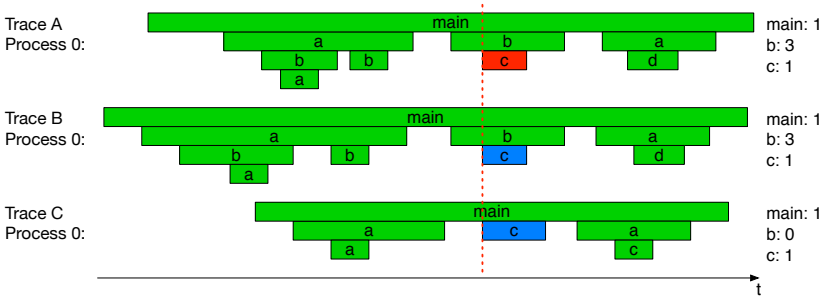

Fig. 5. Example showing the automatic alignment of three processes. Function `c` (shown in red) in trace A is selected. Our heuristic finds the respective function invocations in trace B and C and aligns all traces at that function. The function invocation profiles for the selected functions (red line) are shown at the right side.

respective marker in the Master Timeline. Another way to navigate to a marker in the timeline charts is to use the zooming functionality. Therefore, a user first zooms into the desired zooming level. Clicking a marker in the Marker View will then set the timeline chart to the marker position. Thus, the selected marker appears in the center of the timeline chart. Moreover, the Marker View provides two additional ways of navigating with markers. If two markers of one trace are selected, the Comparison View sets the zoom to the according timestamps of the markers. If two markers of different traces are selected, the Comparison View adjusts the time offset between the respective traces and shows the selected markers next to each other, and consequently, aligns both traces at the respective markers.

3.4 Aligning Traces Automatically Using Call Invocation Profiles

In addition to manual alignment we also provide a heuristic that automatically aligns traces. Using this heuristic users can select an interesting function in one trace and have all other traces aligned automatically to that function. This facilitates direct visual comparison by saving the user from manual aligning.

In this section we describe the alignment heuristic using the example shown in Figure 5. In this example the user selected function `c` of *Process 0* in *Trace A* (marked red in the figure). To align *Trace B* and *Trace C* to that function, we first select the corresponding process in the other traces. Therefore, we search for a process with the name *Process 0* in all open traces[4]. If no exact match is found, we compute the Levenshtein distance [10] between names to find the closest match.

In order to detect the corresponding invocations of function `c` in *Trace B* and *Trace C* we employ an invocation profile based approach. For reference we first

[4] This simple example contains only one process per trace. However, in parallel applications searching for the selected process is necessary.

generate the function invocation profile for *Trace A*. Therefore, we identify all functions contained in the call stack of the selected function (red line in *Trace A* in the figure). In our example these functions are `main`, `b`, and `c`. Then we traverse *Trace A* and count all occurrences of these functions until we reach the selected function. For *Trace A* this results in the following invocation profile (also shown right of *Trace A* in Figure 5): `main`: 1 invocation, `b`: 3 invocations, and `c`: 1 invocation.

For the alignment we then traverse *Process 0* of *Trace B* and *Trace C* and try to match their invocation profiles as good as possible with the profile of *Trace A*. In case of a structurally identical traces (e.g., *Trace A* and *Trace B*), we find the related function with a perfect match between both profiles, i.e., the difference between the invocations of both profiles is zero: $|main_{TraceA} - main_{TraceB}| + |b_{TraceA} - b_{TraceB}| + |c_{TraceA} - c_{TraceB}| = |1-1| + |3-3| + |1-1| = 0$. If both profiles match, we stop searching and align both traces at the related functions. In case of structural differences (e.g., *Trace A* and *Trace C*) we try to find the position with the lowest possible error between both profiles. For instance, when we reach the first function `c` in *Trace C* the corresponding profile is: `main`: 1 invocation, `b`: 0 invocations, and `c`: 1 invocation. This results in a profile difference of 3: $|main_{TraceA} - main_{TraceC}| + |b_{TraceA} - b_{TraceC}| + |c_{TraceA} - c_{TraceC}| = |1-1| + |3-0| + |1-1| = 3$. When reaching the second invocation of function `c` the profile changes to: `main`: 1, `b`: 0, and `c`: 2. This results in a profile difference of 4: $|main_{TraceA} - main_{TraceC}| + |b_{TraceA} - b_{TraceC}| + |c_{TraceA} - c_{TraceC}| = |1-1| + |3-0| + |1-2| = 4$. The second profile exhibits a higher difference to the reference profile than the first profile. Thus, we stop searching and select the first invocation of `c` as related function in *Trace C*.

The comparison of m trace files requires the traversal of m processes. The complexity for the traversal of one process is $\mathcal{O}(n)$ with respect to the number of recorded events (each function invocation consists of one *enter* and one *leave* event) in the related process. Thus, in total the complexity for an alignment of m processes is $\mathcal{O}(m \cdot n)$, assuming maximal n events in each process. In practice, the computation of the heuristic does not introduce any noticeable lag in the visualization of Vampir. Interestingly, the computation of a *Function Summary* (profile view in the figures) in Vampir poses even higher requirements, involving a full traversal of all processes of a trace[5].

This approach improves the usability of the visual comparison. The heuristic exactly aligns structurally equal processes. While not perfect, it is also sufficiently robust to correctly align trace files in many cases of structural differences between processes.

[5] This example is not directly related to the alignment heuristic. It is only mentioned here to contrast the computational requirements of the alignment heuristic with common processing steps in Vampir.

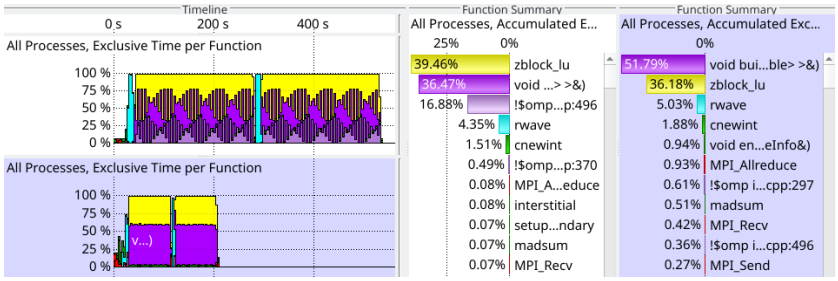

Fig. 6. Overview of a 80-GPU LSMS run on Titan (white background) and Summitdev (blue). The timeline display is on the left. Profiles are on the right.

4 Case Study

This section showcases how the comparison view benefits visual performance analysis. Three real-world optimization scenarios demonstrate its wide applicability. *LSMS* analyzes the performance impacts of different hardware on an application. *CloverLeaf* compares several versions of an application executed using different programming models. *Trinity RNA-Seq Assembler* performs a comparative scalability study of an application and detects scalability bottlenecks.

4.1 LSMS – Comparing Performance between Different Hardware

The Oak Ridge Leadership Computing Facility uses Vampir and its comparison view for visual performance analysis to support porting applications from Titan to Summit. The system employed for early development work is Summitdev.

These new systems bring a number of major changes. Some of them are: Summitdev consists of 20 Power8+ cores and four NVIDIA P100 GPUs per node. One P100 has four times the theoretical DPFLOPS peak performance compared to the Tesla K20X used in Titan. One node has four GPUs instead of one for Titan. The system supports CUDA MPS, which allows sharing GPUs between multiple processes.

To explore how these differences affect the CORAL benchmark code LSMS [3] visual performance analysis is vital. Figure 6 shows an LSMS run on 80 Titan vs 20 Summitdev nodes. The total number of graphics cards for both is 80.

The vastly faster GPUs and the fact that each GPU has at most five CPU threads paired (20 divided by 4), in comparison to 16 on Titan, cause the GPU-accelerated function `zblock_lu` to speed up, while the non-GPU-enabled function `buildKKRMatrix` gains in relative execution time. Thus to further improve LSMS's performance, `buildKKRMatrix` is the new prime function to investigate.

To compare iterations in detail developers use the alignment functionality, shown in Figure 7.

To gauge whether CUDA MPS can speed up LSMS, we run it with varying numbers of threads and processes per node (Figure 8). The first run has four

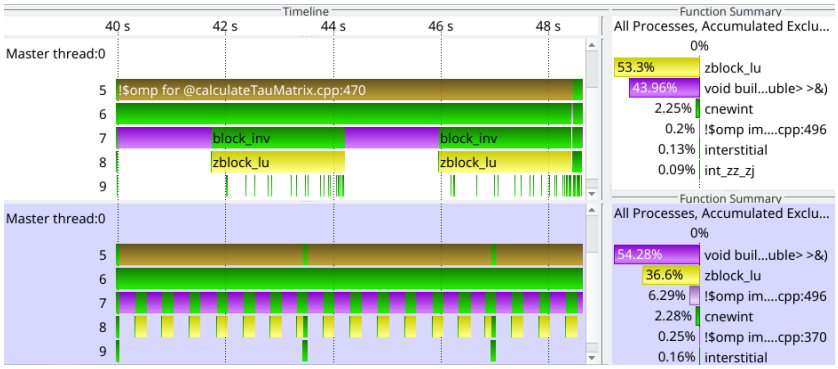

Fig. 7. Detailed comparison of one iteration on Titan vs roughly 2.5 on Summitdev

MPI processes with four threads each. The second one has five threads each. The third and fourth runs are 8×2, and 16×1, i.e. two and four processes share one GPU. Strictly, LSMS is most resource-efficient if the total number of threads and processes divides the number of simulated atoms evenly. But, it turns out using all 20 cores in a four by five setup is faster than the other variants, although it adds occasional waiting time on the "left-over" threads. Note that 8 or 16 processes cannot evenly use 20 cores with the same number of threads per process. Another interesting observation is that the increase in MPI waiting time (more red in the green and cyan timelines) is negated by better GPU utilization.

Summarizing our findings, GPU MPS uses the GPU more efficiently. But not using four cores per node negates this advantage.

The comparison view highly improves visual comparative analysis. With its help, we are able to gain a deeper understanding of LSMS's changing performance characteristics while transitioning to Summit.

4.2 CloverLeaf – Comparing Performance between Programming Models

CloverLeaf is a hydrodynamics mini-app, which solves the compressible Euler equations on a Cartesian grid with an explicit, second-order accurate method [6]. It is composed of small execution kernels, which simplifies the implementation with different programming models. To accelerate the computation, the grid can be split into parts and processed on multiple MPI processes, threads, and target devices, which however requires a halo exchange and thus, data transfers.

This paper compares the CUDA and the OpenACC implementation[6] on an NVIDIA K80 GPU as target device. We ran all experiments with two MPI processes, a fixed grid size of 1920×960 cells, and a fixed number of 87 time steps. The test system was equipped with two Xeon E5-2680v3 CPUs at 2.5GHz

[6] Available at http://uk-mac.github.io/CloverLeaf/, last accessed 26 September 2017

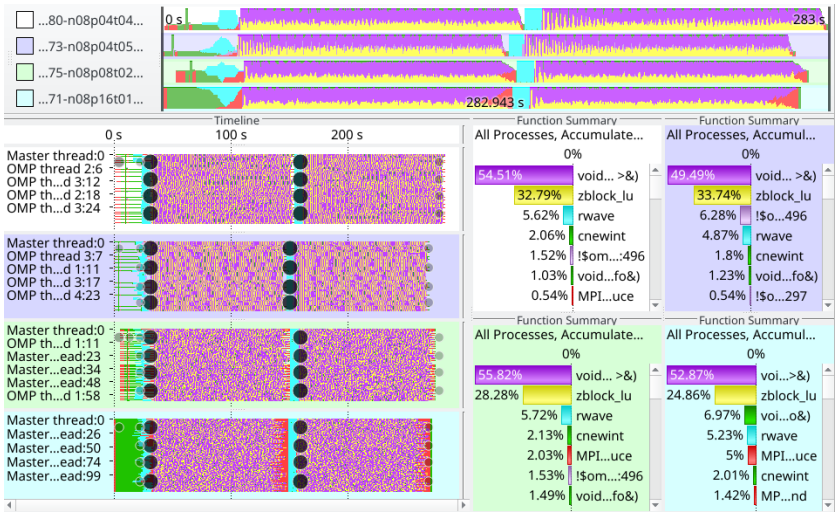

Fig. 8. Exploratory comparison of different process vs thread setups. White: 4 processes times 4 threads per node. Blue: 4 processes times 5 threads. Green: 8 times 2. Cyan: 16 times 1.

and four K80 GPUs at fixed clock rates of 823MHz. We used the PGI 17.7 compilers for the OpenACC implementation and the Intel 16.0.2 compilers for the CUDA implementation. The CUDA toolkit was installed in version 8.0.44.

Figure 9 compares runs of three different versions of CloverLeaf: the initial OpenACC version (white background), the CUDA version (purple background), and an improved OpenACC version with exclusive GPU usage (green background). The *Navigation Toolbar* at the top shows that the initial OpenACC version takes almost twice as much time as the other runs, with regard to the computation phase. A closer look into the execution exposes that it uses the default offloading device on both MPI processes, which results in resource contention with an MPI imbalance as further symptom. CUDA kernels, launched by one MPI process, delay the kernel execution from the other process. Some CUDA kernels, such as `pdv_kernel_80_gpu`, run concurrently on the GPU as they do not fully utilize all compute resources. In the CUDA version and the fixed OpenACC version, both MPI processes use one GPU exclusively, which prevents resource contention and keeps the MPI imbalance negligible. Although the CUDA version is comparatively fast, considering the total runtime, it reveals optimization potential in the selected program phase. Costly `cudaMalloc` and `cudaFree` calls, invoked by thrust library routines, could be avoided, especially as they are nonexistent in the OpenACC implementations.

The automatic alignment of traces facilitates the review of small code changes. Figure 10 shows the effect of an optimization in the halo exchange of the OpenACC version. The traces have been aligned at function `update_halo`. The first optimization avoids two unnecessary host-to-device transfers per `pack kernel`,

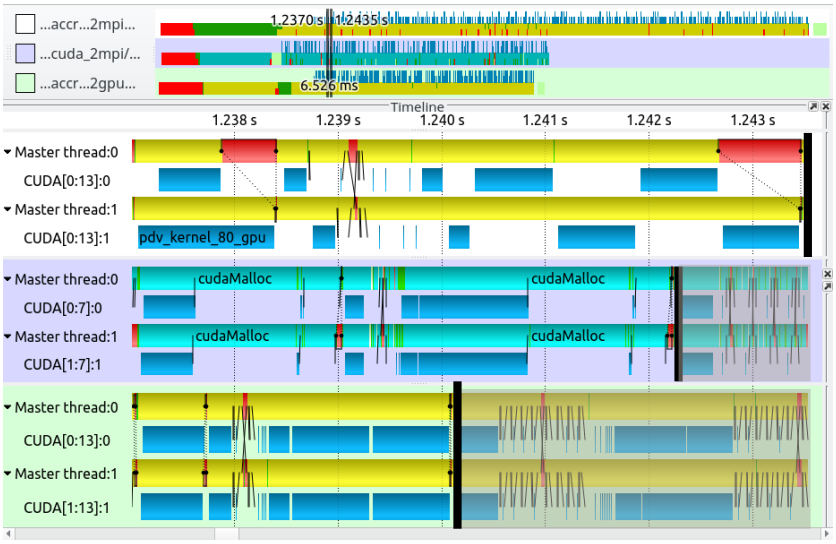

Fig. 9. Comparison of different CloverLeaf implementations: initial OpenACC (top), CUDA (middle), and improved OpenACC (bottom).

indicated by the two missing black lines in the optimized version (purple background). The second optimization replaces synchronous offloading of multiple successive CUDA kernels or data transfers with asynchronous equivalents and a collective synchronization. The effect is obvious for a set of successive kernels, which update the halo on the GPU. They are executed one after another, without the kernel trigger overhead in between. The same optimization has been applied for the `pack kernel` and its following device-to-host transfer.

The comparison view helps porting applications to new programming APIs. It allows users to spot runtime and structural differences, which finally helps in detecting individual weak spots of implementations. Eventually, comparing traces is useful to validate code optimizations.

4.3 Trinity RNA-Seq Assembler – Comparing Performance between Different Process Numbers

In this section we highlight our efforts to analyze and optimize the RNA-Seq assembler Trinity with the help of the Comparison View [16]. Trinity [5] is a software framework for accurate de novo reconstruction of transcriptomes from RNA-Seq data. Trinity is a pipeline of up to 27 individual components in different programming and script languages, including C++, Java, Perl, and system binaries, which are invoked by the main Trinity perl script. The pipeline consists of three stages: first, *Inchworm* assembles RNA-seq data into sequence contigs, second, *Chrysalis* bundles the Inchworm contigs and constructs complete de Bruijn

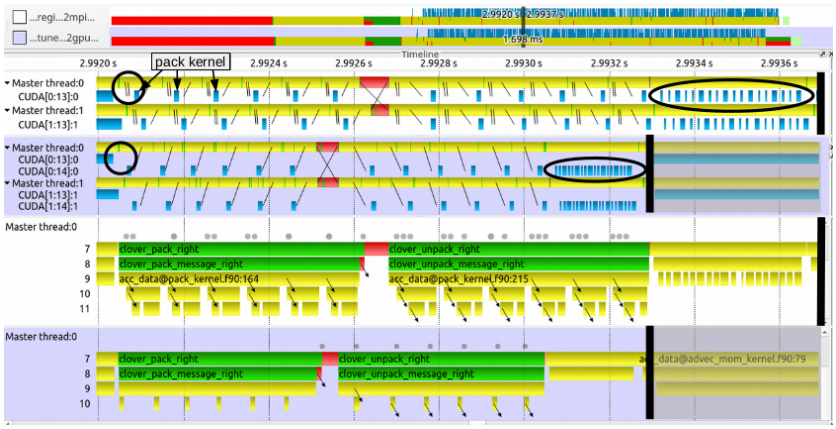

Fig. 10. Validation of code optimizations for the CloverLeaf OpenACC port

graphs for each cluster, and, third, *Butterfly* processes the individual graphs in parallel and computes the final assembly.

Our analysis results refer to the release version 2.0.6, while many of our optimizations were included in the release version 2.1.1. One of the main performance issues that was identified is the poor intra-node scaling of the `GraphFromFasta` module. `GraphFromFasta` is a key part of the *Chrysalis* stage that clusters the Inchworm contigs and constructs complete de Bruijn graphs for each cluster.

The intra-node parallelism using OpenMP showed very poor scalability by achieving a speed up of only 2.27 with a full 16-core node in comparison to the version with only one core [16]. To further investigate this issue we analyzed the parallel intra-node behavior.We recorded application traces with 1, 2, 4, 8, and 16 OpenMP threads using manual instrumentation of code regions in the main loop. Figure 11 shows the recorded behavior in comparison for 1, 2, 4, 8, and 16 threads from top to bottom with white, red, yellow, green, and blue background, respectively. The left side depicts the active code regions over time on the horizontal axis and the executing threads on the vertical axis. The summarized overview on the right side presents the accumulated runtime over all threads for each code region.

The comparison view in Figure 11 reveals that the work load in the first part of `GraphFromFasta` increases nearly linearly with the number of OpenMP threads. Consequently, there is practically no parallel speed up with more than two threads. We identified the root cause for this behavior being the frequent creation and destruction of string stream objects within an inner loop of the massively called function `is_simple`. The string stream creation is internally locked by a mutex, which produces excessive wait time since all threads simultaneously created the string stream objects with a very high frequency. This is visible by the increasing amount of time spent in the code region marked `stringstreams`, from about 25s with one thread to 260s with 16 threads.

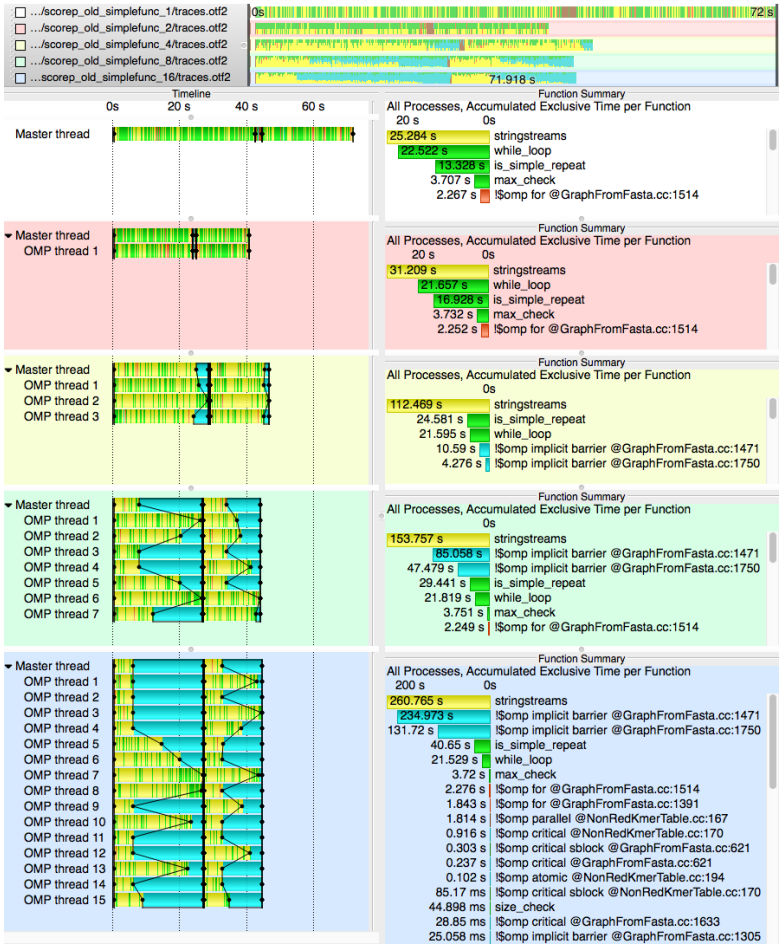

Fig. 11. Resource utilization of original Trinity 2.0.6 version

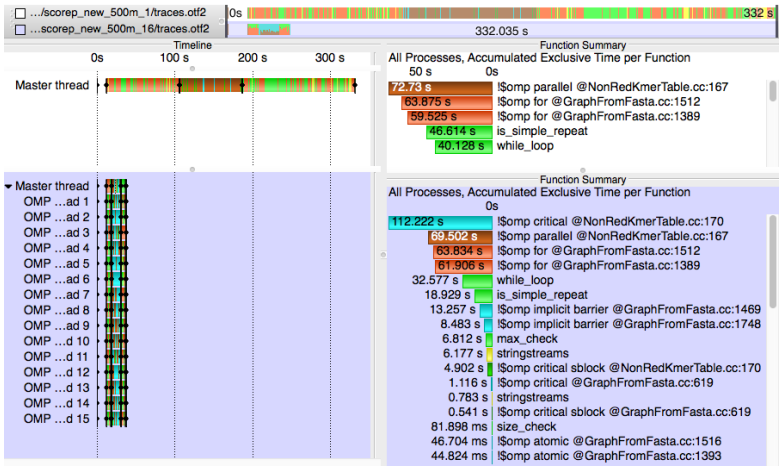

Fig. 12. Resource utilization of optimized Trinity version

Further investigation showed that the string stream creation can be moved out of the inner loop by creating the string stream object before the loop and only clearing the string streams in the inner loop. Consequently, we were able to avoid the serialization in this critical section.

This optimization leads to a substantial increase in parallel performance and, therefore, a remarkable reduced runtime for the first part of `GraphFromFasta`. In addition to the better scaling, the serial runtime is reduced, as well; for the analyzed test data set, the serial runtime decreases from $72s$ to $45s$. Figure 12 shows the improved scaling of the optimized version. The parallel speed up is increased to 8.9 instead of 2.3 with the unoptimized version.

During the analysis of Trinity the comparison functionality was pivotal in understanding the parallel, intra-node behavior of the `GraphFromFasta` module and in identifying and omitting the root causes of poor parallel scalability. Equipped with this knowledge, we were able to introduce modifications resulting in a speedup of 3.9 in the intra-node performance of the `GraphFromFasta` module and in combination with other optimizations a 22 % improvement in overall run time.

5 Conclusions

This work introduces features for visual trace comparison in Vampir. Our contributions enable simultaneous inspection of multiple traces in a synchronized *Comparison View*. This view is already available in current Vampir and VampirServer releases. It greatly simplifies analyzing application performance for, i.a., different input data sets, software versions, processing element setups and hardware architectures.

We present three methods for synchronizing the zoom of multiple traces. Users can align traces manually, automatically using predefined markers, and via a heuristic that aligns according to the call profile. The latter method works well even if the traces have diverging structure.

Three use cases demonstrate the wide applicability of the Comparison View for performance analysis of real-world applications and highlight its benefits for detailed visual comparison of performance data.

In this work we focus on visual comparison and structural alignment of multiple traces. We intend to use this work as a basis for enhanced analysis approaches which automatically analyze structural and temporal differences.

Acknowledgments

This research used resources of the Oak Ridge Leadership Computing Facility at Oak Ridge National Laboratory, which is supported by the Office of Science of the Department of Energy under Contract DE-AC05-00OR22725.

References

1. Brendel, R., Heyde, M., Brunst, H., Hilbrich, T., Weber, M.: Edge bundling for visualizing communication behavior. In: Proceedings of the 3rd International Workshop on Visual Performance Analysis. pp. 1–8. IEEE Press (2016)
2. Brunst, H., Weber, M.: Custom Hot Spot Analysis of HPC Software with the Vampir Performance Tool Suite. In: Proceedings of the 6th International Parallel Tools Workshop, pp. 95–114. Springer (September 2012)
3. Eisenbach, M., Nicholson, D.M., Rusanu, A., Brown, G.: First principles calculation of finite temperature magnetism in fe and fe3c. Journal of Applied Physics 109(7), 07E138 (2011)
4. Gamblin, T., de Supinski, B.R., Schulz, M., Fowler, R., Reed, D.A.: Clustering Performance Data Efficiently at Massive Scales. In: Proceedings of the 24th ACM International Conference on Supercomputing. pp. 243–252. ICS '10, ACM, New York, NY, USA (2010)
5. Grabherr, M.G., Haas, B.J., Yassour, M., Levin, J.Z., Thompson, D.A., Amit, I., Adiconis, X., Fan, L., Raychowdhury, R., Zeng, Q., Chen, Z., Mauceli, E., Hacohen, N., Gnirke, A., Rhind, N., di Palma, F., Birren, B.W., Nusbaum, C., Lindblad-Toh, K., Friedman, N., Regev, A.: Full-length Transcriptome Assembly from RNA-Seq Data Without a Reference Genome. Nature Biotechnology 29(7), 644–652 (2011)
6. Herdman, J. A. et al.: Accelerating Hydrocodes with OpenACC, OpenCL and CUDA. In: SC Companion: High Performance Computing, Networking Storage and Analysis. pp. 465–471 (2012)
7. Huck, K.A., Malony, A.D., Shende, S., Morris, A.: Scalable, Automated Performance Analysis with TAU and PerfExplorer. In: Proceedings of the 14th Conference on Parallel Computing (ParCo 2007). pp. 629–636 (2007)
8. Intel Trace Analyzer and Collector. `http://software.intel.com/en-us/articles/intel-trace-analyzer/` (Nov 2015)
9. Knüpfer, A., Voigt, B., Nagel, W.E., Mix, H.: Visualization of repetitive patterns in event traces. In: Applied Parallel Computing. State of the Art in Scientific Computing, pp. 430–439. Springer (2007)
10. Levenshtein, V.I.: Binary Codes Capable of Correcting Deletions, Insertions, and Reversals. Soviet Physics Doklady 10(8), 707–710 (1966)
11. Llort, G., Servat, H., González, J., Giménez, J., Labarta, J.: On the Usefulness of Object Tracking Techniques in Performance Analysis. In: Proceedings of SC13: International Conference for High Performance Computing, Networking, Storage and Analysis. pp. 29:1–29:11. SC '13, ACM, New York, NY, USA (2013)
12. Mohror, K., Karavanic, K.L.: Evaluating similarity-based trace reduction techniques for scalable performance analysis. In: Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis. p. 55. ACM (2009)
13. Pillet, V., Labarta, J., Cortes, T., Girona, S.: PARAVER: A Tool to Visualize and Analyze Parallel Code. In: Proceedings of WoTUG-18: Transputer and occam Developments. pp. 17–31 (March 1995)
14. Schulz, M., de Supinski, B.R.: Practical Differential Profiling. In: Proceedings of the 13th international Euro-Par conference on Parallel Processing. pp. 97–106. Euro-Par'07, Springer-Verlag, Berlin, Heidelberg (2007)
15. Song, F., Wolf, F., Bhatia, N., Dongarra, J., Moore, S.: An Algebra for Cross-Experiment Performance Analysis. In: Proceedings of the 2004 International Conference on Parallel Processing. pp. 63–72. ICPP '04, IEEE Computer Society, Washington, DC, USA (2004)

16. Wagner, M., Fulton, B., Henschel, R.: Performance Optimization for the Trinity RNA-Seq Assembler, pp. 29–40. Springer (2016)
17. Weber, M., Brendel, R., Brunst, H.: Trace File Comparison with a Hierarchical Sequence Alignment Algorithm. In: Proceedings of the 2012 IEEE 10th International Symposium on Parallel and Distributed Processing with Applications. pp. 247–254. ISPA '12, IEEE Computer Society, Washington, DC, USA (July 2012)
18. Weber, M., Brendel, R., Hilbrich, T., Mohror, K., Schulz, M., Brunst, H.: Structural Clustering: A New Approach to Support Performance Analysis at Scale. In: Proceedings of the 30th IEEE International Parallel and Distributed Processing Symposium (IPDPS). pp. 484–493. IEEE Computer Society (May 2016)
19. Weber, M., Geisler, R., Brunst, H., Nagel, W.E.: Folding Methods for Event Timelines in Performance Analysis. In: Proceedings of the 29th IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW). pp. 205–214. IEEE Computer Society (May 2015)
20. Weber, M., Mohror, K., Schulz, M., de Supinski, B.R., Brunst, H., Nagel, W.E.: Alignment-Based Metrics for Trace Comparison. In: Proceedings of the 19th International Conference on Parallel Processing, pp. 29–40. Euro-Par'13, Springer-Verlag, Berlin, Heidelberg (2013)