2012 10th IEEE International Symposium on Parallel and Distributed Processing with Applications

Trace File Comparison with a Hierarchical Sequence Alignment Algorithm

Matthias Weber, Ronny Brendel, Holger Brunst
Center for Information Services and High Performance Computing
Technische Universität Dresden
Dresden, Germany
Email: {matthias.weber, ronny.brendel, holger.brunst}@tu-dresden.de

***Abstract*—Performance optimization, especially in the field of HPC, is an integral part of today's software development process. One powerful way of optimizing applications is to analyze their event traces. Yet, the comparison of traces of multiple application runs is cumbersome. The impact of optimizations in the source code or the usage of different compiler flags has to be tracked manually. The challenge is to automatically identify exactly those areas that changed in the large amount of trace data.**

We propose a novel solution that combines sequence alignment algorithms with call graph analysis to compare and highlight traces event-wise. Our approach is able to automatically detect differences by aligning event traces. Fine-grained execution time differences can be extracted and displayed in performance charts. The results of our implementation are presented and discussed.

***Keywords*-performance analysis, comparison, sequence alignment, tracing.**

I. Introduction

Today's high-end computers contain hundreds of thousands of cores, pushing the need of application scalability even further. Rising energy costs and almost constant single core performance necessitate efficient application implementations. In order to test and improve implementations, performance analysis and optimization is a necessary part of the development process.

Two key methods for performance optimization are profiling and tracing. Profiling provides summary information of the application run. With help of profiling, functions and code areas that qualify as good candidates for performance optimization can be found. In parallel computing the detection of many performance problems requires more information than can be provided by summaries. Such problems can be correctly identified by an analysis based on event traces. An *event* is the occurrence of an application activity, like enter/leave of a function, MPI messages, or input/output operations to a file. The generation of event traces requires a monitor to be attached to the application. During the application run this monitor writes all activities together with their occurrence time in a buffer or a file. Event traces allow a detailed insight into the temporal order of activities of the application. The analysis methods presented in this paper are based on the tracing approach.

The comparison of trace files provides additional potential for the performance optimization and debugging of applications. This is particularly helpful for the analysis of effects caused by modifications to the application software or changes to the hardware environment. One example, on the hardware side, is that one application is run on different machines and the performance impact on the application needs to be analyzed. Here trace files generated for application runs on each machine could be compared. Another common example, on the software side, is the analysis of the performance of different algorithms in order to find the most efficient solution. By comparing trace files it is possible to analyze every aspect of changed behavior in the application runs. The impact of the changes to, e.g., the computation-, network-, or I/O performance can be analyzed.

The challenge is to track down the performance critical areas in the possibly large amount of trace data. In terms of the comparison of two trace files, this task starts with the identification of areas that are equal or different in both traces or that are missing in one of the files. In order to identify these areas we generate function sequences from the trace data and apply sequence alignment algorithms to them. Sequence alignment algorithms are heavily used in the life sciences to align, for example protein- or nucleotide sequences. During the alignment these algorithms detect equal, different, and gap areas in the sequences. The required time to calculate an alignment between two sequences depends on the sequence lengths. Due to the quadratic computational complexity of dynamic programming algorithms with respect to sequence length the alignment of long function call sequences cannot be computed in reasonable time. For this reason, we combine the sequence alignment algorithm with the call stack of the application. By aligning only sub-function sequences that lie on the same call level, it is possible to cut down the sequence length considerably. This approach produces valid alignment results in a reasonable time. Furthermore, it enables the comparison of trace files generated by real world application runs.

After successful identification of equal areas it is possible to generate an exact execution time difference graph for both traces. This allows to quickly evaluate the impact of changes on the performance of code segments.

II. Related Work

In the area of performance analysis there are various approaches that perform the analysis of trace data. Some

978-0-7695-4701-5/12 $26.00 © 2012 IEEE
DOI 10.1109/ISPA.2012.40

provide plain display functionality for two traces, others use compression techniques to handle and analyze large trace data.

The Intel Trace Analyzer [1] provides functionality to compare two trace files. It displays the two files together in one view and computes differences and speedups on the basis of summary information. This summary information is calculated for user selected ranges in the traces.

Mohror and Karavanic [2] evaluate similarity metrics with the context of trace file reduction. Aim of this work is to use redundancy in application runs in order to shrink the required space for the generated trace files. The evaluated metrics are capable of identifying similar regions in the trace data. The identification process is based on predefined regions.

Casas et al. [3] perform analysis of trace data. Using sampling they construct discrete signals from trace file metrics, e.g., a signal indicating the number of compute processes. These signals are used as input for wavelet and correlation analysis. With help of these methods it is possible to automatically detect application phases and structures and to filter out regions in the application run that are disturbed by trace buffer flushes to disk. The results of this analysis are used to identify iterations in the application run that possibly best represent the application behavior. Afterwards only these iterations need to be saved.

Preissl et al. [4] present an algorithm to detect and to extract communication patterns from MPI event traces. They first use suffix trees to detect local repeating communication behavior. Afterwards global communication patterns of the application are constructed by combining local repeating segments of multiple processes. This approach can help to identify possible performance bottlenecks related to the application's communication behavior.

Knüpfer et al. [5] propose a solution based on call graphs. Their approach exploits redundancy in application runs. Repetitive call sequences need to be stored only once and can be found quickly. This solution is capable of identifying equal patterns in processes.

Ratn et al. [6] developed a scalable tracing method that exploits redundancy and hence is able to detect similarities. Due to the compression of the trace, time accuracy is lost in the trace data.

To our knowledge the solution proposed in this paper is the only one that uses sequence alignment algorithms for trace comparison. It is based on call graph analysis and presents the novelty of using the alignment of two traces to identify equal, difference and gap areas between application runs.

III. Trace Alignment

For the trace file comparison methods described in this paper we assume similarly behaving applications. This is the case if one application is traced multiple times. Applied improvements are unlikely to alter the complete application structure. For the analysis of traces generated from one application it is perfectly reasonable to compare the same processes of each trace, (i.e., process one of trace A with process one of trace B). In order to compare two respective processes we

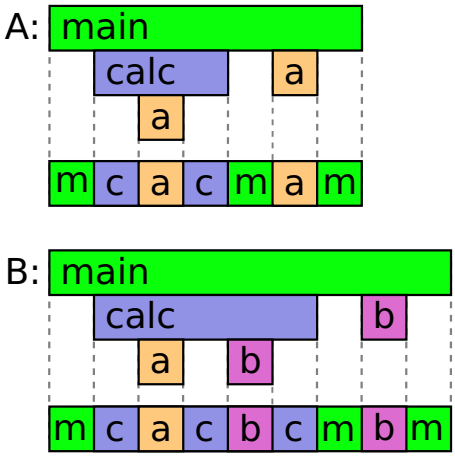


Fig. 1. Construction of function sequences from processes

align them to each other. This allows event-wise matching of the processes.

We consider areas as equal if they have the same sequence of function calls in both traces. This means that we do not take the timing information of the function calls into account. Only their sequence order must be identical. This measure is needed because the timing information in multiple trace files is never exactly the same. Even if the same application is traced two times on the same machine, the respective traces will differ due to effects like OS noise, varying network traffic, or different load on the system. Although their sequence of function calls is rarely guaranteed to be equal, it is likely to be the same for large parts the application. The performance characteristics of those equal areas can be directly compared. Different areas are sections that contain different function calls at the same sequence position. For instance, if a call to function *a* in the first application run is replaced by a call to function *b* in the second application run, these calls would be recognized as different. For performance analysis the identification of these areas is important, as the application is probably executing different code. The third possibility is a missing section, or a gap, in one trace file. This happens for example if a new code section is added to the application. If the first trace file is generated with the original application code and the second is generated with the additional code, the new code segment causes a gap in the first trace file. Gaps enable the analyst to quickly identify code areas that are only present in one of the traces.

The alignment of the processes is calculated with the dynamic programming approach [7] [8] [9]. The dynamic programming algorithm calculates optimal alignments for arbitrary sequences. Hence, sequences need to be constructed out of the trace data.

Figure 1 illustrates how the sequences are constructed from the function call structure. In order to be able to align the sequences, they need to contain unique identifiers for functions. The most reasonable criteria we found is to use

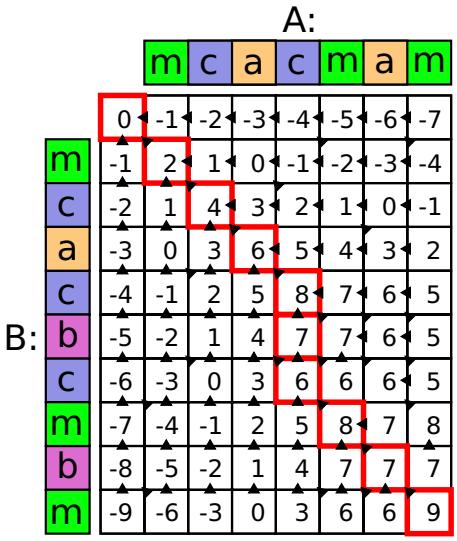


Fig. 2. Dynamic programming matrix

the function name as basis for the identifier. That way equally named functions get the same identifier in both sequences, and therefore make them comparable. The sequences are constructed by building a list of successive function calls. In the example described in this section, the following two sequences are aligned:

Sequence A: m c a c m a m

Sequence B: m c a c b c m b m

The following notation is used: Sequence A is of length M and sequence B of length N. The i^{th} function call in A is A_i and the j^{th} function call in B is B_j.

The dynamic programming algorithm breaks down the full pairwise alignment problem into independently optimizable sub-problems. The algorithm evaluates alignments with scores. A recursive scoring scheme is used to find the optimal alignment of sub-problems. For the comparison of trace files the following scores have been chosen:

$$\begin{aligned} \sigma_{i,j} &= 2 && \text{Match Score} \\ \sigma_{i,j} &= -1 && \text{Mismatch Score} \\ \gamma &= -1 && \text{Gap Score} \end{aligned}$$

$\sigma_{i,j} = 2$ is chosen if the function call A_i is the same as B_j, otherwise, if A_i differs from B_j then $\sigma_{i,j} = -1$ is used. In case that either A_i or B_j are aligned to a gap the gap penalty of $\gamma = -1$ is applied. Based on the scores the following recursive scoring scheme is defined:

$$D_{i,j} = \max \begin{cases} D_{i-1,j-1} + \sigma_{i,j}, & \text{Match/Mismatch} \\ D_{i,j-1} + \gamma, & \text{Gap in Sequence } A \\ D_{i-1,j} + \gamma. & \text{Gap in Sequence } B \end{cases}$$

In order to keep track of already solved sub-problems a dynamic programming matrix D is used, Figure 2. The matrix

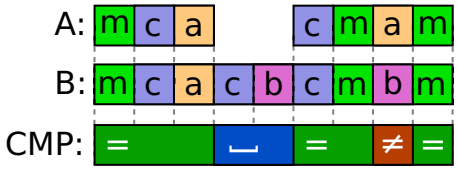


Fig. 3. Constructed alignment of sequence A and B

is initialized at the top left corner with $D_{0,0} = 0$. The entry $D_{0,0}$ gives the start score of the alignment. By applying the recursive scoring scheme the matrix is filled from the top left to the bottom right corner. The bottom right entry $D_{M,N}$ holds the optimal score for the complete alignment of both sequences. In order to find the optimal alignment, each matrix field is filled with the highest scoring option from the scoring scheme. For the construction of the alignment it is useful to remember what choice led to the optimal score for each field. In Figure 2 this is indicated with arrows between the matrix fields.

Backtracking through the matrix allows to find the optimal sequence alignment. Starting point is the bottom right corner, which represents the optimal score. Then a path along the arrows to the top left corner needs to be followed. In case of multiple possible ways from one to the next field, the highest scoring field is chosen. The backtracking step is indicated by the red path in Figure 2. The red path represents the optimal alignment. If the path goes diagonally both respective sequence functions are aligned to each other (Match/Mismatch). If the path goes vertically, the respective function of sequence B is aligned to a gap area, if it goes horizontally, the respective function in sequence A is aligned to a gap area. Figure 3 shows the constructed optimal alignment of sequence A and B. The "CMP" bar in the figure marks equal-, different-, or gap areas.

The classical dynamic programming algorithm has quadratic complexity with respect to the sequence lengths, $O(MN)$, in time and memory. This renders the alignment of large sequences impossible due to the limited size of available main memory. The algorithm proposed by Hirschberg [10] exploits the fact, that for the computation of each cell $D_{i,j}$ in the matrix, only cells $D_{i-1,j}$, $D_{i,j-1}$, and $D_{i-1,j-1}$ are needed. Therefore the algorithm does not have to hold the complete matrix in memory. The trade off is that some results need to be computed multiple times. However, this algorithm is able to calculate the optimal alignment with quadratic time complexity, $O(MN)$, but with only linear memory complexity with respect to the longest sequence, $O(N)$. Therefore, the Hirschberg algorithm has been selected to calculate the optimal alignment.

A. Flat Sequence Alignment

The first approach in order to compare traces is to use the Hirschberg algorithm to align the complete function call sequences of two processes. Figure 5 shows an example comparison.

The "CMP" bar illustrates the alignment. It marks areas

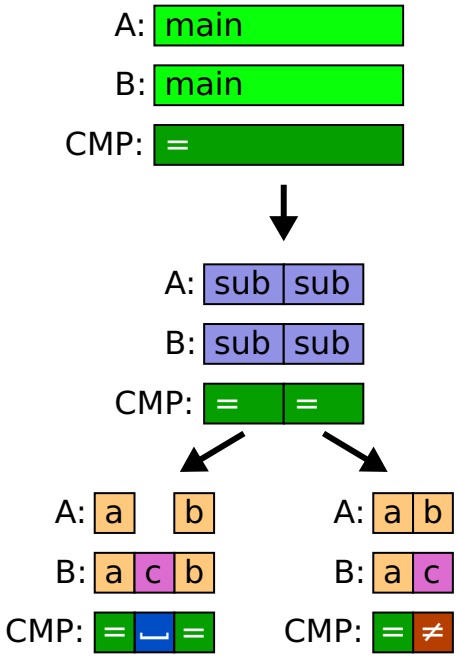


Fig. 4. Hierarchical alignment algorithm scheme

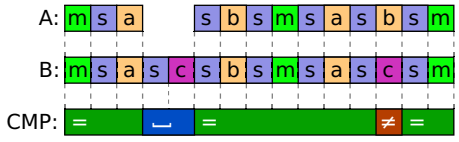


Fig. 5. Comparison of two processes using flat alignment

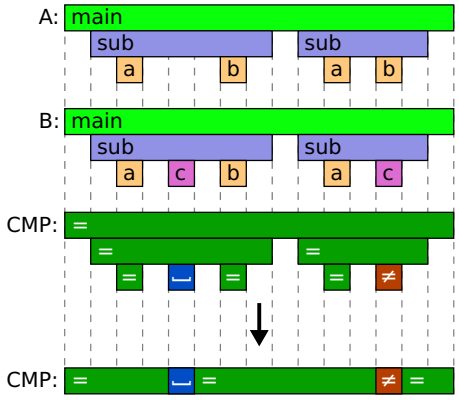


Fig. 6. Comparison of two processes using hierarchical alignment

where both processes execute the same or different functions. Areas where one process is executing functions that are missing on the other are marked as gaps.

This approach satisfies the required functionality completely. Yet, it has a critical downside. Due to the quadratic time complexity of the algorithm, the alignment of medium to long function sequences is not computable in reasonable time. To overcome this drawback a new hierarchical comparison approach has been developed. This solution is described in the next section.

B. Hierarchical Alignment

To be able to align, and subsequently compare long application runs, the function sequence length has to be reduced. The truncation of the sequence is not an option, if the complete application run is to be considered. It is however possible to split up the sequence into several smaller parts. If the length of the smaller parts is short enough, the alignment becomes feasible again. The challenge is to split up the complete sequence in a reasonable way.

One promising solution appears to be the exploitation of the already available call tree structure of an application run. Figure 4 shows how this approach works. Depicted are the required steps to generate a complete alignment.

While traversing the call tree, sequences are build for sub-function calls of each function only. This cuts down the complete sequence length significantly. The alignment of sub-function call sequences requires no extensive calculation time anymore. Figure 6 gives an overview of the result of the hierarchical alignment applied to the example in Section III-A. The assembly of the sub-alignments to the final flat alignment follows the call tree structure and is also depicted in Figure 6. As can be seen, the results for the flat and the hierarchical alignment are almost equal. There is a small difference though. The gap area in the flat alignment is one element wider than in the hierarchical alignment. The reason is that the jump back from the function *c* to the functions *sub* or *s* is counted as gap in the flat alignment. This is correct because for the flat alignment the second gap is needed to align the two sequences properly. In case of the hierarchical approach the sequences are aligned accordingly the call tree structure. Here, the second gap is not required. In the end both methods produce the same alignment for the given example sequences.

If non-matching areas are detected, our algorithm stops traversing deeper into the call tree of the respective function. All sub-function calls of this function are marked as dissimilar. This prevents unnecessary alignment steps and could improve the performance of the hierarchical alignment algorithm.

IV. Trace Alignment Evaluation

This section evaluates the proposed hierarchical algorithm. The time required to calculate an alignment is discussed. Additionally, the correctness of the resulting alignment is demonstrated by the comparison of multiple runs of two HPC applications.

A. Alignment Performance

The measurement of the performance of the flat alignment approach is depicted in Figure 7. The graph shows the expected quadratic behavior of a dynamic programming algorithm. Most important is the fact, that the compute time required for the alignment of two sequences of length 20,000

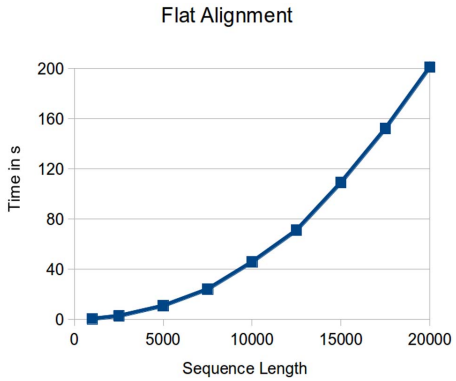


Fig. 7. Performance of the flat alignment

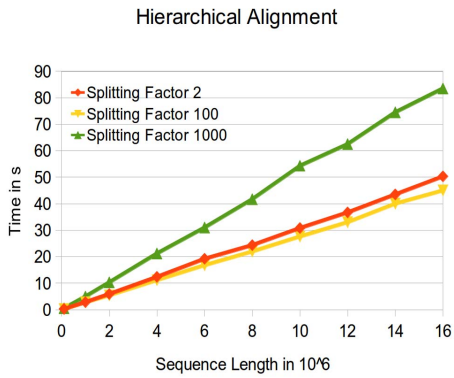


Fig. 8. Performance of the hierarchical alignment

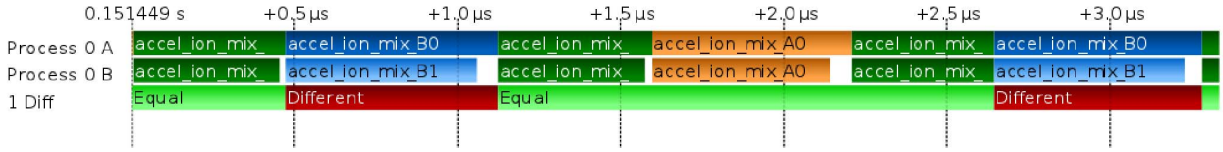
already exceeds three minutes. In practice even short measurement runs of applications easily produce sequences with of more than 100,000 elements. This renders the flat alignment algorithm infeasible for practical use. The calculation of the alignment of real applications would take much longer than the actual application runs.

The performance of the hierarchical alignment algorithm, described in more detail in Section III-B, is primarily influenced by two parts. The first major part is the construction and the traversal of the call tree. For this step the algorithm builds a call tree for each application sequence and then traverses them in parallel. This part can be computed in linear time. During the traversal of both trees all corresponding nodes are compared. In this step the sub-call sequences held by each node need to be aligned. This task forms the second performance relevant part of the algorithm. The alignment of the sub-call sequences is computed with the Hirschberg algorithm. This step has quadratic complexity. As a result the performance of the hierarchical algorithm not only depends on the application sequence lengths, but also on the sub-call structure of the application. A function call is represented by one node in the tree. The sub-sequence length of a node is given by the number of sub-functions called by the represented function. Critical are long sub-sequence lengths. Since the sub-sequences are aligned with the Hirschberg algorithm, its quadratic complexity could dominate the performance if the sub-sequences are very long. To provide performance characteristics for the hierarchical algorithm, a benchmark application has been written. This application generates two function sequences of variable length. The sequences are assembled by multiple sub-sequences of similar length. The length of the sub-sequences is given by the splitting factor. A splitting factor of, e.g., 100 means, that a function sequence of x length is constructed by multiple sub-sequences of 100 elements length. The generated two function sequences are then aligned using the hierarchical alignment algorithm. The benchmark application allows to analyze how varying sub-sequence lengths influence the algorithm performance. Figure 8 depicts the measurements using the benchmark application. The graphs show, that the hierarchical algorithm calculates the alignment of 16,000,000 elements in less than a minute. This allows to compare even long application runs in reasonable time. The results depend on the splitting factor, i.e., the sub-sequence length. One cause for long sub-sequence lengths is functions that are called in a loop. If such a loop would create a sufficiently high sub-sequence length, i.e., above several tens of thousands, the algorithm shows the performance characteristics of the flat alignment. Yet, many applications do not produce high sub-sequence lengths. For instance the Semtex [11] application, used as example in the next two sections, generates only sub-sequence lengths ranging from less than ten to a few hundred elements. For such applications the influence of the parts with quadratic complexity is only marginal and the hierarchical algorithm computes the alignment with linear time complexity. In these cases the proposed algorithm allows the comparison of application event traces with small computational effort.

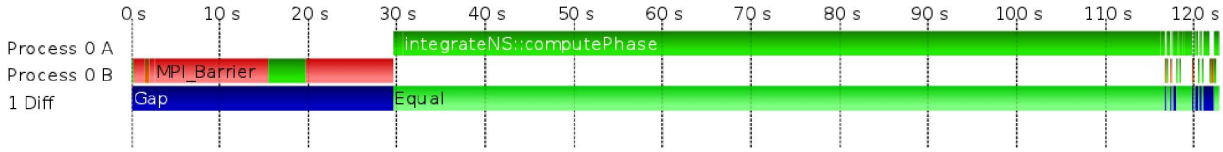
B. Alignment Correctness

The correctness of the computed alignments needs to be evaluated with respect its to suitability for trace file comparison. Therefore, the hierarchical alignment algorithm is used to compare trace files of two real applications. In the given scenarios the correct alignment and detection of gaps and differences is tested. The flat alignment algorithm is not evaluated due to its large computational demand.

The first application used for evaluation is Semtex [11], a classical quadrilateral spectral element DNS code for numerical fluid mechanics. The Semtex code can be run in an MPI parallel and a serial version. Using the same input data we recorded a parallel and a serial application run. The hierarchical alignment algorithm is used to compare rank 0 of the MPI version with the serial version. In this comparison we expect gaps that mark the MPI parts that are missing in the serial version. Figure 9b shows the comparison result. Depicted as timelines are the function sequences of the two application runs. *Process 0 A* indicates the serial run and *Process 0 B* indicates rank 0 of the MPI run. The third timeline *1 Diff* indicates the result of the hierarchical alignment algorithm. Here, equal parts and gaps are indicated as green and blue areas, respectively. As can be seen in Figure 9b both



(a) Alignment of two molecular dynamic application runs using different code blocks



(b) Alignment of a serial version (*Process 0 A*) with an MPI parallel version (*Process 0 B*) of Semtex

Fig. 9. Evaluation of the alignment correctness with two examples. In both examples two processes are aligned using the hierarchical alignment algorithm. The alignment result is depicted with the *Diff 1* bars. Example *a* shows the aligned sequences of two MD code runs (*Process 0 A* and *Process 0 B*). Both runs differ only in code blocks *accel_ion_mix_B0* and *accel_ion_mix_B1*. These differences are detected correctly. Example *b* shows the aligned sequences of a serial- and an MPI run of Semtex. The missing MPI parts in the serial version are detected correctly as gaps.

application runs are aligned correctly. The MPI initialization and data distribution in the beginning of the MPI version is marked as gap in the serial version. Then both versions perform calculations. These areas are marked as equal. In the end of the application runs the data gathering and MPI finalization are correctly identified as gaps again.

The second application is a molecular dynamics code (MD code) [12] developed at Indiana University. The code simulates the diffusion in dense nuclear matter in white dwarf stars. It is highly configurable allowing serial, MPI, OpenMP, or hybrid runs and additional fine-tuning with a range of parameters. For the alignment algorithm tests the serial version of the MD code was used. The actual molecular movements are calculated in two code blocks named A and B. For each code block exist several different implementations. Each version is identified with an individual number following the block name, e.g., $A0$, $A1$, $A2$, $B0$, $B1$, etc. We use the hierarchical alignment algorithm to compare two MD code runs using different block combinations. The first MD code run used blocks $A0$ and $B0$. The second run used blocks $A0$ and $B1$. Both runs are completely identical except for the described block combinations. Figure 9a shows the comparison of both runs. Depicted are the two aligned function sequences of both code runs (*Process 0 A* and *Process 0 B*). Additionally, the third bar (*1 Diff*) indicates the result of the hierarchical alignment algorithm. Equal and differing parts are marked as green and red areas, respectively. For better readability, Figure 9a shows only a short interval of the entire sequences. As can be seen, both application runs are aligned correctly. The B code blocks, *accel_ion_mix_B0* and *accel_ion_mix_B1*, are detected correctly as differences. The other equal functions are matched correctly.

V. Analyzing Execution Time Differences

The described alignment algorithm enables additional comparison opportunities. With the knowledge of the alignment it is possible to match respective function invocations of multiple

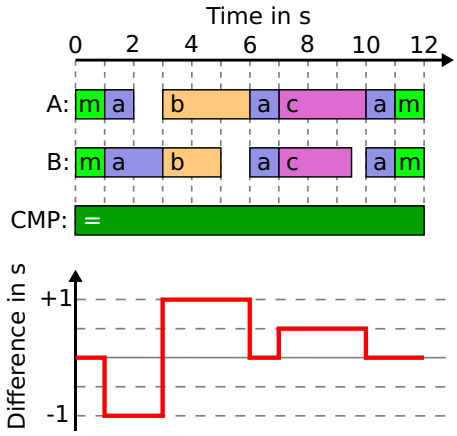


Fig. 10. Generation of an execution time difference timeline

application runs. If functions are matched, their execution time can be compared one by one. Figure 10 depicts this process. Two processes are aligned. Then their function sequences can be arranged together on a timeline. This is done in such a way that all matching functions start at the same time. Then, as depicted in Figure 10, an execution time difference chart can be generated. The values in the chart represent the difference in execution time between matching functions. Positive values indicate that process B runs faster, whereas negative values indicate that process A runs faster.

For evaluation of the proposed methods we performed an analysis using the Semtex [11] application code. We ran the code on two machines in order to extract and analyze execution time differences. Trace A was recorded on an *Intel Itanium II Montecito - 1.6 GHz* system and trace B on an *Intel Xeon E5520 - 2.27 GHz* system. In order to visualize the results we display the alignment and the execution time differences with the tool Vampir [13].

Figure 11 shows the analysis of the Semtex code run.

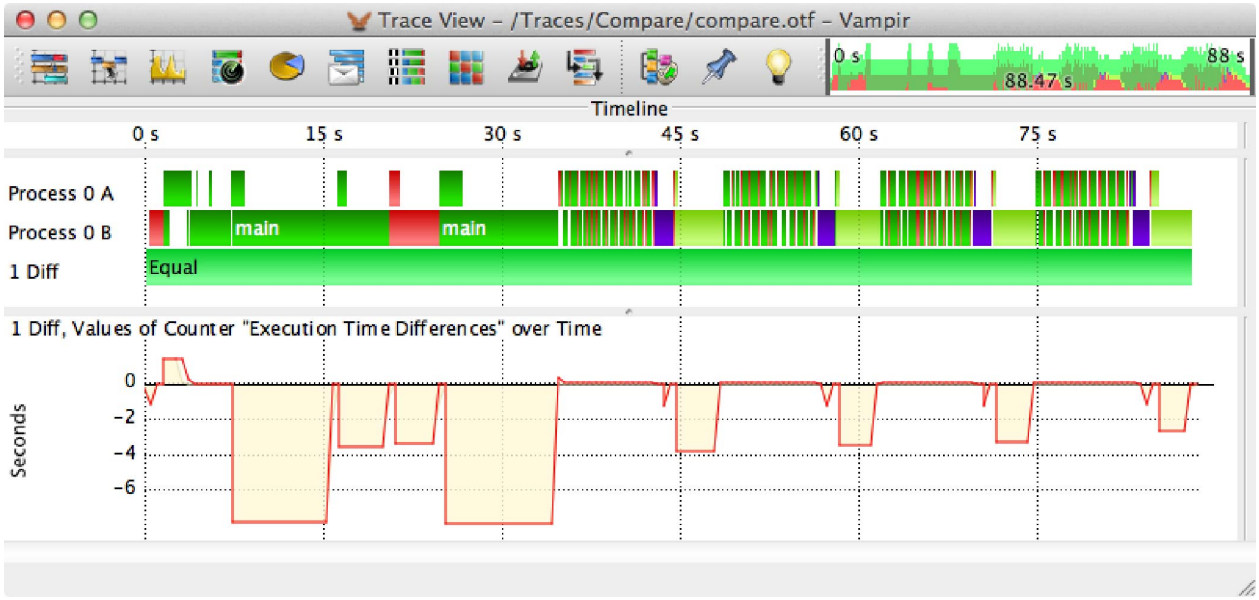


Fig. 11. Alignment of two identical Semtex code runs. *Process 0 A* and *Process 0 B* represent the Semtex run on a Montecito and a Xeon system, respectively. The line chart at the bottom indicates the execution time differences between each matched function pair.

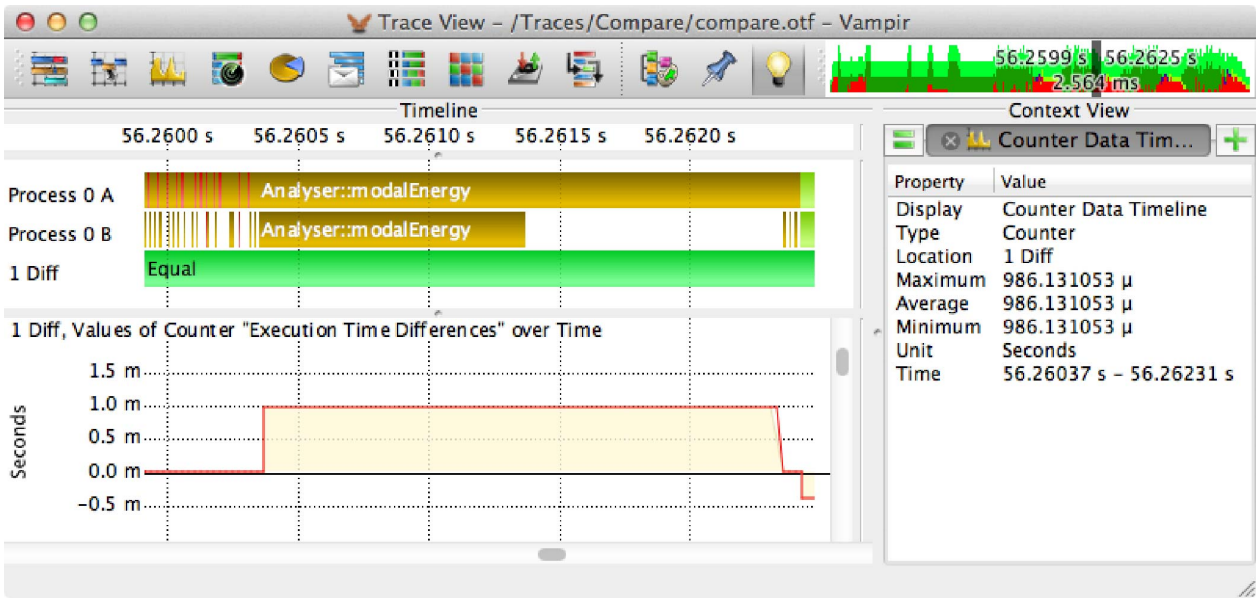


Fig. 12. Execution time difference of one matched *modalEnergy* function pair. On the Xeon system (*Process 0 B*) the function runs 986.1 μs faster.

The timeline chart shows the result of the alignment of both applications. *Process 0 A* represents the application run on the Xeon system, *Process 0 B* the application run on the Montecito system. As can be seen by the additionally added stream *1 Diff*, both processes are identical. This is the expected result for two runs of the same Semtex code.

The duration time of trace A and B is 78.371 seconds and 36.784 seconds, respectively. Hence both runs differ in total execution time by 41.587 seconds. Utilizing the alignment information all individual function calls can be matched. Of course, in this particular example function call matching is trivial, as the call sequences of both application runs are equal. But, as shown in the previous section, our solution also allows the matching of call sequences in nontrivial cases with difference- and gap areas. By comparing the individual execution time of matched function calls, fine-grained differences are extracted. The bottom chart in Figure 11 shows this execution time difference timeline for the Semtex application.

As can be seen in the chart the application runs not entirely faster on one or the other machine. On the contrary, there are functions that run faster on the Xeon system as well as others that perform better on the Montecito system. A closer look reveals that the functions performing output of calculated data are running faster on the Montecito system. A different situation exists for functions performing calculations. There, due to the higher clock frequency and newer architecture, the Xeon processor performs better. An example is shown in Figure 12. The execution time difference chart in the bottom shows that the depicted function invocation *modalEnergy* runs almost 1 ms faster on the Xeon system. The exact time difference of 986.1 μs can be read in the context view located on the right side of Figure 12.

This approach allows the detection of many kinds of performance issues. On the execution timeline chart it is easy to spot outliers that behave differently. Also differences developing over time can be quickly detected.

VI. Conclusion and Future Work

The comparison of event traces has strong potential for the optimization of applications. The tracking of changes in a large amount of trace data can be supported by event-wise matching and comparison.

In this paper we present a novel algorithm for trace file comparison. By combining call graph analysis with a pairwise sequence alignment algorithm our approach is able to automatically detect changes between multiple event traces. We analyze the performance of our implementation with respect to trace file length and sub-sequence lengths. We also test the correctness of the alignment results with real application codes.

An example comparison of the Semtex code proves the usability of our approach for performance optimization. Based on the results of our algorithm, we extract fine-grained execution time differences between multiple traces and present them in a performance timeline chart.

Future tasks include the evaluation of heuristic alignment algorithms. Combining heuristic algorithms with the hierarchical approach might improve its performance and usability. Particularly in the case of long sub-call-sequences the hierarchical algorithm could benefit from heuristics.

Also the analysis of individual processes to enable automatic matching of processes between multiple traces is planned.

References

[1] Intel Trace Analyzer and Collector (website). [Online]. Available: http://software.intel.com/en-us/articles/intel-trace-analyzer/

[2] K. Mohror and K. L. Karavanic, "Evaluating similarity-based trace reduction techniques for scalable performance analysis," in *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, ser. SC '09, Portland, Oregon, 2009, pp. 55:1–55:12.

[3] M. Casas, R. M. Badia, and J. Labarta, "Automatic phase detection and structure extraction of MPI applications," *International Journal of High Performance Computing Applications*, vol. 24, no. 3, pp. 335–360, 2010.

[4] R. Preissl, T. Kockerbauer, M. Schulz, D. Kranzlmuller, B. Supinski, and D. Quinlan, "Detecting patterns in MPI communication traces," in *Parallel Processing, 2008. ICPP '08. 37th International Conference on*, 2008, pp. 230–237.

[5] A. Knüpfer, B. Voigt, W. E. Nagel, and H. Mix, "Visualization of repetitive patterns in event traces," in *Proceedings of the 8th international conference on Applied parallel computing: state of the art in scientific computing*, ser. PARA'06, Umeå, Sweden, 2007, pp. 430–439.

[6] P. Ratn, F. Mueller, B. R. de Supinski, and M. Schulz, "Preserving time in large-scale communication traces," in *Proceedings of the 22nd annual international conference on Supercomputing*, ser. ICS '08, Island of Kos, Greece, 2008, pp. 46–55.

[7] R. Bellman, *Dynamic Programming*. Princeton, NJ, USA: Princeton University Press, 2010.

[8] S. B. Needleman and C. D. Wunsch, "A general method applicable to the search for similarities in the amino acid sequence of two proteins," *Journal of Molecular Biology*, vol. 48, no. 3, pp. 443–453, 1970.

[9] D. Gusfield, "Algorithms on stings, trees, and sequences." *Computer Science and Computational Biology*, 1997.

[10] D. S. Hirschberg, "A linear space algorithm for computing maximal common subsequences," *Commun. ACM*, vol. 18, no. 6, pp. 341–343, June 1975.

[11] Semtex spectral element Fourier code (website). [Online]. Available: http://users.monash.edu.au/~bburn/semtex.html

[12] C. Horowitz, J. Hughto, A. Schneider, and D. Berry, "Neutron star crust and molecular dynamics simulation," 2011.

[13] H. Brunst, H.-C. Hoppe, W. E. Nagel, and M. Winkler, "Performance optimization for large scale computing: The scalable VAMPIR approach," in *International Conference on Computational Science (2)*, 2001, pp. 751–760.