Structural Performance Comparison of Parallel Software Applications

Dissertation

zur Erlangung des akademischen Grades
Doktor rerum naturalium
(Dr. rer. nat.)

vorgelegt an der
Technischen Universität Dresden
Fakultät Informatik

eingereicht von

Matthias Weber, M.Sc.
geboren am 8. April 1980 in Cottbus

Gutachter:
Prof. Dr. rer. nat. Wolfgang E. Nagel, Technische Universität Dresden
Prof. Dr. techn. habil. Dieter Kranzlmüller, Ludwig-Maximilians-Universität München

Tag der Einreichung: 30. August 2016
Tag der Verteidigung: 9. Dezember 2016

Acknowledgments

First and foremost, I want to thank Prof. Dr. Wolfgang E. Nagel for giving me the opportunity to work on this subject and for his continuous support over the last years. Also, I would like to thank Prof. Dr. Dieter Kranzlmüller for serving as the second reviewer.

Special thanks are due to Holger Brunst, Ronny Brendel, Kathryn Mohror, and Martin Schulz for their valuable feedback, ideas, and encouragement. Their knowledge and tremendous support improved this work a lot and helped me through the ups and downs of my dissertation work.

Furthermore, I am grateful to Tobias Hilbrich and Ronny Tschüter for their time and effort spent reading my dissertation, and for providing valuable criticisms and suggestions that helped bringing this dissertation into shape.

I also want to thank all of my colleagues and friends at the Center of Information Services and High Performance Computing for a very nice and pleasant working atmosphere that simply made my life better. It has always been fun to go to work over the last years.

Finally, very special thanks go to my beloved partner Claudia and to my mother Christina and brother Bernd-Uwe for their unconditional long term support. If it weren't for you, the road would have been much more difficult.

Abstract

With rising complexity of high performance computing systems and their parallel software, performance analysis and optimization has become essential in the development of efficient applications. The comparison of performance data is a key operation required in performance analysis. An analyst may conduct different types of comparisons in order to understand the performance properties of an application. One use case is comparing performance data from multiple measurements. Typical examples for such comparisons are before/after comparisons when applying optimizations or changing code versions. Besides comparing performance between multiple runs, also comparing performance characteristics across the parallel execution streams of an application is essential to detect performance problems. This is typically useful to detect imbalances, outliers, or changing runtime behavior during the execution of an application. While such comparisons are straightforward for the aggregated data in performance profiles, only limited solutions exist for comparing event traces. Trace-based analysis, i.e., the collection of fine-grained information on individual application events with timestamps and application context, has proven to be a powerful technique. The detailed performance information included in event traces make them very suitable for performance analysis. However, this level of detail also presents a challenge because it implies a large and overwhelming amount of data. Currently, users need to perform manual comparison of event traces, which is extremely challenging and time consuming because of the large volume of detailed data and the need to correctly line up trace events.

To fill the gap of missing solutions for automatic comparison of event traces, this work proposes a set of techniques that automatically align traces. The alignment allows their structural comparison and the highlighting of differences between them. A set of novel metrics provide the user with an objective measure of the differences between traces, both in terms of differences in the event stream and timing differences across events.

An additional important aspect of trace-based analysis is the visualization of performance data in event timelines. This has proven to be a powerful approach for the detection of various types of performance problems. However, visualization of large numbers of event timelines quickly hits the limits of available display resolution. Likewise, identifying performance problems is challenging in the large amount of visualized performance data. To alleviate these problems this work proposes two new approaches for event timeline visualization. First, novel folding strategies for event timelines facilitate visual scalability and provide powerful overviews of performance data at the same time. Second, this work presents an effective approach that automatically identifies and highlights several types of performance critical sections in an application run. This approach identifies time dominant functions of an application and subsequently uses them to analyze runtime imbalances throughout the application run. Intuitive visualizations present the resulting runtime variations and guide the analyst to performance hot spots.

Evaluations with benchmarks and real-world applications assess all introduced techniques. The effectiveness of the comparison approaches is demonstrated by showing automatically detected performance issues and structural differences between different versions of applications and across parallel execution streams. Case studies showcase the capabilities of the event timeline visualization techniques by demonstrating scalable performance data visualizations and detecting performance problems and code inefficiencies in real-world applications.

Contents

Acknowledgments iii

Abstract v

1 Introduction 1
- 1.1 High Performance Computing 1
- 1.2 Performance Analysis and Optimization 3
- 1.3 Challenges for Performance Data Comparison 5
- 1.4 Contributions 7
  - 1.4.1 Structural Comparison of Process Pairs 7
  - 1.4.2 Alignment-Based Comparison Metrics 8
  - 1.4.3 Structural Comparison of Multiple Processes 8
  - 1.4.4 Visualization Techniques for Event Timelines 8
- 1.5 Organization of This Dissertation 8

2 Background and Related Work 9
- 2.1 Measurement of Performance Data 9
- 2.2 Performance Analysis Tools 11
- 2.3 Analysis and Comparison Techniques 14
  - 2.3.1 Visual Event-Trace Comparison 14
  - 2.3.2 Measurement Management Support 16
  - 2.3.3 Similarity-Based Compression Techniques for Event-Traces 17
  - 2.3.4 Automatic Analysis Techniques for Event-Traces 20
- 2.4 Limitations of Existing Techniques 25

3 Methods for Structural Comparison of Process Pairs 29
- 3.1 Jitter in Timestamp Measurements 29
- 3.2 Comparison of the Flat Function Call Structure 30
  - 3.2.1 Structural Differences between Pairs of Sequences 30
  - 3.2.2 Sequence Alignment Algorithms 30
  - 3.2.3 Speedup of Special Alignment Cases 40
  - 3.2.4 Fast Alignment Heuristic 41
  - 3.2.5 Accuracy of Detection of Structural Differences 41
- 3.3 Accelerating the Alignment by Exploiting Hierarchy in Function Call Structures 42
  - 3.3.1 Hierarchical Alignment Algorithm 42
  - 3.3.2 Scalability Considerations 45
- 3.4 Differences between Flat and Hierarchical Comparisons 46
  - 3.4.1 Alignment Errors Introduced by Hierarchical Approach 46
  - 3.4.2 Comparison of Hierarchical and Flat Alignments 49
- 3.5 Evaluation 52
  - 3.5.1 Performance of Flat Alignment Algorithms 53
  - 3.5.2 Performance of the Hierarchical Alignment Algorithm 58
  - 3.5.3 Accuracy of the Hierarchical Alignment Algorithm 61

4 Alignment-Based Comparison Metrics for Processes 63
4.1 Similarity Metric 63
4.2 Dissimilarity Timeline Metric 64
4.3 Runtime Skew Timeline Metric 65
4.4 Function Time Difference Table 66
4.5 Case Studies 66
  4.5.1 NAS Parallel Benchmark BT 67
  4.5.2 AMG2006 68
  4.5.3 ParaDiS 70

5 Methods for Structural Comparison of Multiple Processes 73
5.1 Pre-Clustering Structurally Similar Processes 73
  5.1.1 Determining Similarity 74
  5.1.2 Grouping Structurally Similar Processes 76
5.2 Alignment of Clustered Processes 81
  5.2.1 Multiple Sequence Alignment 82
  5.2.2 Hierarchical Alignment Algorithm for Multiple Processes 83
5.3 Performance Evaluation 94
  5.3.1 Performance of the Clustering Algorithms 94
  5.3.2 Performance of the Hierarchical Multiple Sequence Alignment Algorithm 97

6 Visualization Techniques for Event Timelines 101
6.1 Timeline Folding Methods 101
  6.1.1 Folding Operations 102
  6.1.2 Use Case Specific Folding 103
  6.1.3 Time Complexity and Scalability 104
  6.1.4 Case Studies 104
6.2 Detection and Visualization of Runtime Imbalances 109
  6.2.1 Identification of Time Dominant Functions 109
  6.2.2 Analysis of Runtime Imbalances 111
  6.2.3 Visualization of Runtime Imbalances 112
  6.2.4 Case Studies 112

7 Conclusions and Future Work 119

List of Figures 121

List of Tables 123

Bibliography 125

1 Introduction

This work is centered in the field of High Performance Computing (HPC). Performance optimization of parallel applications is critical in this field to efficiently use available HPC resources. The comparison of performance data is an essential part of this optimization process. Especially the detailed structural comparison of performance data is still very cumbersome and involves manual comparison and alignment of related application sections. This work describes alignment-based solutions for automatic structural comparison of performance data and thereby fills a gap of missing comparison functionalities.

After a short introduction to HPC and performance analysis, this chapter describes open challenges for structural comparison of parallel applications, followed by a summary of the contributions of this work and an overview of the subsequent chapters.

1.1 High Performance Computing

Computers are general purpose devices that can be programmed to execute selected arithmetical and logical operations. Early computers have been large machines filling entire rooms [42,119]. Those machines were available to a limited group of people only, and performed specialized tasks. With rising capability and gradual miniaturization computers have become omnipresent over the last decades. Today, computers are found in many devices surrounding us, such as mobile devices, home appliances, or cars. In the majority of cases small embedded computers are used to carry out dedicated tasks. Also very common are general purpose computers, found in laptops or workstations. Besides building small devices and general purpose computers, humans always have been putting great effort into creating the most powerful computers possible with currently available technology. Such machines, so-called *supercomputers*, still fill entire rooms today. The computer itself along with the required supporting machinery for cooling, power supply, etc. usually require a dedicated building. With the capability of executing large numbers of operations as fast as possible, supercomputers are employed to solve complex and computationally intensive problems. The field of high performance computing (HPC) refers to all activities related to supercomputers, from the development of software to the design and construction of supercomputers. In the last few decades, HPC has been a critical factor for achieving and preserving scientific and economic competitiveness.

Main driver for HPC over the last 50 years has been an exponential increase in computational speed. This growth, related with Moore's Law [102], has been accompanied with a continuous increase of memory and storage capacities. This enormous increase of available computing power has enabled unprecedented advancements in science and engineering.

One example are simulations that try to model nature as accurately as possible. Over the years, simulations have become increasingly powerful and complex. Continuous improvements included higher model resolution, longer simulation time, and incorporation of more sub-models. Figure 1.1 gives an example showing the enhancements of climate simulations in the last decades.

Today, computational simulation is established as the third pillar of scientific methodology alongside theory and experiment. Simulations of physical, biological, and chemical processes allow insight when experiments are too expensive, dangerous, or infeasible. Famous examples of such simulations are the computation of molecular interactions in stars or simulations of the earth's climate covering time spans of tens of thousands of years.

Besides improvements to existing applications, the growth of available computing power made new application domains possible. Application fields such as computational genomics or bioinformatics had been considered infeasible two decades ago. Nowadays, supercomputers are used to solve problems in

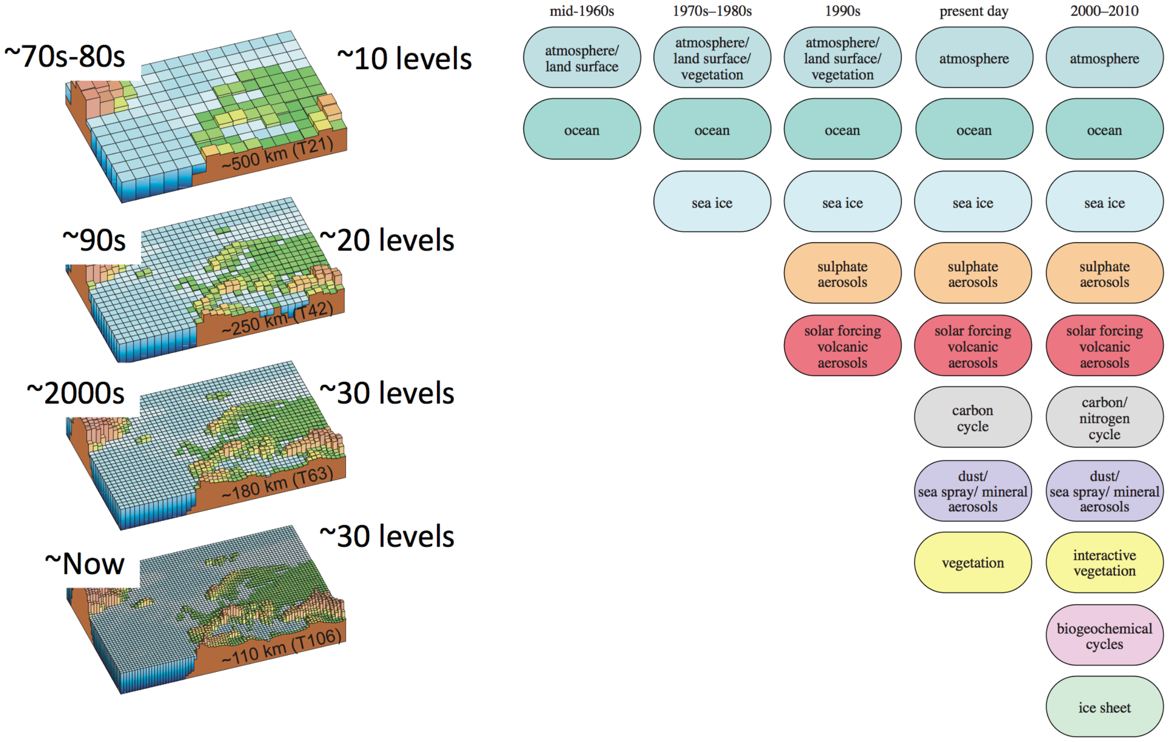


Figure 1.1: History of horizontal resolution and complexity in climate models used for century scale simulations. The image on the left depicts the increase of the horizontal resolution used in climate models during the last decades. The image on the right lists the components of the Earth system added to climate models during the last decades [80, 139].

various fields, including physical simulations, oil and gas reservoir exploration, molecular modeling, or quantum mechanics. Moreover, scientists already anticipate steady improvements in compute capability to generate scientific advancements.

To satisfy the need for more computing power, ever-more powerful machines have been developed. As described by Moore’s Law the ever-shrinking cost of transistors combined with engineering advances of the processor vendors resulted in an exponential performance improvement of microprocessors. First, this performance improvement resulted directly in higher processing speeds of the microprocessors. On the software side, this effect resulted in direct performance improvement by simply upgrading the hardware. Then, in the beginning of the 21st century, physical limits inhibited further advancement of the clock speeds. Since then, additional transistors (which are no longer exclusively used for performance improvements of a single processor) have been used to build extra processors. This change resulted in an ever-increasing amount of parallelism in supercomputers. Figure 1.2 shows an example of the increasing parallelism using the last two supercomputers at the Lawrence Livermore National Laboratory (LLNL), USA. In four and a half years the parallelism between both systems increased by a factor of 7.4. In addition to the rising parallelism, today’s supercomputers increasingly utilize hardware accelerators. The system *Tianhe-2* at the National Super Computer Center in Guangzhou, China—No. 1 on the TOP500 list from June 2013 to November 2015—is a hybrid system consisting of 32,000 Intel Xeon processors and 48,000 Xeon Phi accelerators resulting in a total of 3,120,000 compute cores [131].

Increasing heterogeneity and parallelism result in rising complexity of HPC systems. Since almost all computing power improvements of current supercomputers stem from increased parallelism, applications need to exploit parallelism to achieve higher performance. However, the complexity of parallel programs is much higher than the complexity of sequential programs. Due to the current trend of rising complexity in both hardware and software, writing efficient parallel applications gets increasingly harder. Thus, performance analysis is necessary to efficiently exploit HPC systems.

IBM BlueGene/L – 212,992 cores
No. 1 on TOP500 list (November 2007)
Linpack Performance: 478.2 TFlop/s [131]

IBM BlueGene/Q – 1,572,864 cores
No. 1 on TOP500 list (June 2012)
Linpack Performance: 17,173.2 TFlop/s [131]

Figure 1.2: Comparison of the last two IBM BlueGene systems installed at Lawrence Livermore National Laboratory (LLNL). Both systems reached No. 1 in the TOP500 list of the world's fastest supercomputers in November 2007 and June 2012, respectively. Compared to the four and a half year older BlueGene/L system the current BlueGene/Q system provides substantially increased parallelism and computing power [81].

1.2 Performance Analysis and Optimization

HPC systems provide a theoretical peak performance in terms of floating point rate. However, as the name theoretical suggests, actual production codes achieve only a fraction of that performance in practice. Even highly optimized benchmark codes with considerable adaption for an individual system fall short of achieving theoretical peak performance. The Linpack benchmark [30], widely used to measure performance of HPC systems, primarily achieves only about 60%-85% of the theoretical peak performance across top 10 systems [29, 131]. Real scientific applications usually achieve less sustained performance. For instance, highly optimized applications, nominated for the Gordon Bell Prize—an award presented yearly for outstanding achievements in HPC applications—achieve only 55%[1] or 26%[2] of the theoretical peak performance. Yet, for the majority of scientific applications their fractions are even below these numbers.

The performance optimization goal is to achieve as much of the theoretical peak performance as possible. However, this is no trivial task. The first viable part is to design and implement computationally efficient code, e.g., by choosing the best performing algorithm for a given problem. The second viable part to increase efficiency is to adapt the software to the particular hardware. Yet, modern computing systems are very complex, due to the addition of numerous performance optimization techniques. Current systems may include complex memory access behavior via multi-level caches, pipelined instruction execution (including complex dependencies), branch prediction, speculative execution, and parallel execution features. All of these concepts provide considerable performance gains. However, failing to fully utilize any one of these concepts will result in loss of a noticeable fraction of the theoretical peak performance. Additionally, HPC machines are usually individual systems that require individual adaption strategies. The top HPC systems are replaced every three to five years with new systems, that may exhibit significant architectural differences. Thus, performance optimization of parallel software is a continuous and challenging task.

HPC systems exhibit a high degree of parallelism. Applications need to exploit that parallelism to fully utilize available resources. Therefore, the calculated problem is split up into sub-tasks (parallelization)

---

[1] *11 PFLOP/s Simulations of Cloud Cavitation Collapse* on the IBM Blue Gene/Q system *Sequoia* [120].

[2] *Radiative Signatures of the Relativistic Kelvin-Helmholtz Instability* on the Cray XK7 system *Titan* [21].

that are distributed across the available processing cores of the parallel machine. The *speedup* S then describes the relative performance improvement. In theory, the ideal parallelization of an application would result in a linear speedup, i.e., an application divided into ten parts computed in parallel on ten compute cores should run ten times faster than its sequential version on one core. However, due to synchronization and data dependencies, it might not be possible to parallelize all parts of an application. For instance, a simulation may first need to load input data before it can start with parallel calculations, or some calculations may require results of other calculations. In such cases, according to Amdahl's Law [5], the sequential fraction of the application limits the maximal possible speedup. An alternative approach, suggested by Gustafson [50], to measure parallel performance improvement, is to increase the problem size along with rising compute core counts. An application version computing a problem twice the size than the initial version, on twice as many processors should still require the same runtime as the initial version. However, applications rarely achieve ideal speedups. Usually, with rising process counts and problem sizes, the relative performance improvement decreases. Consequently, the *scalability* of an application describes in general how well it can exploit parallelism to reduce its runtime and to compute larger problem sizes. Efficient parallelization of codes is no trivial task. The degree of scalability for an application highly depends on the possibilities of partitioning the underling problem. In the easiest case, the problem can be split up into many independent sub-tasks. In practice, HPC systems often solve problems that can only be separated into sub-tasks that depend on each other. For instance in climate simulations the simulated area is split up into small blocks. Each processor calculates the climate in its block. For the calculation of its block each processor additionally needs the results from the neighboring blocks. The dependence between sub-tasks introduces the need for communication. This dependence also requires synchronization between sub-tasks. Ideally, the problem is split up into completely equal sub-tasks. In practice, this is rarely possible. Using the example of a climate simulation: Even if the block sizes are completely similar, the individual workload also depends on the simulated weather conditions inside the block. As a result, some processes (calculating "faster" blocks) will need to wait for the result of other processes (calculating "slower" blocks) before they can continue with their computations. Consequently, communication and synchronization can severely limit scalability and thus reduce application efficiency.

To evaluate how good an application is adapted to the available hardware and how efficient the parallelization is implemented, performance analysis is necessary. Yet, to assess the performance of an application and to identify performance problems, it is not sufficient to merely measure the application's execution time. Application runtime alone reveals only few details about the complex execution behavior of parallel applications. Performance problems can have multiple causes that may be related to sequential and parallel behavior of arithmetic operations, memory accesses, I/O, communication, synchronization, and load balancing. Users require detailed insight into the application runtime behavior to reasonably evaluate the application's performance. However, the complexity of many HPC applications renders a manual analysis cumbersome at best. For instance some HPC applications, like Trinity [55, 136] (a tool for de novo reconstruction of transcriptomes from RNA sequencing data), consist of a set of linked independent software components. Each component represents an own application itself. Results are passed from one component to the next. It is not unusual that components programmed in different programming languages or using different parallelization paradigms are linked together. The components in Trinity are programmed using C++, Java, or Python. The code connecting all components of Trinity is written in Perl. To evaluate the performance of such applications, analysts first need to find a way to measure how individual components contribute to the overall application runtime. Ideally, analysts need to assess more parameters, like the effectiveness of the parallelization and the memory consumption of each component. Then, after this initial analysis, promising components for performance analysis can be chosen. Only after this step starts the actual search for performance bottlenecks of individual components. Then, after performance problems are detected and fixed, the optimized component's performance and the complete application's performance need to be compared to the performance of the initial application version. Manually conducting and analyzing these numerous measurements is challenging and time consuming.

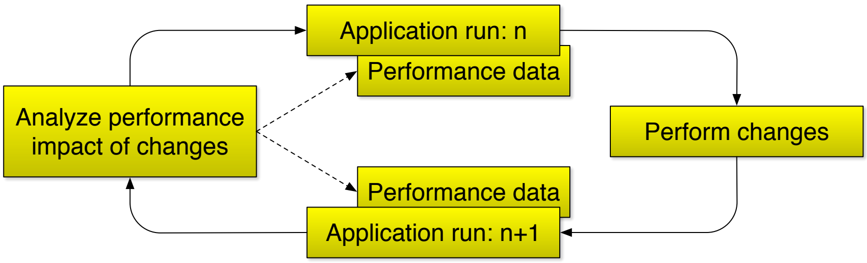


Figure 1.3: Performance optimization is a repeating process.

Therefore analysts rely on performance analysis tools to support and alleviate their work. These tools manage measurements throughout the application's execution and visualize the recorded data in profile and timeline charts. Profiles provide a statistical overview of performance critical parts of an application. Timeline visualizations allow insight into dynamic runtime behavior. Analysis of this performance data allows users to evaluate individual application executions and to find potential code sections for optimizations. Moreover, detailed performance data of an application run allows to detect complex performance issues and to identify causes of insufficient performance. Typical HPC applications do not pose the problem of comprehensively analyzing multiple linked components. They usually consist of only a single binary that is executed with a high degree of parallelism. When optimizing such applications, the user needs to analyze the behavior of hundreds of thousands of processes [37, 150, 151]. Manually conducting and analyzing measurements for such applications is not feasible anymore. Analysts require tools to manage and process large-scale measurements. In fact, it is challenging for performance analysis tools also, to keep up with the scale of applications and still produce meaningful results [150]. The amount of measurement data increases along with application scale. In order to support large scale applications, analysis tools need to improve and adapt their underlying technology [78]. To provide meaningful visualizations of large data sets, tools cannot afford to simply visualize everything, but need to focus on relevant information [91].

1.3 Challenges for Performance Data Comparison

Performance optimization is usually an iterative process. As depicted in Figure 1.3 users first start with an initial measurement as a baseline for comparison. Then, they try to find and optimize inefficient parts of an application and conduct a new measurement. A comparison of the two measurements shows the effect of the optimization. Depending on the result of the comparison, the user may choose to change or add new optimizations and conduct new performance measurements for further comparisons. Without such comparisons, it is challenging to evaluate the performance impact of applied changes. Consequently, comparison of performance data is a key operation in performance analysis. Scenarios for comparisons include:

- Before/after comparisons when applying optimizations or changing code versions,
- Comparisons of runs on different platforms to study performance portability, and
- Contrasting the performance of different processes to study load balance or synchronization delays.

Performance data is usually stored in form of profiles or traces. Profiles aggregate data during collection and provide an overview of the performance of an application. In principle, profiles consist of a list of numbers. Each number describes the performance of a specific part of the application, typically a function. This renders the comparison of profiles rather straightforward as related numbers can be compared directly. Numerous tools already support the comparison of profiles [63, 124].

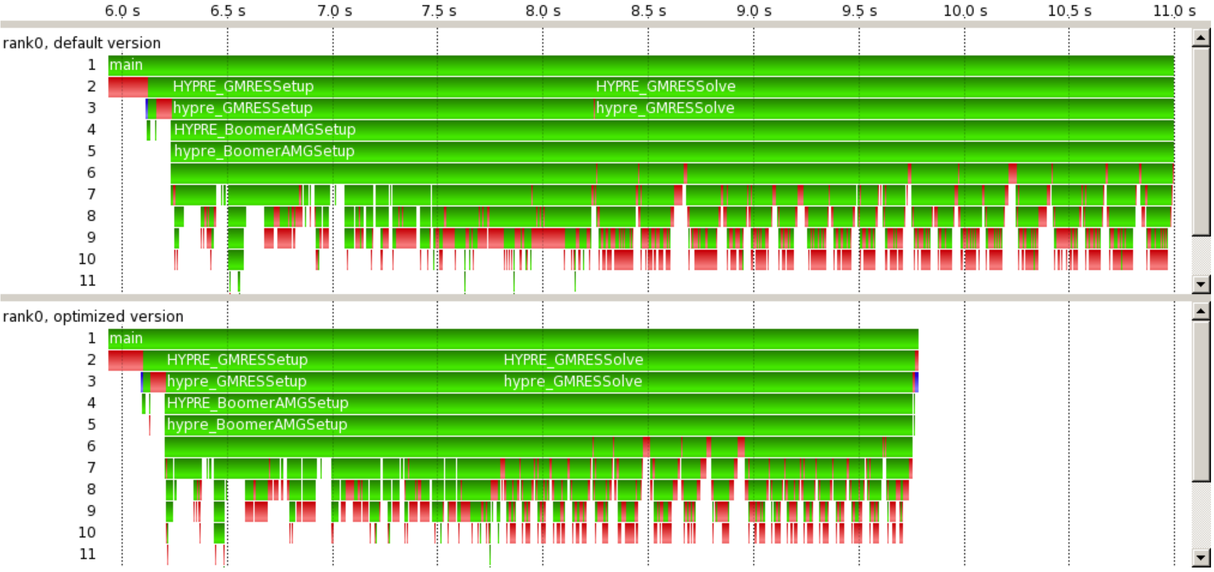


Figure 1.4: Manual visual comparison of two processes. The two timelines contrast a run of the default version (top) with an optimized version (bottom) of an application.

A trace is a sequential record of application behavior. Each application activity, e.g. entering or leaving a function, or sending a message from one process to another, is stored as time-stamped events. Traces allow in-depth insight into the performance behavior of an application. They are especially useful for detecting root causes of performance problems with temporal components. However, the comparison of traces is a challenging task due to the possibly large amount of complex performance data recorded. Figure 1.4 gives an example for this situation. The figure shows two timelines. Each timeline visualizes the trace data recorded on one process (the execution stream running on one compute core) of a parallel application. The top timeline shows a run of the initial version of the application. The bottom timeline shows a run of the optimized version of the application. Each timeline consists of multiple horizontal bars labeled with numbers on the left side. The numbers show the call level of the respective horizontal bar. The colors represent executed functions on the process. Red colors relate to MPI [103] functions (a library providing functionality for communication and synchronization between processes), while green colors relate to functions of the application code. When visually comparing both timelines, it is obvious that the optimized version runs faster than the initial version. The bottom timeline is shorter than the top timeline, thus the optimized version of the application finishes earlier. However, detailed structural differences or performance differences throughout the application run are not immediately visible. To see details, the analyst needs to manually compare both timelines. As the bottom timeline is shorter than the top timeline, related application areas do not appear above each other. For a detailed comparison, the analyst needs to manually align the timelines in order to compare related events. However, manual comparison is extremely challenging due to the large number of events and the need to correctly line up trace events. This renders this task cumbersome and error-prone. Analysts require automatic support for event-wise trace comparison. This dissertation helps to improve this situation by providing automatic alignment-based comparison methods for trace data.

A similar situation arises when comparing multiple processes of a parallel application. Figure 1.5 demonstrates such a comparison case. The figure shows 63 timelines, representing individual application processes. The colors in the timelines represent different functions invoked on the processes. Visible are eight iterations, separated by red vertical blocks. The application invokes MPI calls for communication and synchronization in that area. Each process executes several different functions during its iterations, as the colors between the red areas suggest. Judging if all processes execute the same sequence of functions in their iterations is a tedious task when relying on manual analysis alone. Clearly visible

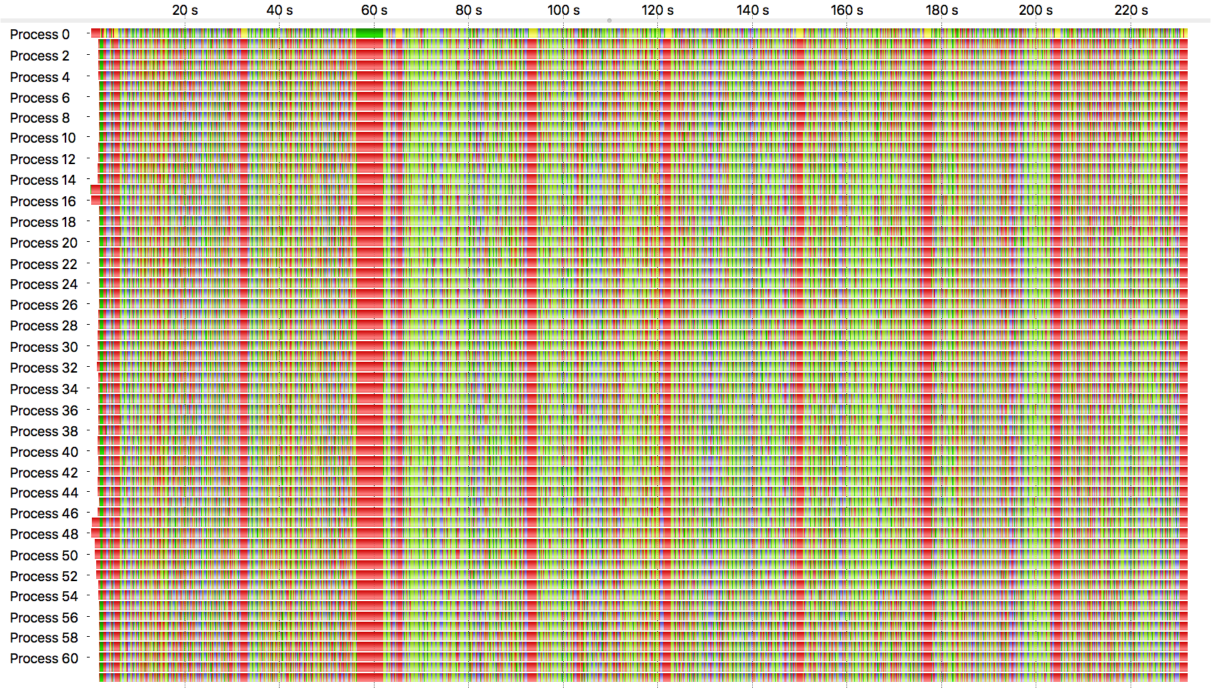


Figure 1.5: Manual visual comparison of multiple processes of one application run.

is that the first process executes different functions than the other processes. But exactly quantifying other differences between processes is time consuming and error-prone. An additional difficulty pose time-shifts and different function durations between the processes, that cause related functions to appear at different horizontal positions between processes. Correctly performing such an analysis manually for larger process counts is almost impossible. To alleviate this task, this work introduces methods that allow an automatic structural comparison of multiple processes.

1.4 Contributions

The comparison of the structure of processes currently needs to be performed manually. Users have to align related events by hand and are required to compare large numbers of events. This work introduces methods that alleviate this cumbersome task for users. Automatic analysis methods allow fast structural comparisons of processes and build the basis for subsequent detailed performance comparisons between processes. This work provides missing functionalities for the task of comparing performance data and additionally introduces novel approaches for the visualization of event timelines. The following contributions are made.

1.4.1 Structural Comparison of Process Pairs

This work introduces a method to compare the event streams of two processes [140]. Based on algorithms from bioinformatics a fast hierarchical sequence alignment algorithm is developed. The novel hierarchical algorithm exploits the function call structure of a process to speed up the required alignment time. With this performance improvement, the alignment, and thus, structural comparison of processes becomes feasible. The introduced hierarchical sequence alignment algorithm allows a detailed detection of equal and differing areas between two processes.

1.4.2 Alignment-Based Comparison Metrics

Based on the novel hierarchical alignment algorithm, new comparison metrics are introduced [144, 145]. These metrics include a definition of the structural similarity between two event streams and provide fine-grained insight into structural and temporal differences between two processes. A case study, applying the alignment-based metrics for the comparison of different versions of several applications, demonstrates their potential for the comparison of parallel applications. The metrics exposed differences that otherwise would have been hard or even impossible to find.

1.4.3 Structural Comparison of Multiple Processes

A new scalable clustering approach enables grouping of large numbers of processes according to their structure [141]. This novel grouping approach for processes has linear time complexity and results in a low number of clusters for many application types. The method is designed as a pre-clustering step for subsequent detailed analysis techniques.

A novel hierarchical multiple sequence alignment algorithm that is capable of aligning large numbers of processes is introduced. Using the pre-clustering result, the algorithm compares processes of one cluster in detail and thereby identifies structural differences and similarities. Key components providing the algorithm speed are the hierarchical approach and a new heuristic that evaluates structural similarity between processes. The algorithm computes a compact data structure, a so-called merged call tree, that combines the structural information of all compared processes and provides rich potential for performance analysis.

1.4.4 Visualization Techniques for Event Timelines

New visualization methods for the analysis and comparison of event timelines are introduced. By applying different folding strategies for timelines, scalable visualizations that facilitate easy detection of performance problems and provide condensed overviews of the performance behavior are demonstrated [142].

Additionally, this work introduces an analysis approach based on performance variations [143]. The approach automatically identifies and highlights performance critical sections. The visualization of performance variations is applicable to guide analysts in the task of identifying performance bottlenecks.

1.5 Organization of This Dissertation

Chapter 2 explains the theoretical background of this work and describes related work. It identifies missing functionalities for the comparison of performance data. Chapter 3 provides solutions for the pairwise structural comparison of two processes. Event streams of two processes are compared using alignment techniques. Chapter 4 introduces performance metrics based on the pairwise alignment techniques described in Chapter 3. The effectiveness of introduced metrics for the comparison of processes is demonstrated in a case study with two applications and one benchmark. Chapter 5 provides approaches for the structural comparison of multiple processes. In a first step, processes are clustered based on their structural information. Then, for detailed comparison, processes of a cluster are aligned using multiple sequence alignment methods. Chapter 6 introduces novel techniques for the scalable visualization of performance data and automatic highlighting of performance hot spots. Chapter 7 summarizes this work and shows possible directions for further developments.

2 Background and Related Work

Performance analysis of software is getting increasingly important. This is especially true for the field of high performance computing (HPC). Rising complexity of both hardware and software requires performance analysis and optimization for the efficient exploitation of current high-end systems. Inefficient applications may waste valuable resources or may only provide reduced capacity due to performance bottlenecks. Consequently, considerable effort is put into performance analysis technology.

This chapter introduces fundamental concepts and tools for performance analysis. It discusses techniques related to performance data comparison and puts a special focus on automatic analysis and compression methods for trace data. It concludes with an analysis of discussed techniques in regard to trace file comparison and lists missing functionality required for a comprehensive comparison of parallel applications.

2.1 Measurement of Performance Data

This section introduces fundamental techniques for the measurement of performance data and discusses trade-offs associated with them.

The performance analysis of an application requires measurements at application runtime. For accurate time measurement systems provide hardware support by high-resolution timers offering precision in the microsecond or nanosecond range. Besides timers, most systems also provide additional hardware performance counters. Such counters may count events like floating point operations, cache misses, instructions, or branch mispredictions.

In order to use hardware counters for measurements, instructions need to be inserted into the application's control flow. This modification can be implemented in several ways.

The first major method to acquire performance data is called *instrumentation*. Instrumentation techniques insert measurement instructions directly into the control flow of an application itself. That way the instructions will be performed in the course of the application's execution. Such inserted instructions may measure time intervals using high-resolution timers or collect values from hardware counters. There are three common techniques used to implement this method of performance data acquisition.

- **Source code instrumentation** adds measurement instructions to the source code of an application. In this way measurement routines are compiled together with the original application code. Usually, this is the first approach used by programmers to debug an application or to produce performance measurements by manually adding instructions to print data or timings to the screen. For the instrumentation of larger code projects, tools that automatically insert measurement instructions into the source code may be used.

  The disadvantage of source code instrumentation is that included measurement instructions are always executed. In order to dynamically enable and disable individual measurements, guard statements need to be placed around measurement instructions. Consequently, even disabled measurements will cause some perturbation during the application execution.

- **Binary instrumentation** adds measurement instructions directly into the application's object code. This saves recompilation of the application. During runtime instructions for measurement are executed along with application instructions. Typically, measurement instructions are inserted into an application's object code by the usage of so-called *trampoline functions*. A trampoline function is an unconditional jump that transfers control to the measurement code that itself performs the actual measurement and returns control back to the application.

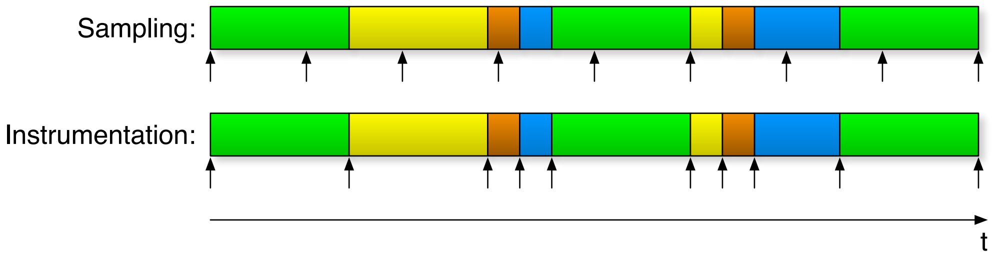


Figure 2.1: Sampling and instrumentation. Both methods differ primarily in the way the measurements are triggered. Colored timelines indicate a series of executed functions. Black arrows below the timelines indicate measurements. In case of sampling, measurements are triggered at periodic time intervals. In case of instrumentation, measurement instructions are embedded into an application's control flow and triggered at function begin and end points.

An advantage of binary instrumentation is that filtered/disabled measurement instructions can be dynamically removed from the object code. Thus, filtered measurements do not induce overhead as they are never executed.

- **Link-level instrumentation** uses the linker to insert measurement instructions into an application's control flow. If an application uses libraries, its calls to these libraries are resolved by the linker. When using link-level instrumentation, a *wrapper library* that implements wrappers for function calls of the target library is linked together with the application. Calls to the target library will first resolve to measurement routines in the wrapper library and then pass on to the target library.

  For statically linked libraries re-linking of the application is necessary. With dynamically linked libraries, it may be possible to load the wrapper library at the application launch, e.g., via the `LD_PRELOAD` environment variable in Linux.

The second method of performance data acquisition is called *sampling*. Sampling issues measurement instructions asynchronously. Measurement instructions are not embedded into an application's control flow but triggered at periodic time intervals. For this purpose most operating systems provide interrupt handlers that interrupt an application's execution at a regular time interval. At each interrupt samples describing where an application spends its time or what resource it uses are taken. Sampling does not require any re-compiling or re-linking and can be directly used with the unmodified application binary.

Besides the different implementation approaches, both methods differ primarily in the way the measurements are triggered. Figure 2.1 depicts the difference between sampling and instrumentation. This difference induces inherent advantages and disadvantages for each method. Instrumentation measures all events as they occur in runtime, since measurement instructions are embedded directly into an application's control flow. An instrumented function is guaranteed to be observed each time it is executed. This ensures a complete coverage. With sampling, as measurements are triggered at asynchronous time points, there is no such guarantee.

In case of instrumentation the amount of measurement overhead depends on the specific application and on how often instrumentation code is executed. The instrumentation of frequently called, short running functions can induce a high perturbation to an application. Typically, individual functions causing a high measurement overhead need to be filtered. Depending on the method of instrumentation, even filtered functions may cause measurement overhead. In case of sampling the measurement overhead is controlled by the sampling rate. A fixed sampling rate allows to exactly estimate the overhead prior to the measurement. The chosen sampling rate is a trade-off between overhead and sampling error. If the sampling rate is too low, infrequent or short events can escape observation. If the sampling rate is too high, an application can be perturbed severely.

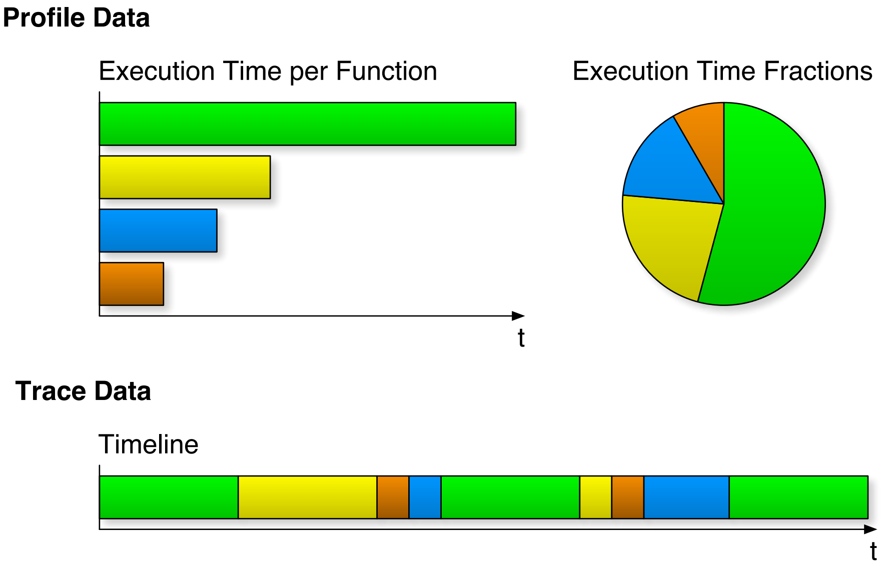
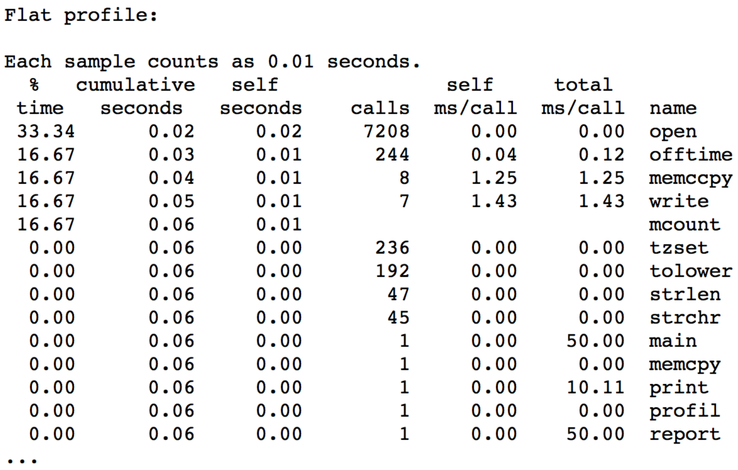


Figure 2.2: Profile and trace data representations. A colored timeline at the bottom indicates a series of executed functions. Trace data allows the complete reconstruction of an application's execution. A typical timeline generated from trace data, indicated at the bottom, displays the complete execution. Profile data aggregates information and provides statistics about an application's execution. Indicated at the top, profile displays present function statistics of the application run indicated at the bottom.

Recorded measurement data is typically stored employing a *profile* or a *trace* data format. Figure 2.2 depicts typical performance data representations generated from profile and trace data respectively. Profiles are an aggregated record of an application's behavior. Due to the data aggregation this approach is scalable. However, the data aggregation also limits the analysis potential of this technique. Performance problems occurring dynamically might not be visible in profile performance data. Profiles are typically used for an overview or a first analysis of an application's performance characteristics. A trace is a record of an application's behavior as a series of events, such as function entry and exit, or message passing. Each event consists of a time-stamp along with relevant data, e.g., function name, or bytes of data transmitted. The detailed information included in traces make them very suitable for detection of many performance problems on HPC systems, e.g., the causes of synchronization delays. However, the level of detail also presents a challenge because it implies a large amount of data from a single run, even exceeding hundreds of megabytes for a single process [97]. A common approach to cope with large trace sizes is to use filtering mechanisms to exclude irrelevant information.

2.2 Performance Analysis Tools

Performance optimization is required to fully exploit available hardware and for the design of efficient code. Often the first step in understanding an application's performance behavior is to manually instrument some parts of the application with available timer calls, like `gettimeofday` on Linux systems, and print the results to the screen. The drawback is that this method only measures user selected code sections and provides no overview of the complete application's performance characteristics. This manual approach is prone to miss important performance information. Also, manual instrumentation tends to be cumbersome, especially with rising code complexity. When analyzing parallel software the manual approach becomes even more involved and the results are harder to analyze. To assist the analyst in this cumbersome task a large number of performance analysis tools have been developed. The tools automate the measurement process, using one or a combination of the above described performance data



```
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  ms/call  ms/call  name
 33.34      0.02     0.02     7208     0.00     0.00  open
 16.67      0.03     0.01      244     0.04     0.12  offtime
 16.67      0.04     0.01        8     1.25     1.25  memccpy
 16.67      0.05     0.01        7     1.43     1.43  write
 16.67      0.06     0.01                             mcount
  0.00      0.06     0.00      236     0.00     0.00  tzset
  0.00      0.06     0.00      192     0.00     0.00  tolower
  0.00      0.06     0.00       47     0.00     0.00  strlen
  0.00      0.06     0.00       45     0.00     0.00  strchr
  0.00      0.06     0.00        1     0.00    50.00  main
  0.00      0.06     0.00        1     0.00     0.00  memcpy
  0.00      0.06     0.00        1     0.00    10.11  print
  0.00      0.06     0.00        1     0.00     0.00  profil
  0.00      0.06     0.00        1     0.00    50.00  report
...
```

Figure 2.3: Example of a flat profile generated by gprof [33]. The table is ordered by the percentage of the total execution time spent in individual functions. The profile provides additional information about the time spent in respective functions and their number of invocations.

acquisition methods, and provide convenient representations of the performance data. Depending on the employed measurement technology and the corresponding performance data representation these tools divide into two major groups.

Profilers The best-known group of performance analysis tools consists of *profilers*. A large number of profiling tools are available and basic profilers are often pre-installed on most operating systems. One example of a common profiler is *gprof* [33, 47]. gprof is distributed along with Linux/Unix systems. The key characteristic of a profiler is the aggregation of the measured performance data. This data aggregation results in a profile, hence the name *profiler*, providing an overview of the performance characteristics of the measurement run. Looking at a performance profile the developer can quickly identify performance critical parts during an application's run. For instance a profile may list the most time consuming functions that thereby present good candidates for performance optimization. Typical representations of a performance profile use bar or pie charts as well as tables, see Figures 2.2 and 2.3.

Figure 2.3 depicts an example of a flat performance profile represented in a table view. A flat profile typically presents average times and frequencies of functions measured during an application run, ignoring the caller-callee relationship of functions. In addition to flat profiles most tools, e.g., gprof [33, 47], Intel VTune Amplifier [64] or AMD CodeXL [4], also support call-graph profiles. A call-graph profile presents average times and frequencies of functions broken down by the call-graph based on the callee.

Profilers use sampling or instrumentation techniques to measure performance data. Like in case of gprof, also hybrid approaches of sampling and instrumentation are employed. Depending on the measurement system profilers may also be able to aggregate performance data from multiple execution streams. This data aggregation provides a scalable method for performance data storage and presentation. Consequently, a range of profilers focusing on the analysis of multiple processes has been developed in the field of parallel computing. These parallel profiles also employ sampling as well as instrumentation techniques. Examples of parallel profilers using sampling are Allinea MAP [2], HPCView [90], or HPCToolkit [1]. Profilers employing the instrumentation approach are, e.g., Cube [129], TAU [9, 61], mpiP [135], or Paradyn [92]. Like shown by Open|SpeedShop [125], also combinations of both techniques, sampling and instrumentation, are employed by parallel profilers.

Usual performance profiles present aggregated data of the complete application run. To achieve a more detailed and differentiated view of the performance behavior, some tools employ a technique called phase-based profiling. The phase-based technique splits the application execution into separate phases. Figure 2.4 shows a phase-based profile generated by TAU [89]. In the depicted example three

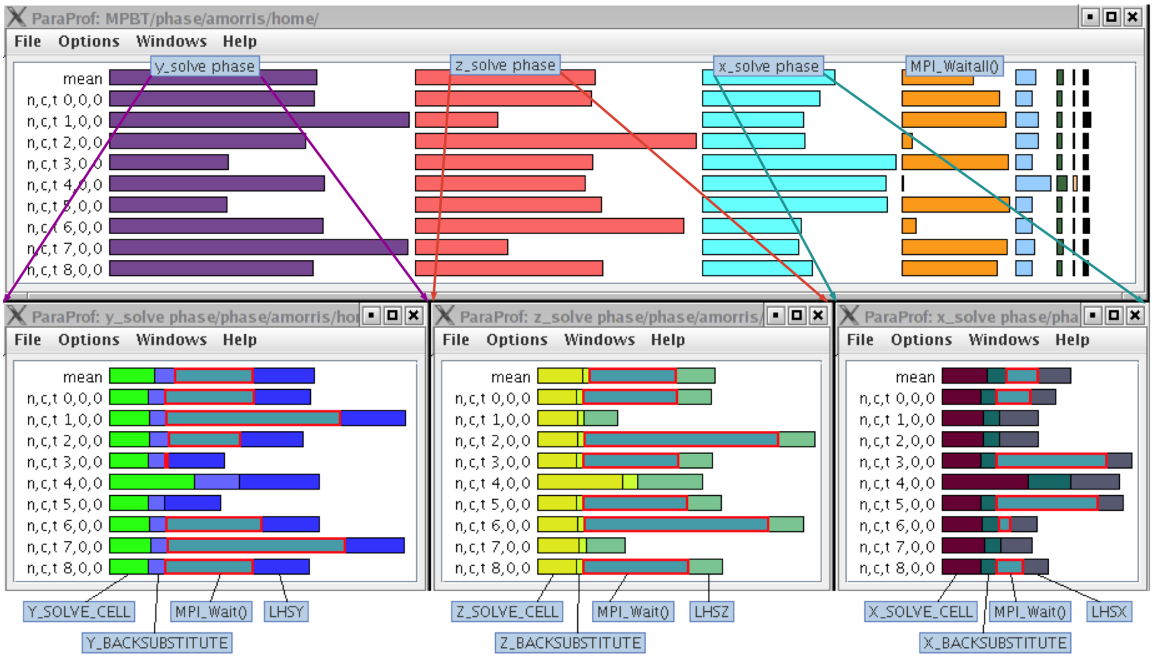


Figure 2.4: Phase-based profile showing individual profiles for x, y, and z solver phases. The top view shows the three phases in a compound way. Functions occurring outside the phases are shown separately. The bottom view shows detailed information for each individual phase. [89]

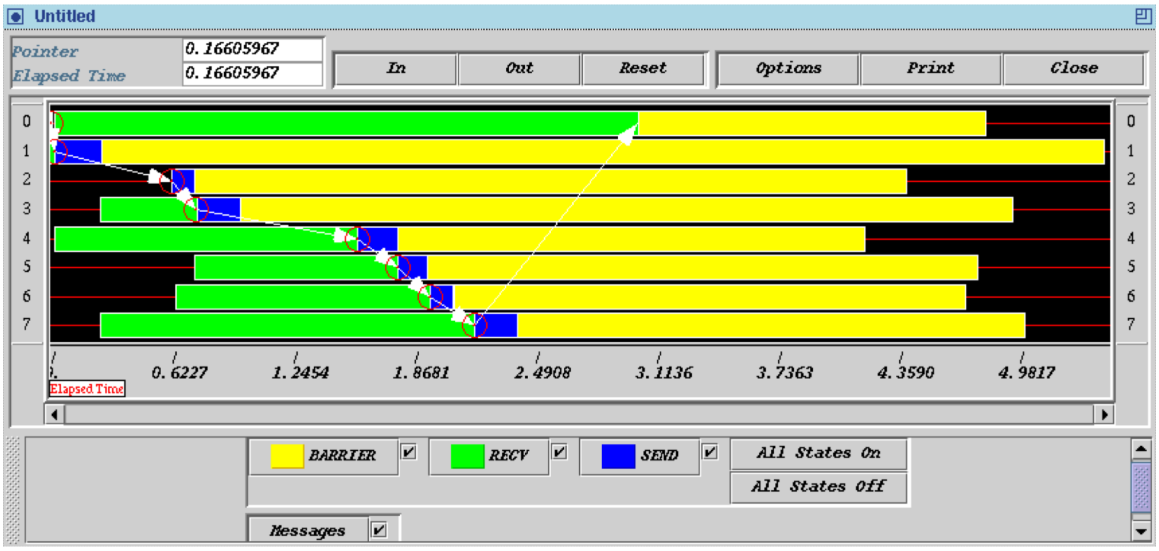
individual phases for the solver have been defined, i.e., `y_solve_phase`, `z_solve_phase`, and `x_solve_phase`. For each phase performance data is aggregated individually. Hence, the application is characterized by multiple profiles, each representing an individual phase. This approach allows a distinct performance analysis of individual code sections.

Tracing Tools *Tracing tools* form the second group of performance analysis tools. These tools allow the most detailed analysis of an application's performance behavior. Contrary to the data aggregation applied by profilers, tracing tools keep all recorded performance data. During the course of an application's execution its behavior is recorded as a stream of events. Additionally to event specific data, like function name or bytes of data transmitted, each event also includes a time stamp indicating its time of occurrence. Tracing tools use this record of events, the trace file, to visualize and analyze the application's behavior. Especially the preservation of all timing information allows the detailed reconstruction of application executions. Moreover, the full timing information is required for the analysis of dynamic performance behavior. For instance, dynamic load imbalances can severely limit performance of parallel applications [13]. Consequently, for performance analysis traces are more powerful than profiles and enable the detection of many performance problems critical to HPC applications.

In order to measure performance data tracing tools employ the methods described in the previous section. Many tools use instrumentation, e.g., Vampir [18, 106], Intel Trace Analyzer and Collector [63], or Jumpshot [28, 149, 152]. Also sampling is applied by some tools, like for instance Paraver [112] or HPCToolkit [1].

Tracing tools typically use timeline views to illustrate the behavior of parallel applications, see Figure 2.5. Timeline views depict the state of each process at any point in time along with the communication between processes. Additionally to process states also performance metrics like values from hardware performance counters may be visualized in timeline views.

Besides timelines tracing tools also compute profiles from event data. In contrast to profiling tools where the profiles cover either the complete application run or defined application phases, the computation of profiles from trace data is more flexible. Due to the timing information included in each stored event, profiles can be computed for arbitrary time intervals.



Pointer 0.16605967
Elapsed Time 0.16605967

Figure 2.5: A timeline view showing a simple ring program executed with eight processes. Each process is depicted as a separate timeline. One message, indicated by white arrows, is passed from one process to the next until it arrives back at the initial sender process. [152]

In addition to performance data visualization some tools provide automatic analysis features. For instance Jumpshot [152] provides detection of function invocations with irregular durations. The detection is driven by the assumption that all invocations of a particular function should run for approximately the same period of time. Using a normal distribution in combination with a high and low cutoff, Jumpshot identifies function invocations with irregular durations. Other examples are Scalasca [38, 148] and Periscope [11]. Both tools automatically detect wait states in parallel applications due to inefficient communication behavior.

The trade-off for more detailed analysis capabilities is a larger volume of performance data. To cope with the possibly large amount of performance data most tools apply filtering methods to exclude irrelevant data. Additionally to filtering, the Vampir tracing framework provides a parallel analysis engine [17, 19], allowing to harness the power of a distributed system for the performance data analysis.

2.3 Analysis and Comparison Techniques

This section provides an overview of analysis techniques for event traces. Additionally, it introduces comparison techniques for performance data. First, tools for visual trace comparison are presented. Then tools for management and comparison of multiple measurement runs are discussed. Last, automatic techniques for compression and analysis of event traces are covered.

2.3.1 Visual Event-Trace Comparison

A straightforward method to compare event traces is to manually perform visual inspection. Users can open several instances of one tool, each showing an individual trace for comparison. Yet, this manual comparison is likely to be extremely challenging and time consuming. Since related events may appear in different places, it involves to correctly line up individual trace events. To alleviate this cumbersome task some tools provide a visual comparison mode. This mode shows multiple traces next to each other for easier comparison.

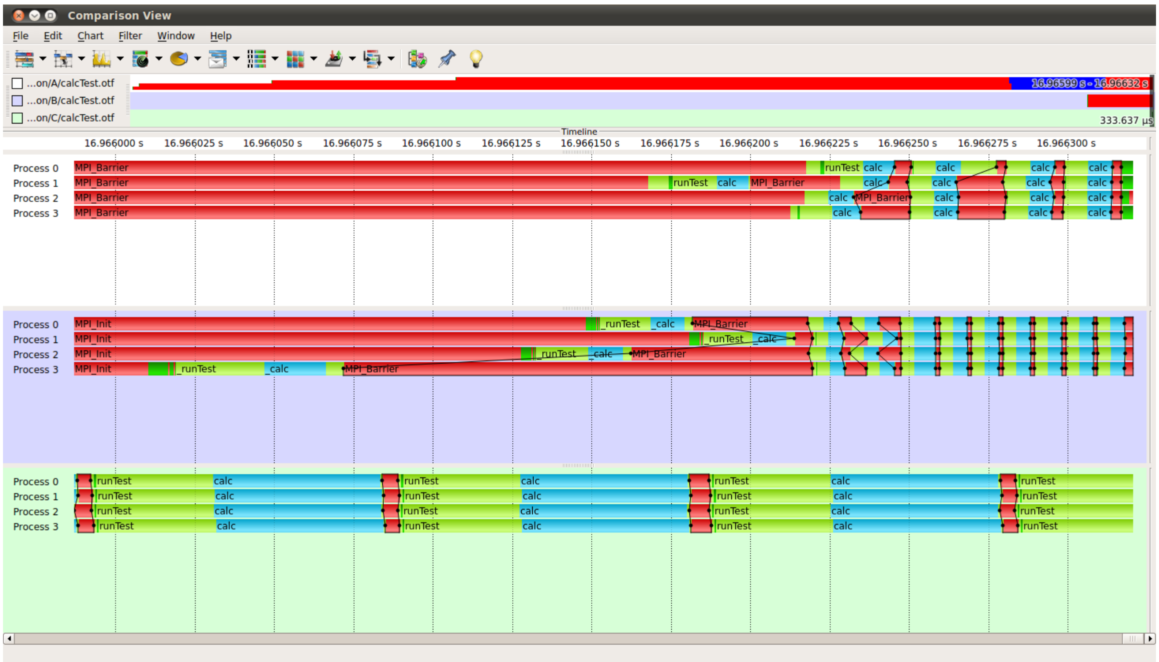


Figure 2.6: Three different traces displayed in the comparison view of Vampir. The traces are distinguishable by their background color: white, light blue, and light green. Each trace shows the same application but measured on a different machine. Green and blue areas in the timelines depict functions performing computations. The different lengths of the computing functions indicate unequal performance of the measured machines. [134]

Vampir The comparison view provided by Vampir [19] arranges multiple traces side by side in one central display. To facilitate detailed comparison of selected events, traces can be individually shifted in time. This enables a manual alignment and compensates for varying start times of events, e.g., due to different initialization durations in the traces. Figure 2.6 depicts the comparison view showing three traces. Each trace represents the same application but measured on a different machine. The figure displays the traces after manual alignment. The compute iterations are aligned next to each other. The runtime differences between the traces become obvious. Compared to the bottom trace, the computations (blue areas in the figure) are considerably faster in the top two traces. Figure 2.6 also shows the limitations of this approach. In case of high runtime differences it may be hard to display events side by side. As shown in Figure 2.6 the iterations of the middle trace barely consume half of the timeline space while the iterations of the bottom trace do not even fit on the available space.

An approach proposed by Knüpfer et al. [77] is based on C3G, an alternative data structure for trace data. When using C3G the trace data is stored in a tree-like graph. Similar repeated events share the same node in the graph. Hence the graph inherently identifies repetitive patterns. Knüpfer exploits this advantage and visualizes repetitive patterns in one process. This method simplifies the detection and visual analysis of patterns inside one trace. Consequently, this method assists in the visual comparison of patterns between multiple traces as well.

Intel Trace Analyzer Similar to Vampir the Intel Trace Analyzer and Collector [63] also provides support for visual trace comparison. The Intel Trace Analyzer limits the comparison to two traces at a time. It offers two modes of operation. The first mode depicts each trace according to its real execution time. The second mode visually stretches the shorter trace to the length of the longer trace. That way both traces appear to have the same total execution time and related areas are shown approximately next to each other. This type of visual display does not show the real trace timings anymore but may help in the visual comparison process. Additionally to the visual timeline comparison, the Intel Trace Analyzer also computes differences between the profile data of both traces.

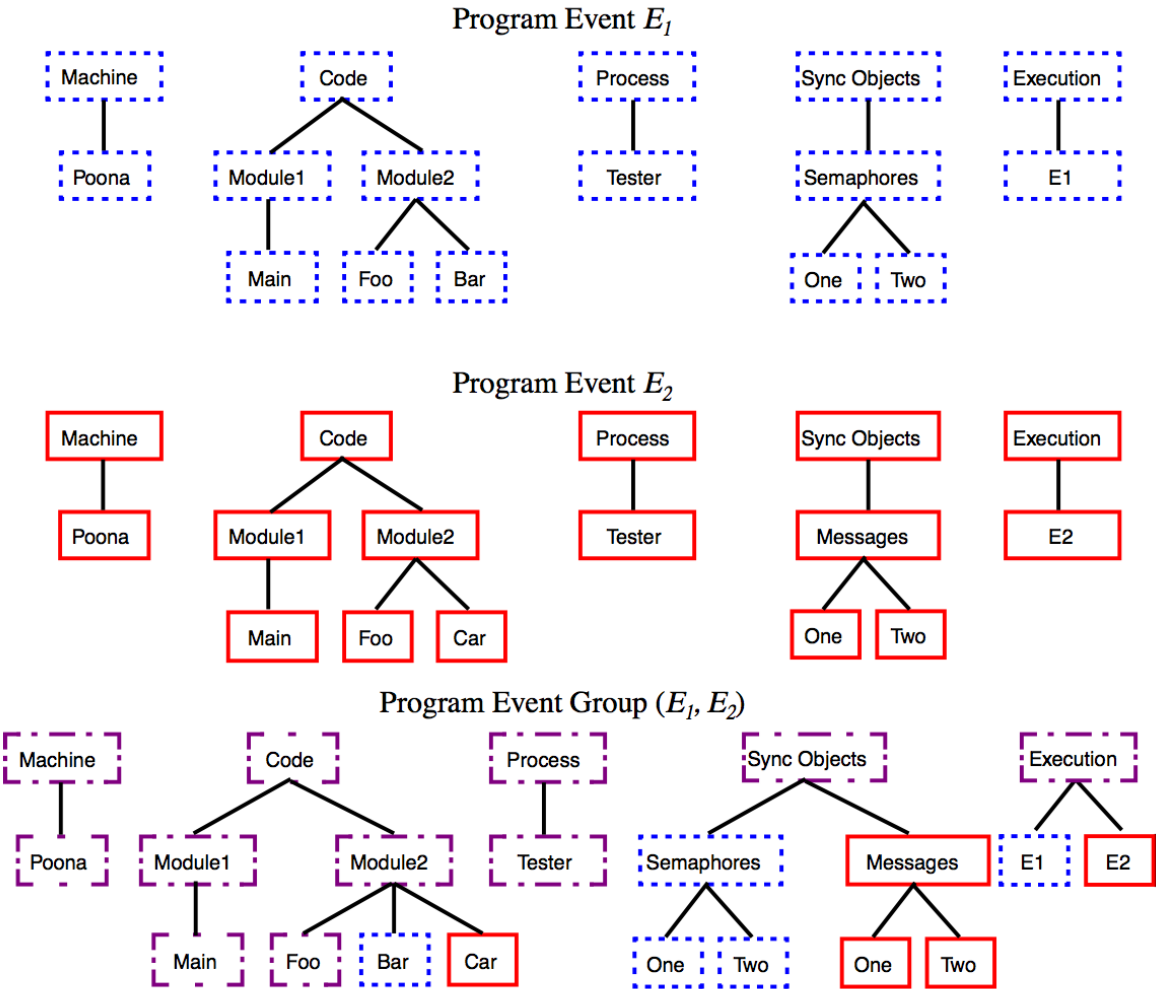


Figure 2.7: Structural comparison of the resource hierarchies of two application executions. Events E_1 and E_2 describe the first and the second application run respectively. The bottom set indicates similarities and differences in the structure between both runs. [69]

2.3.2 Measurement Management Support

Performance optimization of parallel applications usually involves multiple measurement runs. Typical use cases are comparing results of different optimization strategies or studying application portability when moving from one machine to another. Several solutions for managing and analyzing performance data from multiple measurement runs exist.

Differential Profiling The comparison of profiles from two application executions is an essential performance analysis step that users routinely perform. However, most profiling tools do not provide direct support for this analysis task. Yet, the comparison of profiles is straightforward. Usually, first the differences in function runtimes between two application runs are computed. Then the results are sorted from largest to smallest difference. This allows users to quickly identify the key differences between two application executions, typically by just looking at the top functions in the profile. Schulz and de Supinski provide tool support for this approach in their work *Practical Differential Profiling* [124]. They introduce an extension of the commonly used profiler *gprof* [47]. Their tool *eGprof* facilitates comparisons of two performance profiles inside gprof. The tool allows to “subtract” two performance profiles and provides call-graph visualization of the differences.

Experiment Management Support More comprehensive functionality allowing the management of multiple application executions provides the framework for multi-execution performance tuning presented by Karavanic [67, 69, 70]. Each execution of an application is considered as an experiment. The framework defines a *program space* that gathers all information related to one application. The collected information consists of details about all experiments (application runs) like the components of the code executed, the execution environment, and the recorded performance profile data. An experiment management tool facilitates exploration of the program space. The tool allows analyzing differences between multiple application executions. Displayed differences may regard changes in program source code and the resources used at runtime as well as differences in the application's performance. Figure 2.7 depicts a structural comparison between two resource hierarchies. The collection of profile data spanning multiple program executions additionally allows analyzing the performance evolution of one application. The Paradyn performance tool [92] provides an automated search for performance bottlenecks. The incorporation of information from the program space into the Paradyn Performance Consultant [60] allowed to guide the tool's search strategy based on performance data gathered in previous executions. This resulted in a more effective diagnosis of bottlenecks. Later work [68] added database technology for more flexible collection and storage of performance data from multiple locations.

An algebra building upon the described framework for multi-execution performance tuning is proposed by Song et al. [129]. It provides additional arithmetic operations to merge, subtract, and average data. The algebra is used for cross-experiment performance analysis. It allows comparing, integrating, and summarizing performance data from multiple sources. Sources of the performance data may build multiple experiments of MPI and/or multithreaded applications as well as results obtained from simulations and analytical modeling. The algebra represents performance data in a platform-independent fashion. The algebra output data is presented in the same way as the input data, allowing the use of the same set of tools for visualization.

The PerfExplorer tool [61, 62] provides a framework for performance data mining. Measured performance profile data is stored in a database and can be processed with data mining methods. Additionally to clustering and correlation algorithms also comparative analyses are supported. Runtime, relative speedup, and efficiency can be compared across different sets of profiles. PerfExplorer provides tool support for parameter or scalability studies of an application.

Vertical Profiling A work presented by Hauswirth [52, 53] introduces a technique called *Vertical Profiling*. The intention is to record and compare numerous metrics for one application. The metrics cover the entire system including, e.g., hardware, operating system, libraries, and application. For each metric a trace is collected. To control the measurement overhead only a limited number of metrics is recorded during a single application run. Therefore, multiple consecutive measurement runs are taken for the collection of all necessary metrics. The recorded traces are subdivided into successive interval parts. For each interval an aggregated profile value is computed. For the comparison and analysis of the metrics all traces are vertically arranged. Due to the inherent jitter in timing measurements the traces need to be aligned prior to the comparison of related interval parts. The alignment is computed using a dynamic time warping algorithm that requires a common metric in each measurement run. This approach only works for traces measured with the same application configuration and in presence of a suitable common metric for the alignment.

2.3.3 Similarity-Based Compression Techniques for Event-Traces

Traces store an application's behavior as a series of events. Especially the preservation of all timing information enables the detection of a wide range of crucial performance problems. However, this detailed information also results in large data volumes. Performance measurements of parallel applications may lead to unmanageably large trace files. Studies investigated the overhead of tracing on parallel systems [94, 95, 97]. One critical cause of overhead is writing of trace data to disk. Periodic flushes of trace data may cause severe perturbation in the target application. Additionally, the overhead for

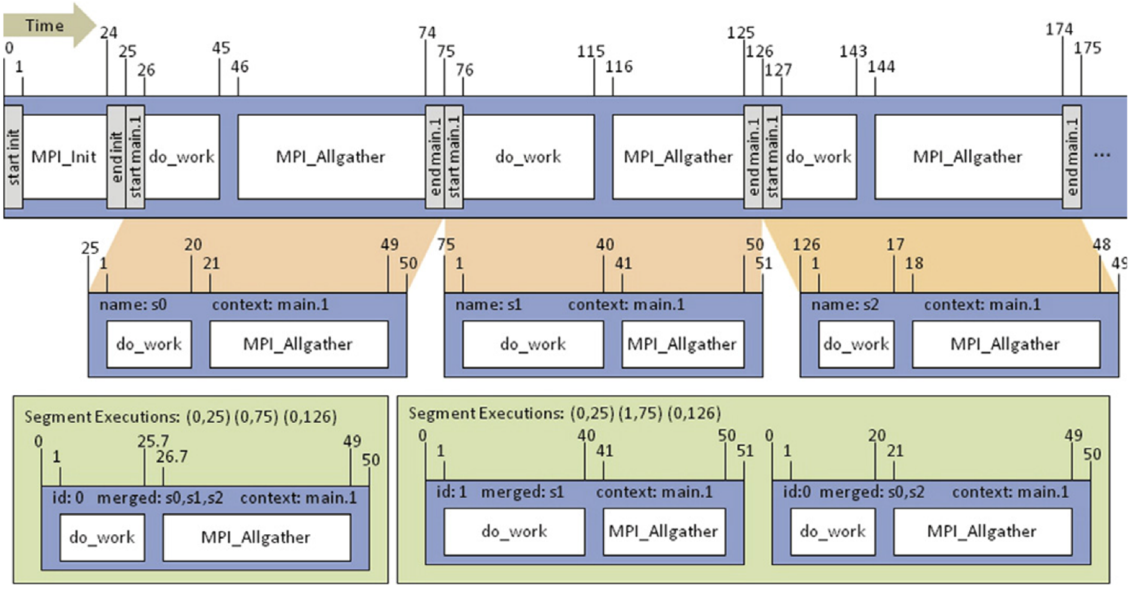


Figure 2.8: Intra-process segment matching scheme. The top bar represents a portion of an example trace with time values indicated above the trace. Segment markers are shown as light gray rectangles. The three resulting segments are displayed below the trace (s0, s1, and s2). In each segment the time stamps are adjusted relative to the segment start time. The bottom row shows two examples of segment matching. In the left example all three segments are merged together. In the right example the threshold is smaller allowing less variation. Hence, the segment s1 is incompatible and only segments s0 and s2 are merged together. [101]

writing strongly depends on the file system speed and increases with rising numbers of processors. Consequently, a range of approaches has been developed to compress the trace data at runtime and reduce the data volume required for writing. The compression techniques may target wide traces (large number of processes) and/or long traces (long runtime of the application). In order to achieve high compression rates all techniques aim to exploit similarities. Compression techniques for long traces benefit best from the iterative behavior of many HPC applications by exploiting repetitions in one process. Compression techniques for wide traces are based on the Single Program Multiple Data (SPMD) paradigm and try to exploit similarities across the processes of a parallel application.

Trace Profiling One solution addressing the reduction of overhead of writing traces is *trace profiling* [96, 99–101]. Trace profiling is a hybrid between tracing and profiling. This measurement technique collects summary information about event patterns that occur during program execution. The technique reduces the trace data volume and writes an approximate trace of a complete application's run. The trace retains enough information to diagnose performance problems that traditionally require traces. The tool Scalasca [148] has been used to evaluate the retention of correct performance behaviors, by comparing automatically detected performance bottlenecks between the original trace and the reduced trace.

To accomplish the data reduction loops are used to partition an application into segments or patterns. The segments build the basis for intra- and inter-process comparison. Segments with the same context (equal patterns) are matched. If they have similar durations within a predefined threshold, only one representative segment is saved. Figure 2.8 depicts the intra-process segment matching scheme. Intra-process segment matching is done at runtime, reducing the data volume written to disk. For effective segment matching a study evaluated several similarity metrics for compression [98]. The study indicated that the average wavelet transform method provided the best trade-off between retention of performance trends and file size reduction.

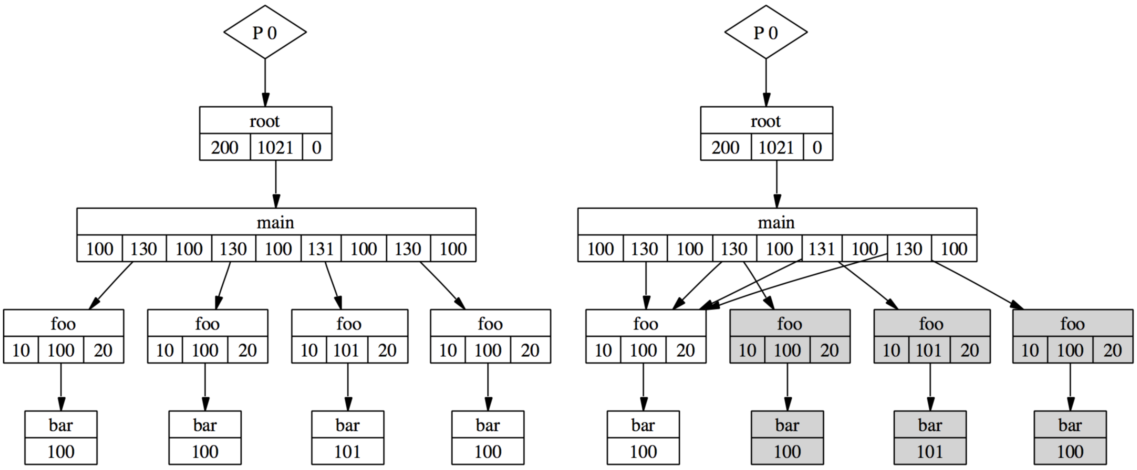


Figure 2.9: Compression scheme of Compressed Complete Call Graphs (C3G). The function `main` calls function `foo` four times. In each invocation function `foo` calls function `bar`. The left hand side depicts the uncompressed call graph. The right hand side depicts the compression scheme. The four `foo,bar` function invocations are compressed to a single invocation, sharing one sub-tree. Note the differing duration of the third `foo,bar` invocation. In this case the difference lies within the deviation bounds and the third invocation is unified with the other invocations. [76]

The second compression step is inter-process segment matching. Processes that have the same number and order of segments are compared. If all segment pairs are similar within a predefined threshold, only one process is kept. Additionally to comparing event measurements, also message-passing parameters are checked. All parameters except source/target rank must be identical. The source/target rank must either be the same rank or the same offset. The trace visualization also benefits from the inter-process matching step. In case of multiple similar processes, only one representative process needs to be visualized.

The trace profiling technique reduces the trace data volume by exploiting the repeated behavior in applications. This compression technique targets long as well as wide traces. A factor of ten for trace reduction has been reported.

Tree-Based Compression Techniques An alternative data structure for potentially lossy compression of trace data is presented by Knüpfer [73, 74, 76]. Unlike the common linear storage approach the described *Compressed Complete Call Graph* (C3G) stores traces in a tree-based data structure. Regular codes benefit from the tree structure. Repeating code sections can share nodes in the tree, resulting in less memory requirement for storage. In order to preserve temporal information while also allowing sharing of nodes, only durations of executed code sections are stored. Figure 2.9 depicts the compression scheme using C3G. The achievable compression rate depends on two major factors. First, the efficiency of the exploitation of the tree structure depends on the level of repetition in the code. Second, for successful reduction of multiple similar code sections to one node, their individual durations need to stay within specified deviation bounds. The size of the tolerated deviation is a trade-off between compression rate and accuracy of the performance data. The described approach builds one C3G structure exclusively for each application process. Consequently, this compression technique only targets long traces. Theoretically, the approach can be extended for compression of wide traces. It is possible to store multiple processes together in one C3G structure. This way nodes and sub-trees could be shared across processes. However, this concept has not been tested in practice.

The tool *ScalaTrace* provides compression of communication traces for parallel applications [104,109, 110,118]. Only the communication patterns of an application are recorded. Specific section descriptors allow a very efficient compression of loops. Communication end-points are encoded using relative distances, e.g., process i communicates with process $i + 1$. The combination of both techniques provides a high lossless intra-node and inter-node compression rate. The first version captured only structural information resulting in near constant trace sizes for applications with regular communication patterns. The collected traces are useful for replay of the communication behavior but have limited value for performance analysis due to the omission of all temporal information. To alleviate this disadvantage a second version additionally stored approximate delta-timings between events using aggregated statistics and path-specific histograms. This compression technique targets long as well as wide traces.

Clustering A compression method targeting wide traces including many processes is clustering. The aim is to group similar behaving processes together. For each group one representative process is selected and stored. All remaining processes only need to refer to their cluster representative. Therefore, the compression factor depends on the number of clusters and the size of the representative processes for each cluster.

Work on this approach is reported by Roth and Nickolayev et al. [108,121]. Performance properties of processes are summarized over a sliding window. Depending on the properties the processes are grouped using a centroid-based clustering algorithm. To adapt the clustering to changing behavior all processes are re-clustered adaptively or in fixed intervals. The approach has been evaluated with real-time compression with up to 128 parallel processes.

Gamblin et al. [35,36] enhanced the method and showed sub-linear scaling for on-line clustering with up to 131,072 processes. Performance data required for the clustering process is recorded only in selected loops and packed using wavelet compression. To ensure short runtime only clusterings for small random sub-groups of all processes are calculated. The clustering results for the sub-groups are calculated in parallel. All results are distributed to all processes for evaluation. The best clustering result is applied to all processes. The clustering process is repeated in fixed intervals. Data volume reductions of up to four orders of magnitude have been reported.

2.3.4 Automatic Analysis Techniques for Event-Traces

Besides technical demands for processing of large trace volumes, the immense amount of data also makes manual analysis difficult. Analysts may easily overlook important performance properties. On the other hand, the comprehensive collection of performance data promises high analysis potential. Consequently, a range of automatic analysis techniques for traces have been developed. The methods either try to help the analyst by automatically categorizing the performance information or by searching for performance problems directly.

Automatic Structure Detection and Analysis In the scope of the CEPBA-Tools Environment [79] a range of new analysis methods for trace files have been developed.

Casas et al. [22,23,25–27] describe methods for automatic structure detection in traces of parallel applications. The approach is based on signal analysis methods and works in three steps, illustrated in Figure 2.10. First, they sample each process of a trace and take values according to a derived metric. The derived metric should describe the application's behavior as clearly as possible. Examples of derived metrics are *Sum of Duration of Computing Bursts* or *Number of Point to Point MPI Calls*. The sampled values are added up across all processes to construct a single suitable signal for the analysis. Areas exhibiting disturbing effects like flushes of trace data to disk are detected and removed from the further analysis steps. The signal is then analyzed using the discrete wavelet transform (DWT). The wavelet transform identifies areas exhibiting high frequencies. These areas most likely represent the iterations of the application, as code is repeatedly executed there. The initialization and finalization phase of

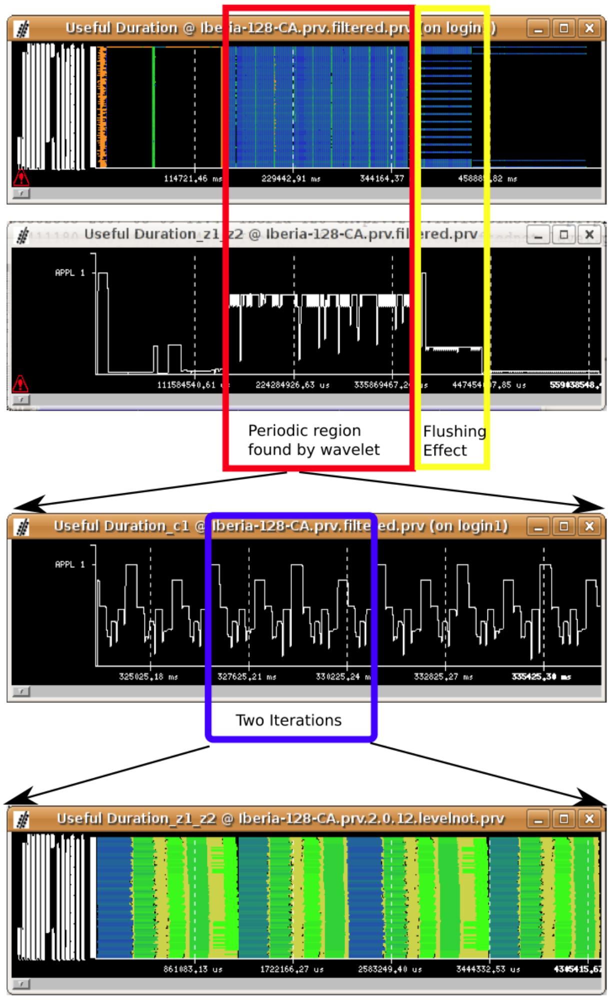
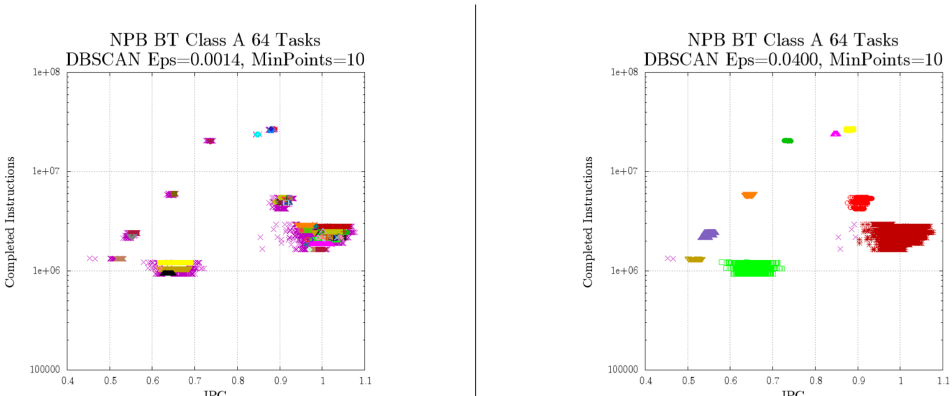

Useful Duration @ Iberia-128-CA.prv.filtered.prv (on login1)
Useful Duration_z1_z2 @ Iberia-128-CA.prv.filtered.prv
Periodic region
found by wavelet
Flushing
Effect
Useful Duration_c1 @ Iberia-128-CA.prv.filtered.prv (on login1)
Two Iterations
Useful Duration_z1_z2 @ Iberia-128-CA.prv.2.0.12.levelnot.prv

Figure 2.10: Iteration detection using wavelet and autocorrelation techniques. The top timeline shows the recorded trace data. The second timeline shows the result of the discrete wavelet transform. The red rectangle marks the iterations, an area exhibiting high frequencies. The third timeline shows the result of the autocorrelation function applied to the iterations area. Marked with the blue rectangle are two iterations. The bottom timeline shows the two iterations in full detail. [27]

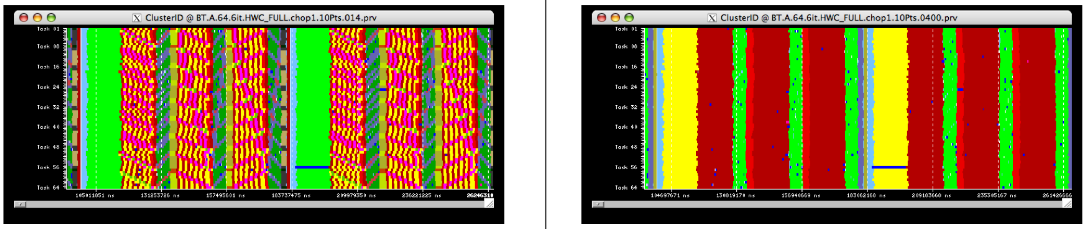
the application is removed in this analysis step. The third step takes the identified iteration phase and analyzes this area using the autocorrelation function. As similar iterations are repeated consecutively, the autocorrelation function efficiently identifies individual iteration occurrences. The results of this work may be used to automatically select a few iterations that represent the application's behavior as good as possible. Afterwards, successive analyses can be focused primarily on these iterations [84]. Additional work extended the described methods by an automatic analysis of the speedup of MPI applications [24].

An approach based on clustering algorithms is proposed by Gonzalez et al. [43]. The goal is to automatically characterize the computation regions of an application. This allows to outline the computation structure of an application's execution, and hence, to contribute valuable information to the performance analysis. The input for the clustering process builds so-called computation bursts. A burst is a region between two communication or synchronization events. The performance properties of each burst are described using hardware performance counters. For instance a combination of the *Instruction Completed* and *Instructions Per Cycle (IPC)* counters may be used as metric to reflect the performance characteristic of one burst. All bursts are clustered using the density-based cluster algorithm DBSCAN [32]. In



(a) Scatter plot of resulting clusters

(c) Scatter plot of resulting clusters


ClusterID @ BT.A.64.6it.HWC_FULL.chop1.10Pts.014.prv
ClusterID @ BT.A.64.6it.HWC_FULL.chop1.10Pts.0400.prv

(b) Paraver time-line

(d) Paraver time-line

Figure 2.11: Detection of the computation structure. Two clusterings of one NPB BT benchmark [8] execution. The DBSCAN algorithm clusters computation regions characterized by the *Instruction Completed* and *Instructions Per Cycle (IPC)* counters. The charts on the left side show the structure in finer detail (more dense clusters required by DBSCAN). The charts on the right side show a coarser computation structure (wider clusters allowed by DBSCAN). [43]

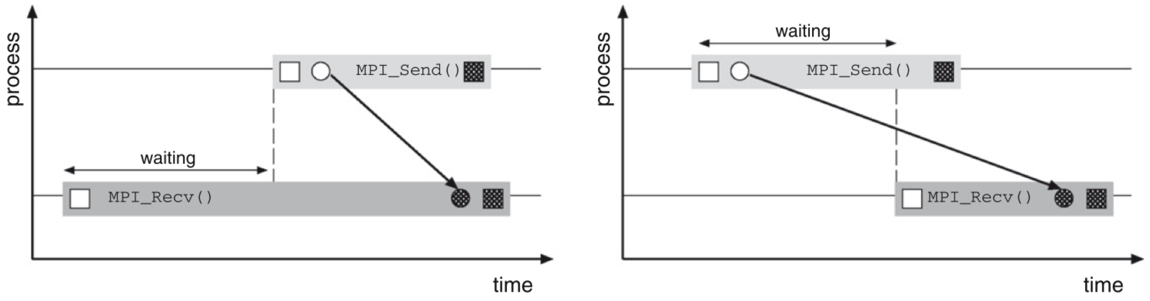
contrast to partitioning cluster algorithms, density-based cluster algorithms are more suitable for this use case as they allow clusters of arbitrary shape. Figure 2.11 depicts two clustering examples.

The detected computation structure (clusters) may be automatically evaluated using a multiple sequence alignment algorithm [44]. The described method evaluates the cluster quality based on the Single Program Multiple Data (SPMD) paradigm. According to the paradigm all processes should execute the same sequence of actions. Thus, a detected cluster should represent a vertical SPMD region. The multiple sequence alignment algorithm takes the sequence of detected bursts of each process as input and aligns them all against each other. The resulting alignment should clearly show a SPMD pattern and allows to determine the *SPMDiness* of the clustering.

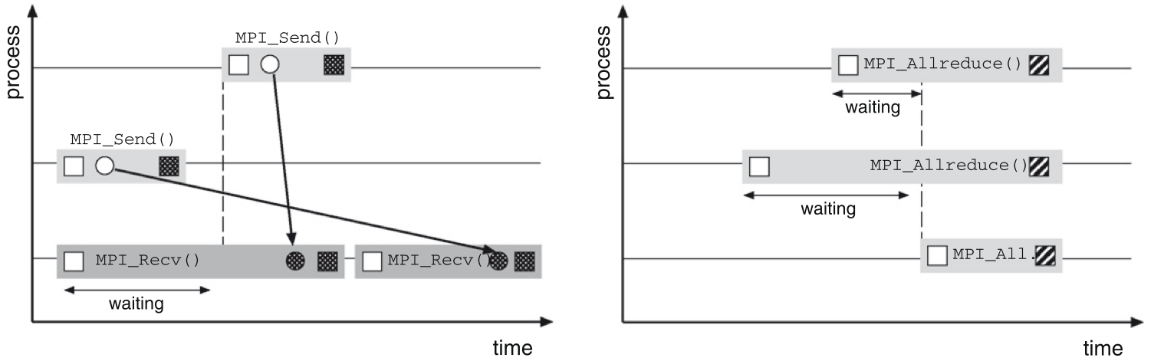
To overcome drawbacks of regular density-based algorithms, an iterative refinement extension of DBSCAN has been implemented [46]. The aggregative cluster refinement algorithm allows detecting clusters with different densities and partly automatizes the algorithm parameterization.

A reported use case of this technique is the extrapolation of performance data [45]. Additionally, the analysis of performance trends using object tracking techniques is described [85]. Therefore multiple application executions with changing conditions are clustered. Changing performance behavior, resulting in moving clusters, is automatically detected and tracked.

Automatic Detection of Communication Patterns Preissl et al. propose a method to automatically detect and exploit patterns in MPI communication traces [113–117]. The goal is to optimize the communication behavior in MPI programs. First, the method finds (local) repeating communication patterns on each process separately. Therefore suffix trees are used to extract arbitrary repeating communication sequences. The maximal (longest) repeat is selected from all detected local repeats for each process respectively. All maximal repeats are searched for specified seed events. Starting with the seed



(a) Late Sender (b) Late Receiver



(c) Late Sender / Wrong Order (d) Wait at $N \times N$

☐ ENTER ▩ EXIT ○ SEND ● RECEIVE ▨ COLLECTIVE EXIT

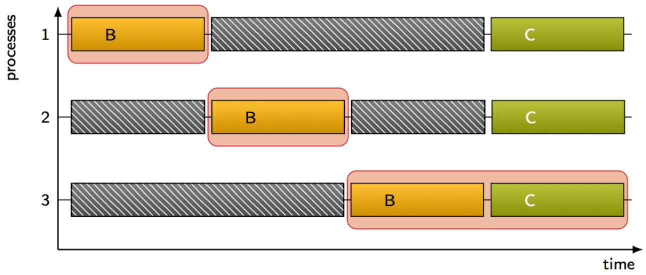
Figure 2.12: A set of patterns of inefficient behavior. In all cases at least one process needs to wait due to an inefficient communication pattern or a previous imbalance. [39]

events the local repeats are grown into global communication patterns spanning multiple processes. A pattern-matching algorithm detects collective communication patterns from global patterns. The detected collective communication patterns build the basis for automatic optimizations implemented by static analysis and source code transformations. An example of applied optimizations is the transformation of point-to-point operations into native MPI collective operations. Also inefficient collective communication can be replaced with better performing point-to-point or native collective operations.

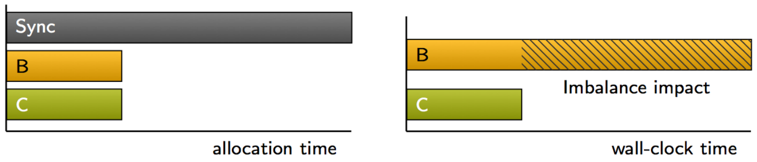
Automatic Imbalance Detection and Analysis In parallel applications load imbalances cause delays of processes. Such delays, even of single processes, may induce wait times across an entire application. Depending on the severity, such wait times can seriously impact the scalability and performance of an application. Performance problems induced by wait states can be formally characterized in terms of execution patterns that represent inefficient behavior. Figure 2.12 illustrates four examples of patterns of inefficient behavior.

The detailed performance information included in traces make them very suitable for detection of wait states. The tool EXPERT [146, 147] automatically parses the trace data for characteristic patterns that indicate performance inefficiencies. The automatic analysis covers the complete trace and quantifies the impact of the detected wait states. The automatic approach saves analysis time and helps to detect all important imbalances in the trace.

EXPERT automatically searches the trace data for occurrences of these predefined patterns. The detected patterns are organized hierarchically, starting from general problems like large communication overhead, to specific problems, like a receiver process waiting for a sender process to start the message transfer. All detected problems are mapped using three interconnected dimensions: problem type, call-



(a) Program timeline. The critical path is marked in red.



(b) Summary profile of the allocation time spent in each activity.

(c) Critical-path profile of the wall-clock time spent in each activity.

Figure 2.13: The critical path profile. The serialized execution of function B is shown in timeline (a). The serialization of B cannot be detected in a usual profile (b). However it is visible as performance bottleneck in the critical path profile (c). [87]

tree position, and process. The analysis results are presented in a single integrated view using the CUBE profiler. The usage of the CUBE output format also enables the comparison of detected performance problems between different application executions. [129].

A new version of the tool, now named *Scalasca*, features a new scalable search algorithm [38, 39, 56]. In contrast to the former serial EXPERT implementation, Scalasca analyses the trace in parallel. The key concept is a parallel replay technique. The application trace for every process is stored in a separate file. The analysis is performed using the same number of processes as have been used for the original application run. Each analysis process traverses the trace of one application process. During the analysis Scalasca replays the target application's communication behavior according to the information stored in the trace data. In the replay Scalasca transfers required information for the diagnosis of inefficient communication. For instance a late-sender pattern, depicted in Figure 2.12, is detected the following way: An analysis process encounters a send operation of a message. It takes the time stamp of the original send operation and sends it to the original receiver process. That way the original communication, a message from sender to receiver process, is replayed. The only difference is the content of the message. Scalasca sends not the original content, but the time stamp of the original send operation. Eventually, the analysis process of the receiving process encounters the receive operation and receives the message. It compares the received time stamp of the send operation with the time stamp of the receive operation. If the sender time stamp is larger than the receiver time stamp, a late-sender pattern is detected. At the end of the parallel replay, the detected performance problems are assembled from all processes and merged into one global report.

The same replay approach may also be used in on-line analysis tools [40]. The tool Periscope detects similar patterns like Scalasca. However, the analysis is not based on trace information but is performed on-line during an application's run. Periscope sends a replayed analysis message directly after each original message. This provides the advantage that no trace data needs to be written. However, the on-line analysis doubles the executed communication of the target application and may induce severe perturbation.

Recent developments extended Scalasca's analysis capabilities by a root cause and a critical path analysis [14, 87]. Processes waiting for a delayed process exhibit the performance problems (wait states) that Scalasca detects. Thus, the wait states are primary symptoms of load imbalance. The root cause analysis [15] tries to determine the delays that cause wait states, i.e., identify processes and individual function invocations responsible for imbalances. In order to detect the responsible delays the analysis replays the traces in parallel both in forward and backward direction. Additionally, the cost of each delay is also computed by addition of all induced wait times.

The critical path analysis identifies the path of execution that determines an application's runtime [13]. Activities on the critical path determine the length of program execution. The optimization of activities not on the critical path only increases wait states and does not improve the runtime of an application. The knowledge of the critical path allows guiding performance optimization more efficiently. One option is to generate a critical path profile, depicted in Figure 2.13. This profile is able to show inefficiencies that otherwise may be hidden through data aggregation. Thus, it exposes crucial imbalances that may be underestimated in usual profiles.

2.4 Limitations of Existing Techniques

The overview of related work reveals that some requirements for a comprehensive comparison of traces are still not covered. Comparing traces implies the investigation of large and complex data volumes. For such a tedious task automated tool support is essential for the analyst. To allow a more detailed discussion of missing functionality, the full comparison process is separated into three major analysis parts.

Visual Analysis The first natural step for users is to visually compare traces. The user manually opens several traces with the tool of his choice and performs visual inspection of the differences. Due to the complexity and large amount of trace data this task may be cumbersome and error-prone. Performing such work, users can greatly benefit from tool support to guide and assist them in the comparison process. Table 2.1 provides an overview of already available assistance.

Essential for a satisfactory comparison workflow is to provide a tool environment with integrated comparison capabilities. This improves the usability for the user and facilitates handling of large performance data sets. Moreover, an integrated comparison environment allows to implement various comparison features. Considering the large amount of events in traces it is essential to support users by automatically highlighting and quantifying differences between traces. Otherwise the analyst might overlook important changes.

From all reviewed tools, only Vampir and the Intel Trace Analyzer provide an integrated environment for visual trace comparison. The compression and analysis techniques are not designed to support visual comparison of complete event streams. The measure management tools primarily focus on the comparison of profiles. Only the Vertical Profiling tool compares event streams and demonstrates visualizations for that purpose. However, this tool focuses on the collection and correlation of traces from exclusively one application configuration. The requirement of certain preconditions for the comparison process inhibits an easy adaption to the comparison of arbitrary traces.

No available tool supports detailed automatic highlighting and quantification of differences between related events.

Structural Analysis Application runs can differ in the sequence of the executed code and in their temporal behavior. The structure of a trace represents the sequence of executed functions as a series of events. An analysis of similarities and changes in the structure is necessary to automatically highlight and quantify differences. Between individual events precisely three different states are possible. Events between traces may be similar; no changes occurred. They may be different; code executed had been changed. Events may also occur only in one trace; code had been added or removed. Table 2.2 provides an overview of structural comparison capabilities of the reviewed techniques.

Limitations of Existing Techniques

	Visual Analysis	
	Support for visual comparison of traces	Highlighting of differences between traces
Visual Event-Trace Comparison		
Vampir	✔	✘
Intel Trace Analyzer	✔	✘
Measurement Management Support		
Differential Profiling	✘	✘
Experiment Management Support	✘	✘
Vertical Profiling	–	✘
Similarity-Based Compression Techniques		
Trace Profiling	✘	✘
Tree-Based Compression	✘	✘
Clustering	✘	✘
Automatic Analysis Techniques		
Structure Detection and Analysis	✘	✘
Communication Pattern Detection	✘	✘
Imbalance Detection and Analysis	✘	✘

Table 2.1: Limitations of existing techniques - Visual analysis.

Tools for visual trace comparison provide no automatic analysis capabilities for structural differences.

Tools for measurement management primarily focus on the comparison of execution timings. The Vertical Profiling approach assumes complete structural identity between traces. Alone the Experiment Management Support tool detects structural differences. However, this tool compares self-generated structural maps. A structural map represents the aggregated structure of a complete application's run in a hierarchical fashion. Hence, it supports the comparison of aggregated structures between application runs. The detection of differentiated differences by comparing complete event streams is out of the scope of this tool though.

All compression algorithms try to exploit similarity. The Trace Profiling and Tree-Based Compression approaches are capable of identifying similar regions between traces. However, a further analysis of differences is not required for this task. The described clustering approaches ignore structural information completely and exclusively group process by temporal information.

The automatic event trace analysis techniques focus on the temporal behavior of application runs. They principally try to detect predefined relevant patterns in the data. One approach analyses the structure of an application's run in terms of initialization, iterations, and finalization. It extracts information from an event stream to automatically categorize the application's execution into phases. A detailed event-wise comparison of event streams is not feasible with this approach.

Limitations of Existing Techniques

	Structural Analysis		
	Detection of equal areas between traces	Detection of changed areas between traces	Detection of new/removed areas between traces
Visual Event-Trace Comparison			
Vampir	✘	✘	✘
Intel Trace Analyzer	✘	✘	✘
Measurement Management Support			
Differential Profiling	✘	✘	✘
Experiment Management Support	–	✘	–
Vertical Profiling	✘	✘	✘
Similarity-Based Compression Techniques			
Trace Profiling	✔	✘	✘
Tree-Based Compression	✔	✘	✘
Clustering	✘	✘	✘
Automatic Analysis Techniques			
Structure Detection and Analysis	✘	✘	✘
Communication Pattern Detection	✘	✘	✘
Imbalance Detection and Analysis	✘	✘	✘

Table 2.2: Limitations of existing techniques - Structural analysis.

Temporal Analysis For performance optimization the analysis and comparison of temporal information between application executions is particularly important. It identifies differences in application performance and allows to detect performance bottlenecks. Likewise, the performance evaluation of changes is usually based on comparison of temporal information. The preservation of all timing information in traces suggests the automatic search for characteristic patterns describing performance bottlenecks. Additionally, the detailed information in traces allows event-wise comparison of runtime differences. This comparison has the potential to expose dynamic runtime changes between application executions. Table 2.3 provides an overview of temporal comparison capabilities of available techniques.

Tools for measurement management support as well as the Intel Trace Analyzer provide various features for the comparison of profiles. Internally the compression techniques also work with comparison of profile information, but do not expose this functionality to the user.

The automatic event trace analysis techniques primarily focus on the analysis of temporal behavior. One approach automatically detects and analyses communication patterns in one trace. The tools for Imbalance Detection and Analysis try to detect predefined patterns that exhibit performance bottlenecks. These tools provide the functionality to compare profile information as well as detected patterns between traces. Dynamic runtime differences captured in detected patterns are also presented to the user. The Structure Detection and Analysis tools automatically categorize an application's execution into phases exhibiting characteristic behavior. They additionally provide automatic techniques for comparison and tracking of detected phases across multiple executions.

Limitations of Existing Techniques

	Temporal Analysis			
	Comparison of profiles	Detection of characteristic patterns in one trace	Comparison of characteristic patterns between traces	Analysis of dynamic runtime differences between traces
Visual Event-Trace Comparison				
Vampir	✘	✘	✘	✘
Intel Trace Analyzer	✔	✘	✘	✘
Measurement Management Support				
Differential Profiling	✔	✘	✘	✘
Experiment Management Support	✔	✘	✘	✘
Vertical Profiling	✔	✘	✘	✘
Similarity-Based Compression Techniques				
Trace Profiling	–	✘	✘	✘
Tree-Based Compression	–	✘	✘	✘
Clustering	–	✘	✘	✘
Automatic Analysis Techniques				
Structure Detection and Analysis	✘	✔	✔	✘
Communication Pattern Detection	✘	✔	✘	✘
Imbalance Detection and Analysis	✔	✔	✔	–

Table 2.3: Limitations of existing techniques - Temporal analysis.

Summary The described related work summarizes existing technology and points out missing functionality for a comprehensive trace comparison. A range of solutions assist the analyst with automatic comparison features for profiles. Also tool support for basic manual comparison of traces is already available. Trace compression techniques detect similarities but focus on one trace. Automatic trace analysis techniques focus on the detection and comparison of specific performance characteristics of applications. Yet, currently no tool is able to detect and visualize detailed structural differences between full event streams. Also no tool covers the analysis and visualization of fine-grained, event-wise runtime differences between application executions.

3 Methods for Structural Comparison of Process Pairs

This chapter describes the development of a method for pairwise comparison of processes[1]. Parallel applications consist of multiple processes. Initially, this chapter only considers comparisons of pairs of processes. Chapter 5 then extends this approach to multiple processes.

The introduced algorithm compares the function call structure of two processes. This approach lays the foundation for subsequent performance comparison techniques described in Chapter 4.

3.1 Jitter in Timestamp Measurements

It is challenging to compare the event streams (traces) of two processes directly. Even if a user tries to exactly reproduce an application run, i.e., running the identical code on the same machine using identical input data, the event streams will differ. The reason is the inherent jitter in timing measurements, caused by a range of effects on the target system, like timer inaccuracy, varying network/system load, or OS noise. Thus, the timings of events are likely to always differ to some extend. For instance when comparing two event streams recorded in two consecutive runs of the same application, the event streams are likely to exhibit a mix of shifted and scaled timing effects. This complicates direct comparison. Figure 3.1 shows possible timing differences that may appear between two event streams.

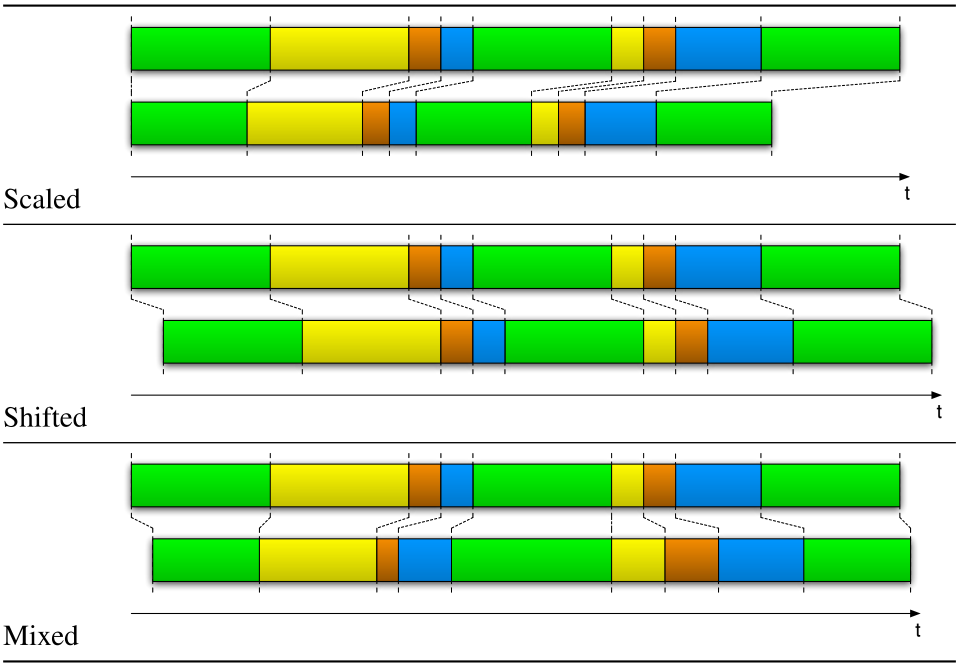


Figure 3.1: Timing differences between two event streams.

Differences between the processes represented by the two streams additionally increase the described timing effects. For instance, if the two processes are executed on different processor types. This might increase the *scaled* effect between the streams, as one process runs faster than the other. Furthermore, if some parts of an application run slower or faster on one process, this may lead to *shifted* timings for

---

[1] In this work, for brevity the term *process* is used to refer to any possible processing element of a parallel application, which may be an MPI process, a thread, a CUDA stream, or a different type of control flow.

the remaining parts of the application. Usually, when comparing event streams of applications, the two streams exhibit a combination, or *mix*, of both effects. Due to these timing effects there is no guarantee that related events between two streams occur visually next to each other. This renders a direct visual comparison of event streams cumbersome, at best. However, it is possible to cope with the timing differences by considering only the structure of the event streams.

3.2 Comparison of the Flat Function Call Structure

An event stream of a process consists of a series of events. These events enable the reconstruction of the application function structure of the process. For deterministic code, this structure is likely to stay constant throughout multiple application runs. This allows to use application function structures as anchors for a comparison. For easier comparison of application structures, flat sequences of executed functions are constructed from application structures. Figure 3.2 shows the construction of function sequences from the application structure. The flat sequence is generated by simply listing functions in the same order as they are executed by the control flow of the application.

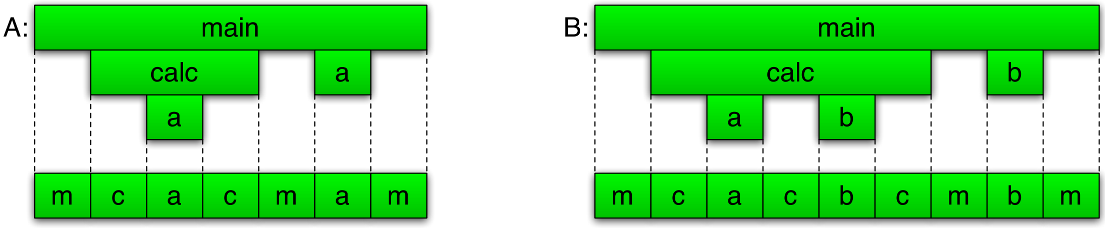


Figure 3.2: Construction of function sequences (bottom) from the application structure (top).

In order to compare different application runs, equal function calls require unique identifiers. A reasonable criteria is to use the function name as a basis for such an identifier. That way, equally named functions get the same identifier in both sequences, and are therefore comparable. During the sequence construction process, timings of events are ignored. Only the order of events is important. If sequences are identical, related events are on equal positions. Such case allows a direct comparison of timings between related events. Figure 3.3 illustrates this process.

In many comparison cases, however, identical sequences cannot be expected. For instance changes to application code can alter the function sequence. When comparing application runs using different input data, after optimizations, or changes of libraries, their sequences are likely to be different. The following section classifies structural differences when comparing non-identical function sequences.

3.2.1 Structural Differences between Pairs of Sequences

The sequence of events can differ between two application runs in a number of ways. Figure 3.4 provides an overview of the possible types of structural differences between two function sequences.

Detecting these differences manually in event streams of real applications is cumbersome. The possibly large size of application event streams requires an automatic comparison method. The next section describes algorithms for automatic sequence comparison.

3.2.2 Sequence Alignment Algorithms

Sequence alignment is a method to measure the similarity of sequences and to describe how sequences are related. It is used in various fields such as spell checking or speech recognition. Probably the best known field using sequence alignment is bioinformatics. In this area it is used (amongst others) for DNA sequence assembly, construction of evolutionary trees, or protein sequence alignment. The difficulty

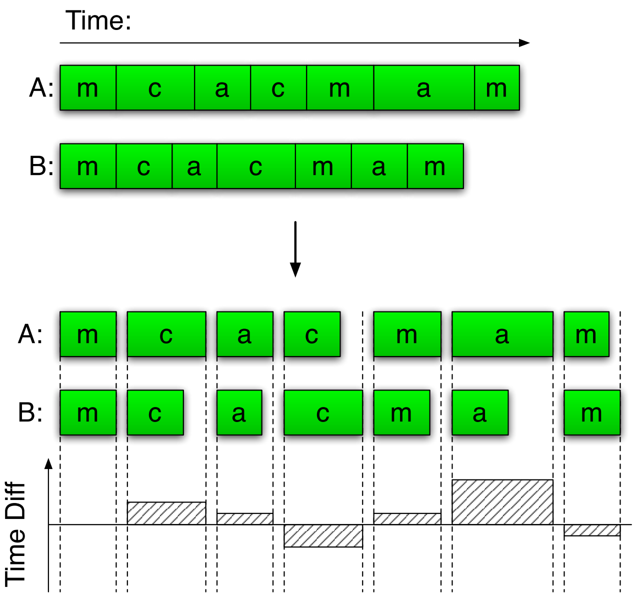


Figure 3.3: Direct comparison of event durations.

when comparing DNA and protein sequences is that genes mutate along the evolutionary process and between species. This forbids search strategies for exact matches. Required are algorithms for *inexact* or *appropriate* string matching. Since this requirement also applies for the comparison of function sequences, algorithms used in bioinformatics are applicable for the structural comparison of processes.

Sequence alignment is one method for approximate matching. It arranges two sequences in a way that allows to identify similar and dissimilar parts between the sequences. During the alignment process, characters of two sequences are matched to each other or may be matched to an empty character "-", called a *gap*. The alignment to a gap indicates an insertion or a deletion. The following example shows alignments of the sequences `m c a c m a m` and `m c a c b c m b m`:

```
m c a c m a m - -        m c a c - - m a m
| | | |     |            | | | |     |   |
m c a c b c m b m        m c a c b c m b m
```

The shown alignments are two examples from a large number of possible alignments of these two sequences. This raises the question for a measure of the quality of an alignment. Levenshtein [82] presented a metric for the evaluation of differences between two sequences. The *Levenshtein distance*, also referred to as *edit distance*, counts the number of edit operations required to transform one sequence into another. It applies a *cost function* assigning fixed costs to each type of operation. Levenshtein assigned a cost of 0 to a match and a cost of 1 to every single edit operation (insertion, deletion, or mismatch). Sometimes a *scoring function* is used instead of a cost function. A scoring function is the inverse of a cost function. A high score implies a good alignment, while a high cost implies a poor alignment.

Depending on the use case, different or more sophisticated cost schemes may be used. For instance protein alignment substitution matrices, e.g., the BLOSUM [54] matrices, which define the cost of the alignment of any two amino acids. In case of event stream alignment the following scoring function is applied. *Equal* areas (matches) having the same sequence of function calls in both processes achieve a score of 2. The performance characteristics of equal areas can be compared directly. Areas containing different function calls at the same sequence position are labeled as *different*, evaluated with a negative score of -1. For instance, if a call to function *a* in the first sequence is replaced by a call to function *b* in the second sequence, these calls would be recognized as different. For performance analysis, the

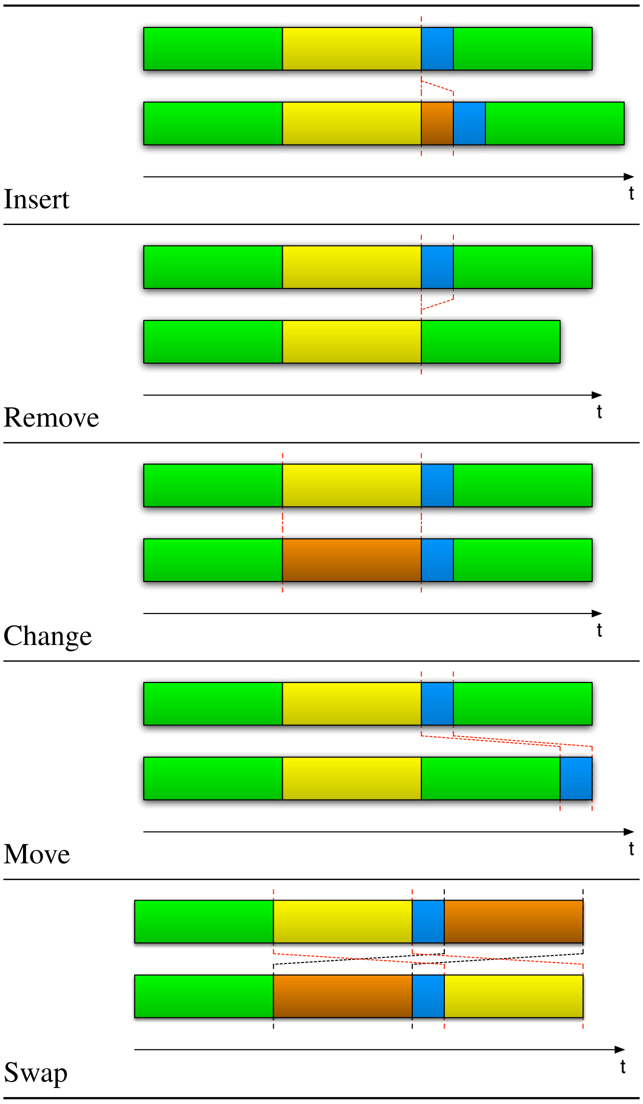


Figure 3.4: Types of changes between two function sequences.

identification of these areas is important, as the application is executing different code. The third possibility is a missing section, called a *gap*, in one sequence, evaluated with a negative score of -1. This happens if some functions are executed during the generation of only one of the event streams, e.g., if a new code section is added to the application, or a process follows a different execution path. Gaps enable the analyst to quickly identify code areas that are only present in one of the event streams.

Finding Optimal Alignments After defining an evaluation metric, the next task is to find the optimal alignment, i.e., the alignment exhibiting the highest score/minimal cost. To solve this problem using the brute-force approach is infeasible. The number of possible alignments for two sequences with the length N can be approximated with the equation $2^{2N}/\sqrt{2\pi N}$ [31]. For two sequences of length 300 there are already about 10^{179} different alignments and sequences constructed from event traces could be considerably larger.

The best known algorithm for solving this optimization problem and finding the optimal alignment is dynamic programming [10, 49]. The dynamic programming algorithm separates the full pairwise alignment problem into independently optimizable sub-problems. The algorithm starts with solving the smallest sub-problems and then iteratively combines the found solutions to solve the next bigger sub-problems. To prevent a compute intensive recalculation of already solved sub-problems, found solutions are stored in a dynamic programming matrix.

The Needleman–Wunsch algorithm [107] was the first application of the dynamic programming algorithm for biological sequence comparison. The purpose of the algorithm is to search for similarities in amino acid sequences of two proteins. A variation of the Needleman–Wunsch algorithm was published by Smith and Waterman [128]. The Smith-Waterman algorithm is designed to compute the local alignment of two sequences. Typically used to search for places in a long sequence that exhibit high similarity to a short (query) sequence.

This work adapts the Needleman–Wunsch algorithm for the comparison of application event streams. To explain the algorithm with an example, the following two event sequences are compared:

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} event in A is A_i and the j^{th} event in B is B_j.

For the comparison of trace files the following scores are applied:

$$\begin{aligned} \sigma_{i,j} &= 2 && \text{Match Score} \\ \sigma_{i,j} &= -1 && \text{Mismatch Score} \\ \sigma_{gap} &= -1 && \text{Gap Score} \end{aligned} \tag{3.1}$$

$\sigma_{i,j} = 2$ is chosen if the event 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 $\sigma_{gap} = -1$ is applied.

Based on the scores, the algorithm defines the following recursive scoring scheme for a dynamic programming matrix D:

$$D_{i,j} = \max \begin{cases} D_{i-1,j-1} + \sigma_{i,j}, & \text{Match/Mismatch} \\ D_{i,j-1} + \sigma_{gap}, & \text{Gap in Sequence } A \\ D_{i-1,j} + \sigma_{gap}. & \text{Gap in Sequence } B \end{cases} \tag{3.2}$$

The dynamic programming matrix D keeps track of already solved sub-problems.

The optimal alignment is found by computing the path with the highest score through the matrix D from $D_{M,N}$ to $D_{0,0}$. The matrix is initialized at the top left corner with $D_{0,0} = 0$. The entry $D_{0,0}$ gives the starting score of the alignment. By applying the recursive scoring scheme the matrix is filled from the top left to the bottom right corner. Each matrix field is filled with the highest scoring option possible based on the scoring scheme. Figure 3.5 illustrates the computation step for one matrix field. The bottom right entry $D_{M,N}$ then holds the optimal score for the complete alignment of both sequences. Figure 3.6 shows the completely filled dynamic programming matrix for this example.

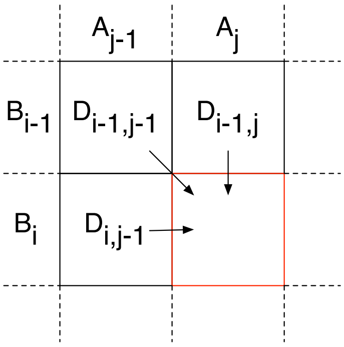
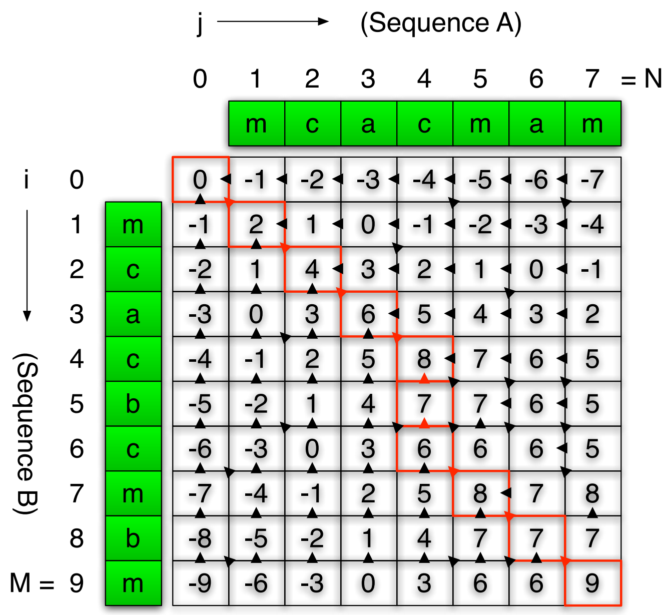


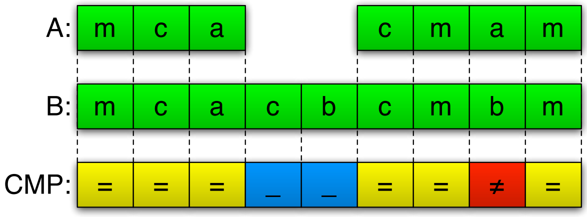
Figure 3.5: Dynamic programming matrix calculation step.



i		0	1	2	3	4	5	6	7 = N
			m	c	a	c	m	a	m
0		0	-1	-2	-3	-4	-5	-6	-7
1	m	-1	2	1	0	-1	-2	-3	-4
2	c	-2	1	4	3	2	1	0	-1
3	a	-3	0	3	6	5	4	3	2
4	c	-4	-1	2	5	8	7	6	5
5	b	-5	-2	1	4	7	7	6	5
6	c	-6	-3	0	3	6	6	6	5
7	m	-7	-4	-1	2	5	8	7	8
8	b	-8	-5	-2	1	4	7	7	7
M = 9	m	-9	-6	-3	0	3	6	6	9

Figure 3.6: Dynamic programming matrix for sequences A and B.

Furthermore, it is practical to remember what choice led to the optimal score for each field, as this is useful for the construction of the alignment. In Figure 3.6 this is indicated with arrows between the matrix fields. This information is used to backtrack through the matrix and thereby to find the optimal sequence alignment. Starting point is the bottom right corner, which represents the optimal score. Then the path is followed along the arrows to the top left corner. In case of multiple possible paths from one field to the next, the field with the highest score is chosen. In the example shown in Figure 3.6, the backtracking step is indicated by the red path, which 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. The alignment result for the example is shown in Figure 3.7. The *CMP* bar in the figure marks equal, different, or gap areas.



A:	m	c	a			c	m	a	m
B:	m	c	a	c	b	c	m	b	m
CMP:	=	=	=	_	_	=	=	≠	=

Figure 3.7: Constructed optimal alignment of sequence A and B.

The basic dynamic programming algorithm has quadratic complexity with respect to the sequence lengths, $\mathcal{O}(M \cdot N)$, regarding time and memory requirements.

Computing the Alignment with Linear Space Requirements Sequences constructed from application event streams may have lengths of millions of elements. This is especially problematic considering the quadratic memory complexity of the dynamic programming algorithm. A solution to this problem is a modification of the Needleman–Wunsch algorithm proposed by Hirschberg [59] that reduces the quadratic memory complexity, $\mathcal{O}(M \cdot N)$, to only linear memory complexity with respect to the longest sequence, $\mathcal{O}(max(M, N))$.

This modification is based on the observations that the dynamic programming matrix is solved one row at a time and that for the calculation of a row only the previous row of the matrix is required. The idea of the algorithm is to avoid the need to save the entire matrix by skipping the backtracking step. Instead, matrix fields lying on the optimal alignment path are computed directly using a divide and conquer approach. Figure 3.8 illustrates the scheme of the algorithm.

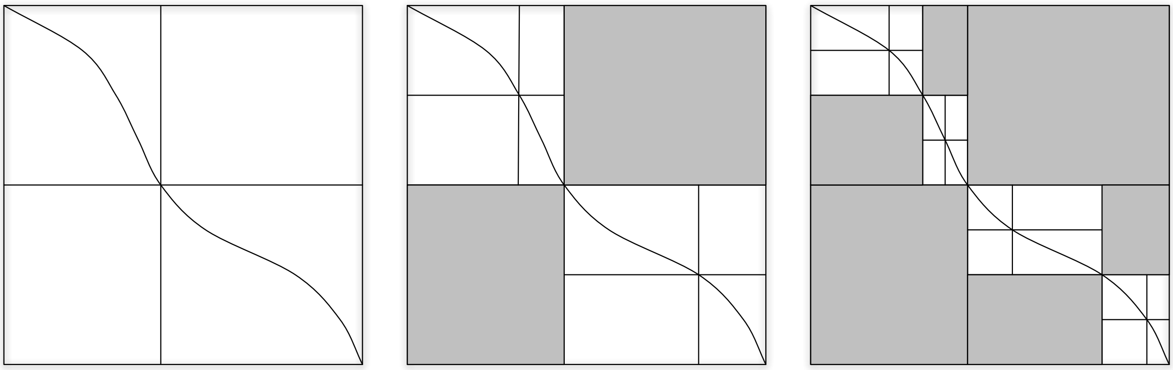
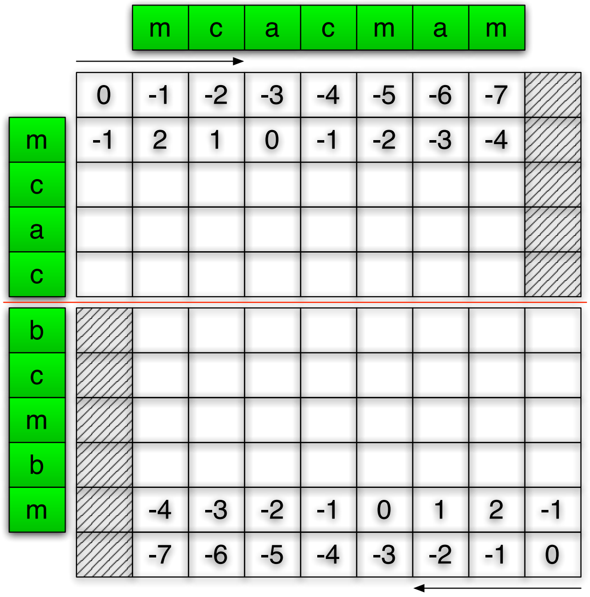


Figure 3.8: Hirschberg algorithm scheme.

First, the matrix is separated in the middle. The top half of the matrix is solved using the basic dynamic programming algorithm. The bottom half is solved backwards using a reverse version of the dynamic programming algorithm. Figure 3.9 depicts this procedure for the example alignment. Computed rows are only kept in memory as long as they are required for the computation of the next row.



		m	c	a	c	m	a	m	
	0	-1	-2	-3	-4	-5	-6	-7	▨
m	-1	2	1	0	-1	-2	-3	-4	▨
c									▨
a									▨
c									▨
b	▨								
c	▨								
m	▨								
b	▨								
m	▨	-4	-3	-2	-1	0	1	2	-1
	▨	-7	-6	-5	-4	-3	-2	-1	0

Figure 3.9: First step of the Hirschberg algorithm. The dynamic programming matrix is split horizontally. The top half of the matrix is solved forward using the basic dynamic programming algorithm. The bottom half is solved backwards using a reverse version of the dynamic programming algorithm. For the computation of one row only the previous row needs to be saved.

After the computation of the two center rows around the separation line, the middle split point is defined. This is the point yielding the highest value after addition of neighboring fields of the two center lines. Figure 3.10 depicts the calculation of the first split point for the example alignment.

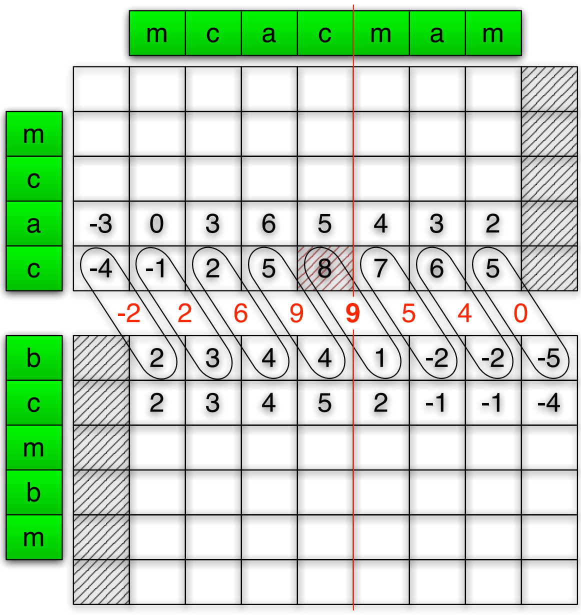


Figure 3.10: Second step of the Hirschberg algorithm. Calculation of the vertical split line. Related pairs of the two rows around the horizontal separation line are added together. The pair with the highest score lies on the optimal path and defines the separation line. In this example the pair $c - c$ is aligned and the matrix is split accordingly.

The split point is used to partition the complete matrix into four smaller matrices. Only the top left and the bottom right matrices need to be considered further. The other, top right and bottom left matrices can be discarded, see Figure 3.11.

According to the recursive scheme, see Figure 3.8, the algorithm is started again in each relevant matrix. This process is repeated until all fields on the optimal alignment path are calculated.

The improvement of the space complexity comes at the price of higher computational cost, compared to the Needleman–Wunsch algorithm. Nevertheless, the Hirschberg algorithm still operates in quadratic time complexity.

Improved Algorithm for Finding the Optimal Alignment Ukkonen [133] and Myers [105] independently developed an improved version of the basic dynamic programming algorithm. The improved algorithm, called *O(ND) Algorithm* by Myers, is used in the UNIX *diff* tool to compare text in files [93]. For clarity, in this document, the improved algorithm is referred to as *Ukkonen's algorithm*.

Ukkonen's algorithm makes the following restrictions to the cost functions. The match cost needs to be 0. All other costs need to be the same and small positive integers. Usually, the algorithm is run using the Levenshtein [82] cost model, i.e., a match costs 0, all other changes cost 1. Compared to the scoring scheme described above, the Levenshtein cost model yields the same alignment.

Ukkonen's algorithm speeds up the basic dynamic programming algorithm by exploiting a couple of observations from the dynamic programming matrix. First, not all entries of the matrix are needed to compute an alignment. Second, when using Levenshtein costs, values on diagonals of the dynamic programming matrix (running from the upper-left to the lower-right cells) are monotonically increasing. Third, only endpoints of running matches are important. Figure 3.12 depicts the full dynamic programming matrix for an example alignment using Levenshtein costs. The matrix diagonals are shown as dashed lines. The end of running matches on diagonals are indicated by red boxes.

To reduce the number of computed matrix cells Ukkonen defines a cutoff rule. All alignments start in the top left corner, diagonal 0, of the dynamic programming matrix. Eventually, they will end at the bottom right corner, diagonal $|M - N|$, of the matrix. Since values in the matrix are monotonically

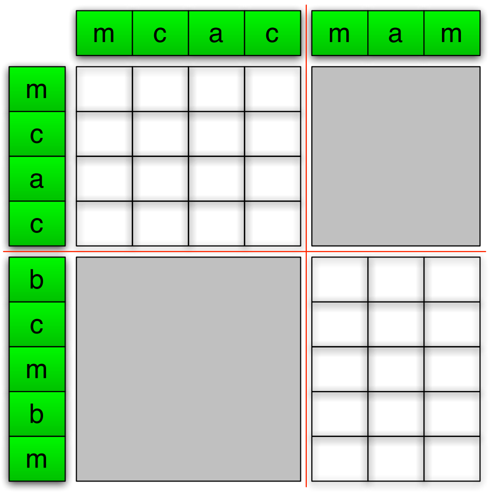


Figure 3.11: Third step of the Hirschberg algorithm. The horizontal and vertical split lines are used to define the new matrices for the next step. The Hirschberg algorithm is now started recursively with the top left and bottom right matrix.

increasing, it is possible to exclude matrix areas that cannot belong to the optimal alignment. Depending on the edit distance $D_{M,N}$ the optimal alignment stays in a band from the top left to the bottom right corner. The smaller the edit distance, the narrower the possible band for the optimal alignment. Alignments that leave the band can only reach the bottom right corner at a higher cost than $D_{M,N}$.

Figure 3.13 depicts the cutoff scheme for $M \leq N$. All alignments start and end in the band between diagonal 0 and diagonal $M - N$. This area is enclosed by the dashed lines in Figure 3.13. Depending on the edit distance $D_{M,N}$ the band enclosing the optimal alignment may be broadened at the sides by additional diagonals. For a given edit distance $D_{M,N}$ the number of additional diagonals p can be calculated with the equation $p = \lfloor \frac{1}{2}(D_{M,N}/\delta - |M - N|) \rfloor$, where δ denotes the cost for changes.

The cutoff band for the example alignment of sequences A and B is illustrated in Figure 3.12. The edit distance of the alignment is $D_{M,N} = 3$. For this example no additional diagonals $p = 0$ are required. Consequently, the optimal alignment is guaranteed to stay within the minimal band between diagonals 0 and 2. All alignments leaving the band will reach the bottom right corner $D_{M,N}$ at a higher cost than $D_{M,N} = 3$. Thus, matrix entries outside the band can be discarded, illustrated as transparent rectangles in Figure 3.12.

Using the cutoff rule, a more efficient way to calculate an optimal alignment can be implemented. Of course, in the beginning the edit distance of the optimal alignment is unknown. However, it is known that the optimal alignment must reach the bottom right corner. Using that knowledge, Ukkonen's algorithm starts the alignment process at the top left corner with an edit distance of 1. Then it evaluates if an alignment with an edit distance 1 reaches to the bottom right corner. If no alignment has been found, the algorithm iteratively increases the edit distance until the bottom right corner is reached. The first alignment reaching the bottom right corner is an optimal alignment. Since the algorithm only computes matrix fields eligible up to the edit distance of the optimal alignment, it saves the computation of unnecessary fields.

In combination with the cutoff rule, Ukkonen's algorithm uses additional properties of the dynamic programming matrix to speed up the computation. In particular, the algorithm works on the diagonals of the dynamic programming matrix. Since the values on a diagonal are monotonically increasing, only the endpoints of running matches are important. If portions of two sequences are similar, the corresponding values on the related diagonal will stay the same, see Figure 3.12. The exploitation of these portions of running matches gives the algorithm its primary speedup.

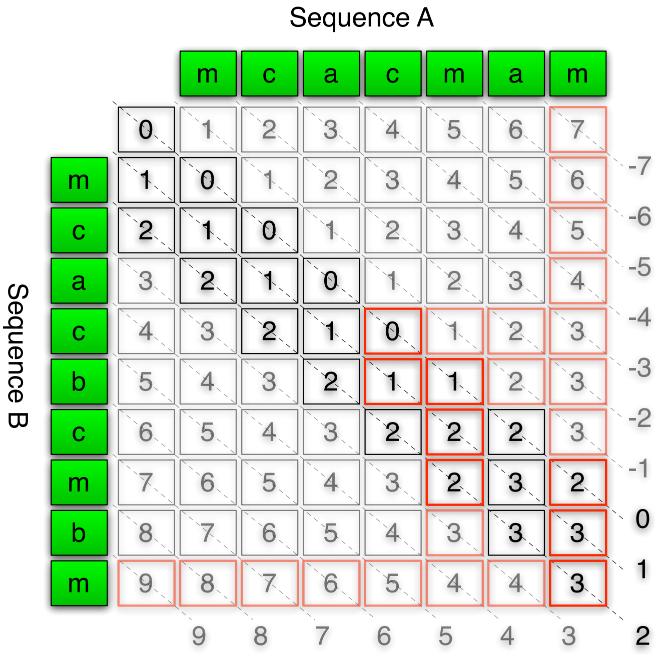


Figure 3.12: Dynamic programming matrix using Levenshtein costs. Diagonals are indicated by dashed lines. Ends of running matches on diagonals are indicated by red boxes. Unnecessary matrix fields inside the cutoff area are depicted as semitransparent boxes.

The pseudo-code for Ukkonen's algorithm is given in Algorithm 1. The key part of the algorithm is the recursive function *ukkonen* that evaluates how far down a diagonal reaches for a given cost. The optimal alignment will end at the bottom right corner of the dynamic programming matrix D. Thus, the algorithm starts with the *final diagonal*, i.e. the diagonal that goes through the bottom right corner, and has an edit cost of 0. Then it iteratively increases the edit cost until the final diagonal reaches down to the corner, with $editCost < |B|$.

The computation of the distance of a diagonal is done by reducing the edit cost by one, $editCost - 1$, and calculating the distances achieved by the same diagonal and the two neighboring diagonals. From the three options the maximum distance is chosen and extended while a set of running matches exists on the chosen diagonal. To store computed results efficiently, the algorithm operates on an alternative U matrix, shown in Figure 3.14. The U matrix saves how far diagonals reach down, counted in positions

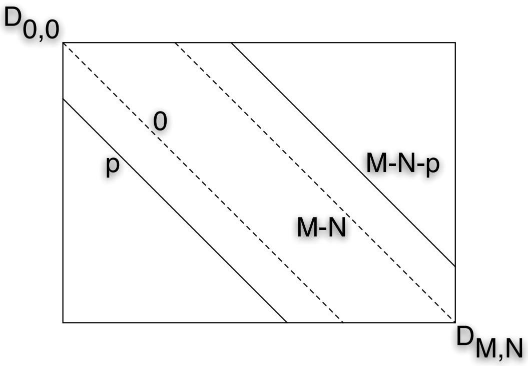


Figure 3.13: Ukkonen cutoff scheme for $M \leq N$. The optimal alignment is guaranteed to stay within the band from p to $M - N - p$. Fields outside this area can be discarded in the computation.

Algorithm 1: Ukkonen's algorithm

```
Input: Sequences $A$ and $B$.
Output: Edit cost $editCost$.
$U[0,0]$ // U matrix
Function editDistance($A$, $B$)
    $editCost = 0$
    // Increase edit distance until diagonal reaches end of B
    while ukkonen($|A| - |B|$, $editCost$) < $|B|$
        $editCost{+}{+}$
    return $editCost$
Function ukkonen($diagonal$, $cost$)
    // Starting condition
    if $|diagonal| == 0$ and $cost == -1$
        return $-1$
    // Cutoff rule
    if $|diagonal| > cost$
        return $-\infty$
    // Entry already computed?
    if alreadyComputed( $U[diagonal, cost]$)
        return $U[diagonal, cost]$
    // Compute distance
    $distance$ = max( ukkonen($diagonal + 1$, $cost - 1$),
                      ukkonen($diagonal$,     $cost - 1$) +1,
                      ukkonen($diagonal - 1$, $cost - 1$) +1)
    // Extend diagonal for a run of matches
    while $A[distance + 1] == B[distance + 1]$
        $distance{+}{+}$
    // Save and return distance for diagonal
    $U[diagonal, cost] = distance$
    return $distance$
```

of sequence B, for a given edit cost. The vertical axis denotes the diagonals, the horizontal axis denotes the edit cost. The U matrix can also be translated into the dynamic programming matrix D. Rows in the U matrix correspond to diagonals in the D matrix. Columns in the U matrix correspond to contours of related costs in the D matrix. The U matrix for the example alignment is shown in Figure 3.14.

In terms of the D matrix, Ukkonen's algorithm only computes matrix fields relevant for the optimal alignment. Fields affected by the cutoff scheme are discarded. Additionally, the algorithm exploits similarity in the form of running matches. Diagonals alongside running matches do not need to be computed. Figure 3.15 depicts the calculated fields in the D matrix for the example alignment.

Ukkonen's algorithm runs in $\mathcal{O}(nd)$ worst-case time complexity, where n is the accumulated length of the sequences A and B, and d is the edit cost. The actual runtime of the algorithm is output dependent. The speedup methods used in Ukkonen's algorithm benefit most from the similarity between sequences. A smaller edit cost d directly translates into a faster runtime of the algorithm. The memory complexity of the algorithm is quadratic with respect to the edit distance $\mathcal{O}(d^2)$. By combining Ukkonen's algorithm with Hirschberg's divide and conquer approach the memory requirement can be reduced to linear complexity $\mathcal{O}(d)$.

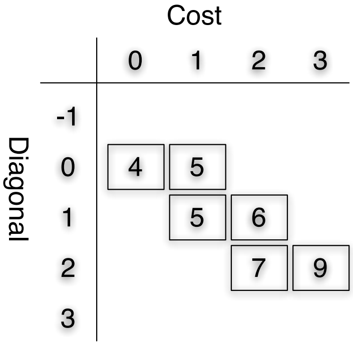


Figure 3.14: The U matrix, used by Ukkonen's algorithm, for the example alignment of sequences `mcacmam` and `mcacbcmbm`.

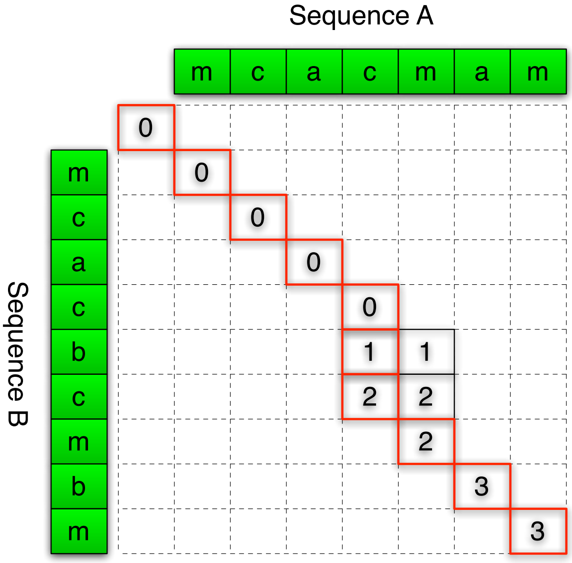


Figure 3.15: Dynamic programming matrix showing the fields calculated by Ukkonen's algorithm for the example alignment. Red boxes depict the path of the optimal alignment.

3.2.3 Speedup of Special Alignment Cases

Dynamic programming algorithms always compute an alignment using the complete input sequences. However, some special cases between sequences do not require the computation of an alignment. Thus, a prior step detecting and handling special cases between two sequences may provide significant speedup. Directly processing the following cases requires only linear complexity, except the check for a common prefix or suffix, which can be computed in logarithmic complexity. Consequently, the handling of these cases avoids their costly computation using algorithms with quadratic complexity.

The following checks of two sequences A and B may be performed prior to using sequence alignment algorithms:

- **Check if A equals B.** If both sequences are similar, no alignment needs to be computed. Detection of this case completely avoids the use of dynamic programming algorithms.
- **Trim off common prefix and suffix.** After the trimming only the remaining middle portion of A and B needs to be aligned. If the middle portion is empty in one sequence, no alignment needs to be computed. The empty positions can be directly labeled as gaps.

- **Check if one sequence (shorter one) is contained in the other (longer one).** If one sequence fits into the other, no alignment is required. The short sequence is directly aligned to the matching part of the longer sequence. The remaining unmatched positions in the longer sequence are aligned to gaps.

All described cases do not change the resulting optimal alignment, but significantly increase its computation time.

3.2.4 Fast Alignment Heuristic

The quadratic time complexity of dynamic programming algorithms renders their usage for the alignment of very long sequences infeasible. When all speedup techniques don't reduce the sequence lengths to a reasonable size that allows the computation of an alignment within a few seconds, a last resort measure for the alignment is required. This section describes such an algorithm applicable to very long sequences. The algorithm is a heuristic designed to quickly compute an alignment even of very long sequences. However, the resulting alignment is not guaranteed to be mathematical optimal anymore.

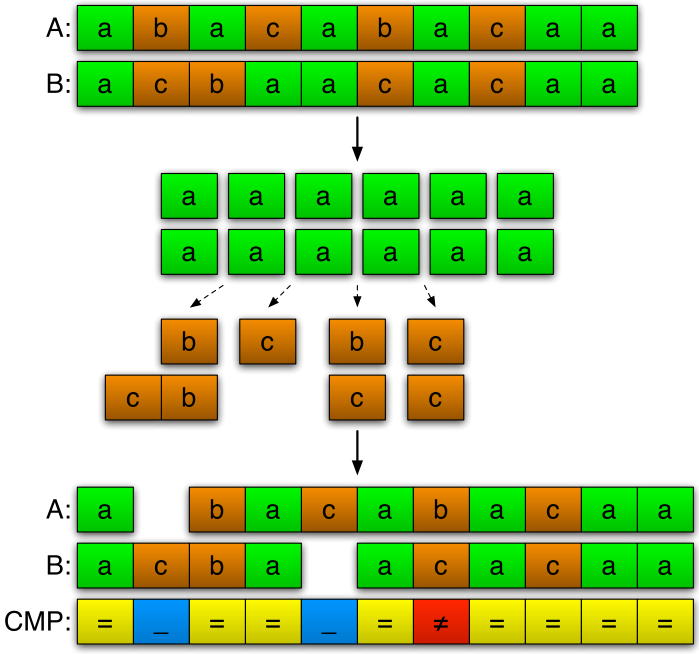


Figure 3.16: Fast alignment heuristic scheme.

Figure 3.16 depicts the algorithm scheme. The heuristic selects the most frequent function of both sequences as anchors and directly aligns all occurrences. If one sequence has more occurrences of this most frequent function than the other, the remaining unaligned occurrences stay simply appended at the end of the sequence and are not used as anchors. The remaining functions between the anchors build new sub-sequences for further alignment. If resulting sub-sequences should still be too long for dynamic programming algorithms, the heuristic is recursively applied again.

3.2.5 Accuracy of Detection of Structural Differences

All above described algorithms, with exception of the fast alignment heuristic, compute a mathematical optimal alignment. However, there may be multiple optimal solutions for an alignment of two sequences. This introduces some ambiguity to an alignment. For instance when aligning the two sequences `a a b a a b a a` and `a a`. Shown next are three alignments that represent optimal solutions to this problem. The three alignments differ, but all are mathematically correct.

```
a a b a a b a a     a a b a a b a a     a a b a a b a a
| |                       | |           |             |
a a - - - - - -     - - - a a - - -     a - - - - - - a
```

Usually a user expects to see only one certain solution. However, this solution directly depends on the changes the user performed on the code. In this example, the user started with the longer sequence `a a b a a b a a` and changed code until only the sequence `a a` remained. Depending on the actual changes, only one solution appears as correct to the user. For instance, if the user removed all calls in the code except the first two calls to `a`, the first alignment shown above would be the correct solution. However, if the user edited the code in a different way, another alignment would be correct.

The introduced algorithms cannot solve this ambiguity, as the actual code changes are not detectable in the available input data. Function sequences do not provide an additional relation other than the function name. Without further knowledge, it is not possible to decide which two `a`-calls in the longer sequence represent the two `a`-calls in the shorter sequence in the above example. Depending on the implementation, all alignment algorithms select one optimal solution. Whether that chosen solution reflects the user expectation cannot be evaluated with the available input data.

3.3 Accelerating the Alignment by Exploiting Hierarchy in Function Call Structures

Although the dynamic programming approach for the comparison of flat sequences satisfies the functional requirement, the quadratic time complexity means that the alignment of large traces cannot be done in an acceptable amount of time. To align and compare long application runs, their function sequence lengths need to be reduced. A truncation of the sequences is not an option if the complete application run should be considered. It is however possible to split up a sequence into several smaller parts. If the length of the smaller parts is short enough, the alignment becomes feasible again. The fast alignment heuristic, introduced in Section 3.2.4, has the potential of separating the complete application sequence into smaller parts. Yet, this heuristic is designed as a last resort measure and not for intelligently splitting complete application sequences. The challenge is to split up the complete sequence in a reasonable way. In that regard, the function structure of application executions provides an advantage over the flat sequences occurring in bioinformatics. Most applications do not consist of flat function call sequences. Functions are rather called in a hierarchical fashion and their call structure can be represented in a call tree. This call tree structure provides a powerful tool for splitting up the flat sequence. Moreover, the consideration of the already available call tree structure allows to design optimized algorithms for program comparison, since it naturally represents the program structure. This section contributes an algorithm that augments the dynamic programming method with a hierarchical comparison approach based on the call tree structure of an application.

3.3.1 Hierarchical Alignment Algorithm

The proposed hierarchical approach uses the call tree structure to split a flat function call sequence into multiple sub-sequences. Therefore, the hierarchical algorithm exploits the natural call structure by only aligning direct sub-functions of related functions in the call tree. For that purpose the algorithm traverses two call trees in parallel. While traversing the call tree, sub-sequences are built for sub-function calls (child-nodes) of each function (parent node), as depicted in Figure 3.17. For the comparison only sub-alignments of the sub-sequences are necessary, as shown in Figure 3.18. The complete alignment is then constructed from the smaller sub-alignments. The result of the hierarchical alignment for the example processes is depicted in Figure 3.17.

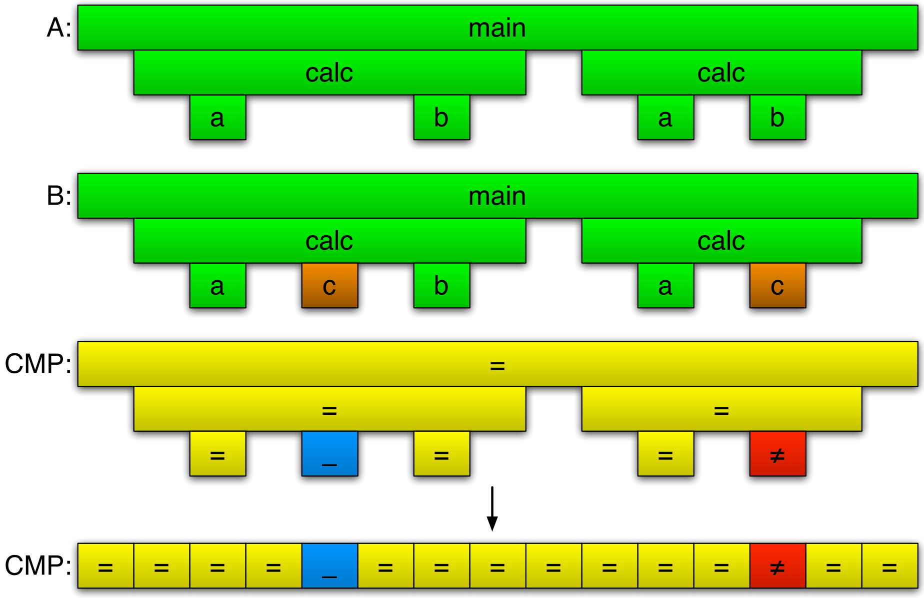


Figure 3.17: Hierarchical alignment scheme. The complete sequence is split up into multiple sub-sequences according to the call tree structure. In multiple sub-alignment steps only related sub-sequence pairs are aligned. Finally, the complete alignment is constructed from the sub-alignment results.

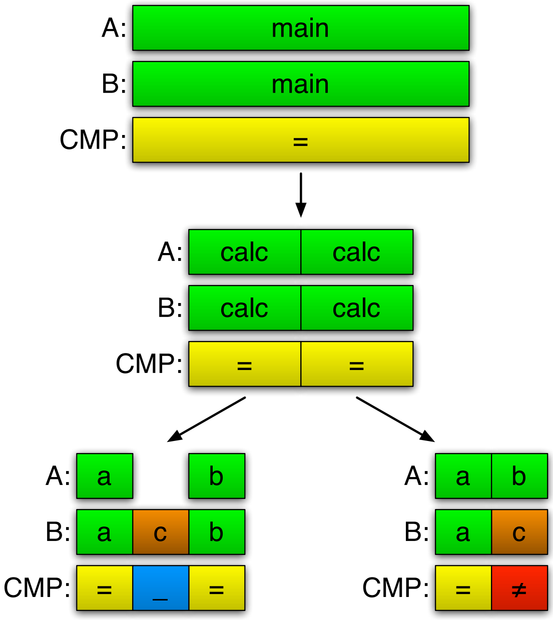


Figure 3.18: Individual alignment steps. Only sub-sequences built from direct child-nodes of the two compared parent nodes are aligned. Individual alignments are significantly shorter than the complete function call sequence.

Algorithm 2: Hierarchical Alignment Algorithm.

```
Input: Processes $A$ and $B$.
Output: Flat $alignment$ of input processes.
$alignedCallTree = 0$ // Call tree holding hierarchical alignment (HA)
Function computeHierarchicalAlignment(A, B)
    // Get call trees from processes
    $callTreeA$ = getCallTree($A$)
    $callTreeB$ = getCallTree($B$)
    // Start hierarchical alignment with root nodes
    compareNodes(getRootNode($callTreeA$), getRootNode($callTreeB$))
    // Traverse HA and construct flat alignment
    $alignment$ = constructFlatAlignment($alignedCallTree$)
    // Return flat alignment
    return alignment
Function compareNodes(nodeA, nodeB)
    // Compare functions of input nodes
    $state$ = compareFunctions($nodeA$, $nodeB$)
    // Add new node with result state (Equal, Diff, Gap) to HA
    addNewNode($alignedCallTree$, $state$)

    // If both nodes have no child-nodes the comparison stops here
    if false = hasChildNodes($nodeA$) $\wedge$ false = hasChildNodes($nodeB$)
        return // Both nodes are leaf nodes, done

    // If one node is empty, add child-nodes of other node
    if $state$ = Gap
        foreach $childNode$ in selectNonGapNode($nodeA$, $nodeB$)
            // Add child-nodes of non-gap node to HA
            compareNodes($childNode$, 0)
        return // done

    // Compare child-nodes of input nodes

    // Generate sequences from child-node calls of both nodes
    $subSequenceA$ = genChildNodeSequence(nodeA)
    $subSequenceB$ = genChildNodeSequence(nodeB)
    // Compute sub-alignment of sequences
    $subAlignment$ = calculateAlignment($subSequenceA$, $subSequenceB$)
    // Iterate over sub-alignment
    foreach $childNodePair$ in $subAlignment$
        // Get aligned child-nodes from sub-alignment
        $childNodeA$, $childNodeB$ = getChildNode($childNodePair$)
        // Continue HA with child-nodes
        compareNodes($childNodeA$, $childNodeB$)
    return // done
```

The pseudo-code given in Algorithm 2 describes the hierarchical alignment algorithm in detail. The algorithm uses the call trees of the two input processes for the comparison. During the traversal of the two call trees, the algorithm constructs an alignment call tree containing the alignment information. The algorithm starts with the root nodes of the two call trees as their start nodes. Each comparison step first compares the current parent nodes and adds the result to the alignment call tree. Then, it constructs sub-sequences from the child-nodes of the parent nodes and aligns these sub-sequences. Subsequently, it iterates over the sub-alignment and starts new comparison steps with aligned child-nodes. This way, the algorithm traverses both input call trees. After complete traversal, the alignment call tree contains the alignment information for both input call trees. This information is sufficient for the comparison of both processes. Optionally, it is possible to construct a flat alignment from the alignment call tree.

3.3.2 Scalability Considerations

The hierarchical comparison approach shortens the event sequence length for individual comparisons by leveraging the call tree structure. As the runtimes of the presented dynamic programming algorithms increase quadratically with sequences lengths, a shortening of the sequence lengths will result in a notable runtime reduction.

The steps required for computing a hierarchical alignment of two processes are the parallel traversal of the call trees and the computation of all sub-alignments of the sub-sequences. The traversal of the call trees can be computed in linear time complexity. The alignments of the sub-sequences are computed using dynamic programming algorithms. Formally, the complexity for comparing two processes is $\mathcal{O}(g + h \cdot n^2)$, with two input call trees containing g elements, requiring h sub-alignments with sub-sequences of maximum length n. Since the quadratic component dominates, the worst-case time complexity of the hierarchical alignment algorithm is still quadratic, $\mathcal{O}(n^2)$.

Yet, before each sub-alignment step, the lengths of the sub-sequences are known beforehand. This enables an estimation of the time required to compute an individual sub-alignment. If the expected alignment time exceeds a set limit, the algorithm can select the fast-alignment-heuristic method for computing the sub-alignment. This allows to control the impact of the quadratic component in the algorithm. Thus, the expected time complexity for computing a hierarchical alignment of two processes is linear, $\mathcal{O}(g)$. The benchmark depicted in Figure 3.19 supports this assumption, by measuring how varying sub-

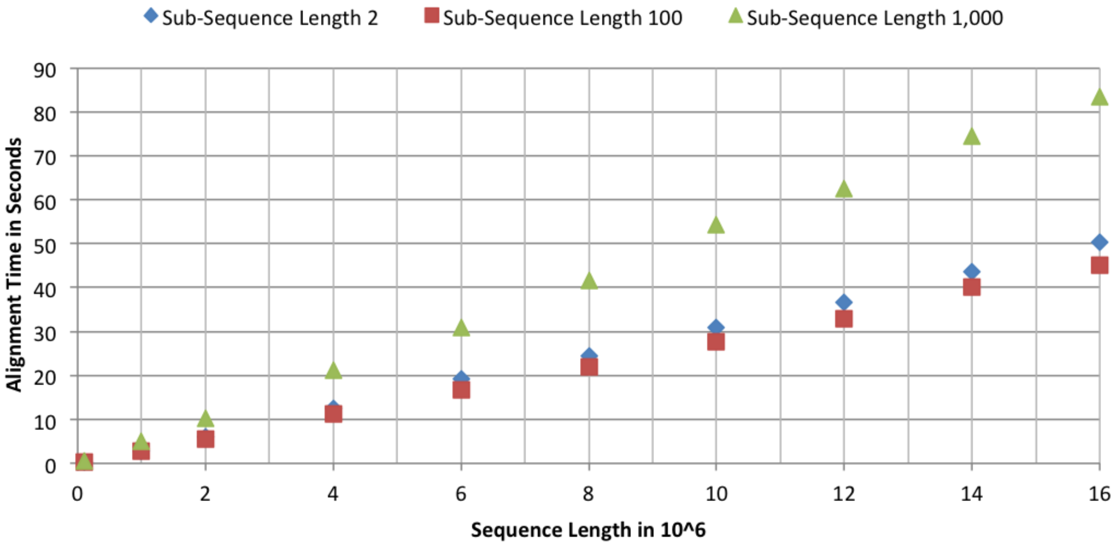


Figure 3.19: Analysis of the influence of varying sub-sequence lengths on the performance of the hierarchical alignment algorithm.

sequence lengths influence the algorithmic performance. The figure shows the time required to align benchmark call trees with the HA algorithm. Sub-sequences are aligned using the Hirschberg algorithm. The benchmark consists of three sets of call trees with rising element numbers. The size of the compared call trees, up to 16 million elements, is given on the x-axis in Figure 3.19. Each call tree set uses a fixed sub-sequence length to split up the complete sequence. For instance, in the case of the second call tree set (red rectangles in Figure 3.19), the complete sequence length of the call tree is separated into multiple sub-sequences of length 100. Figure 3.19 verifies a linear increase in runtime for all test sets. The alignment time for call trees with sub-sequences of length of 1,000 is a bit higher than in the case of sub-sequences with length 100, as the Hirschberg algorithm needs to align longer sub-sequences. Nevertheless, the linear part still dominates the algorithm performance in all test cases. A more detailed performance analysis of the HA algorithm using real-world applications is given in Section 3.5.2.

In this work all comparisons are computed in serial. Yet, the hierarchical alignment algorithm provides potential for parallelization in several ways. A pairwise comparison of two processes does not depend on other processes. Thus, comparing runs with large process counts is embarrassingly parallel. Additionally to the parallel execution of the complete algorithm, also the algorithm itself can be parallelized. During the traversal of the call tree, the algorithm follows multiple paths from the root node to leaf nodes. The computation of different paths can be computed in parallel. Utilizing these options would improve the alignment times presented in this work.

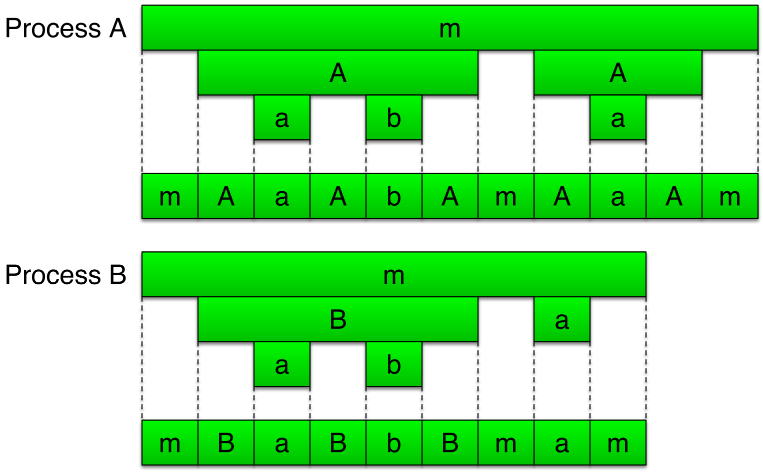
3.4 Differences between Flat and Hierarchical Comparisons

The hierarchical alignment heuristic improves alignment speed considerably, but comes at the price that the alignment result might not be optimal anymore. This section explains the differences between flat and hierarchical alignment results. Furthermore, it introduces a method to measure the error introduced by the hierarchical approach.

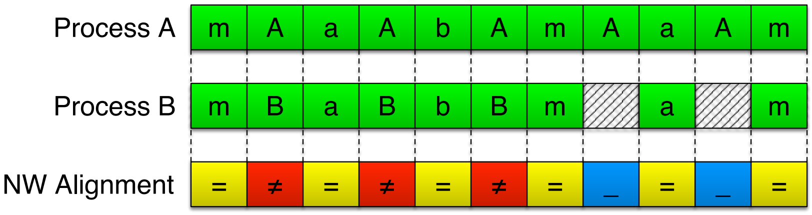
3.4.1 Alignment Errors Introduced by Hierarchical Approach

The hierarchical alignment approach may compute a non-optimal alignment. The source of this error is the task of splitting up the complete function sequence into sub-sequences. In this task the two call trees of the compared processes are traversed in parallel. During the traversal only direct child nodes of related parent nodes are aligned. The resulting final alignment of both processes is constructed from numerous small individual sub-alignment steps. Consequently, each individual alignment step only considers a small part of the complete sequence. While each individual sub-alignment is still optimal, the construction process is not guaranteed to compose an optimal final alignment. This is in contrast to the flat alignment algorithms that consider the complete sequence in one step.

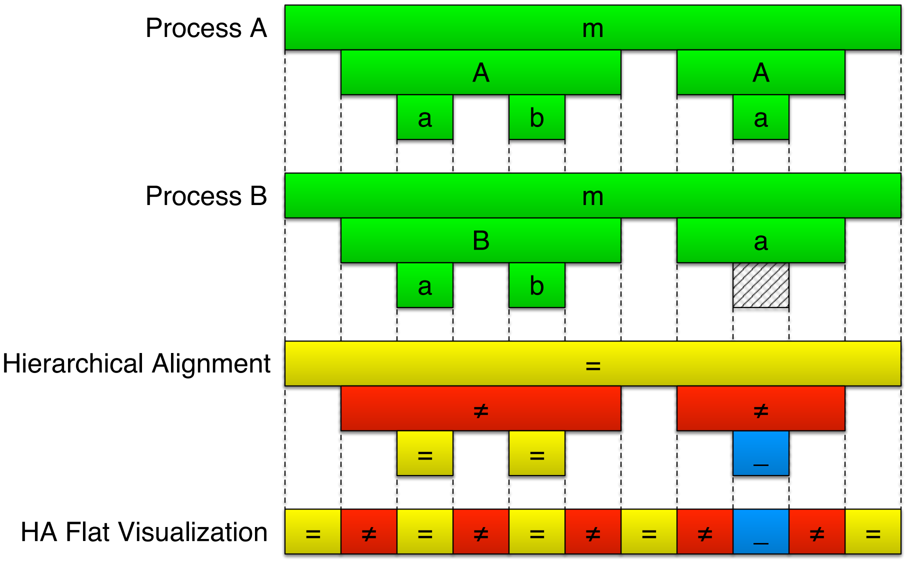
The extent of errors the hierarchical approach introduces depends on how differences between processes are distributed in the call tree. Since the hierarchical approach only considers one call level in each comparison step, it cannot correctly recognize changes spanning multiple call levels. Figure 3.20 demonstrates this situation using an example. In this example the two processes depicted in Figure 3.20(a) are compared. The figure shows the call trees and flat function sequences, respectively. The differences between both processes occur in the middle call level. The first function call `A` in *Process A* is replaced by function call `B` in *Process B*. Additionally, the second function call `A` in *Process A* is deleted in *Process B*. Figure 3.20(b) shows the alignment of the flat function sequences using the Needleman–Wunsch (NW) algorithm. The Needleman–Wunsch algorithm correctly detects both changes. The change of function `A` to function `B` is detected with the three diff states. The deletion of the second function `A` is identified with two gap states. The alignment at the bottom of Figure 3.20(b) represents an optimal alignment of both processes. Figure 3.20(c) shows the result of the hierarchical alignment (HA) algorithm. The HA algorithm detects the first change correctly. Since the algorithm aligns child nodes regardless of the alignment state of the parent nodes, it detects the similar child nodes `a` and `b` correctly. However, the HA



(a) Call trees and flat function sequences of the two input processes. Changes occur in the middle call level.

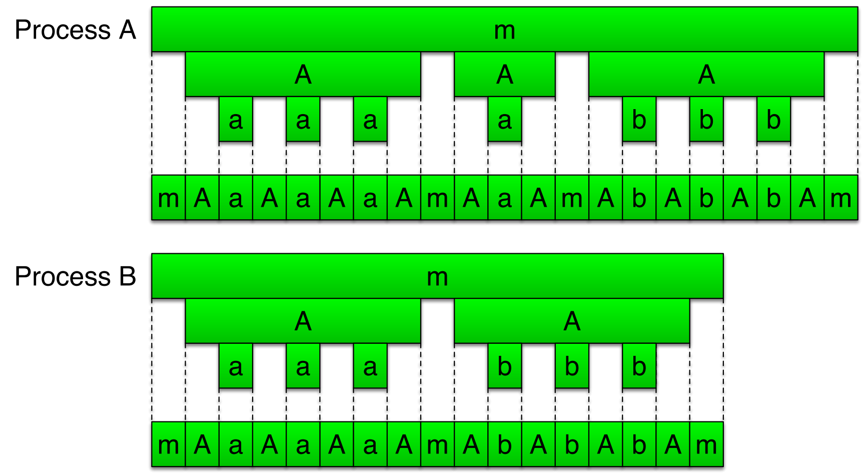


(b) Optimal alignment of the flat function sequences calculated with the Needleman–Wunsch (NW) algorithm.

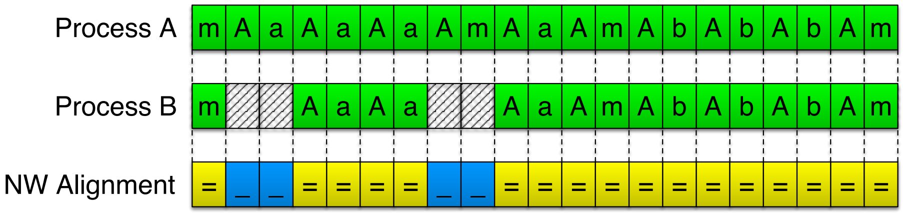


(c) Non-optimal result of the hierarchical alignment (HA) algorithm. The deletion of the second function `A` on *Process B* is detected wrong. Instead with a gap, the function `A` on *Process A* is aligned with the function `a` of *Process B*.

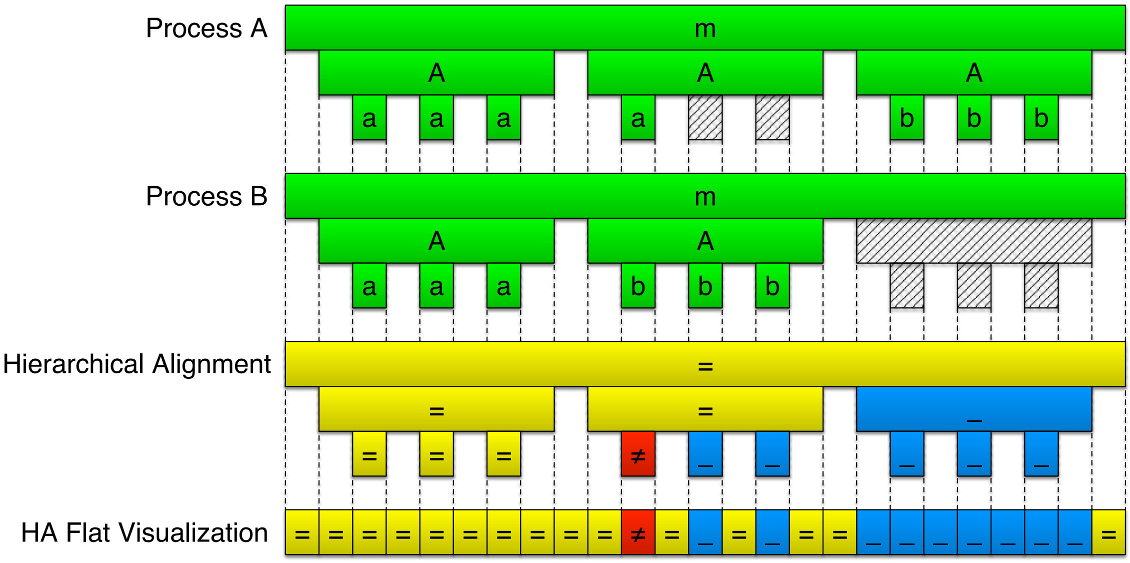
Figure 3.20: Example showing a non-optimal result of the hierarchical alignment algorithm. The algorithm cannot detect changes spanning multiple call levels.


Process A
m
A
A
A
a a a
a
b b b
m A a A a A a A m A a A m A b A b A b A m
Process B
m
A
A
a a a
b b b
m A a A a A a A m A b A b A b A m

(a) Call trees and flat function sequences of the two input processes. One invocation of function `A` is added in the middle of *Process A*.


Process A
m A a A a A a A m A a A m A b A b A b A m
Process B
m A a A a A a A m A b A b A b A m
NW Alignment
= _ _ = = = = _ _ = = = = = = = = = = = =

(b) Optimal alignment of the flat function sequences calculated with the Needleman–Wunsch (NW) algorithm.



(c) Non-optimal result of the hierarchical alignment (HA) algorithm. The algorithm does not detect the insertion of the second function `A` on *Process A* and aligns the wrong invocations of function `A`.

Figure 3.21: Example showing a non-optimal result of the hierarchical alignment algorithm due to disregarded sub-levels (child nodes) of aligned nodes.

algorithm fails to detect the second change correctly. Since the second sub-alignment step only considers child nodes of the function `m`, the HA algorithm aligns the function `A` of *Process A* with function `a` of *Process B*. This results in a diff state for this comparison. The subsequent alignment step for these two nodes can only align the function `a` of *Process A* to a gap in *Process B*. Due to this inaccurate detection of the second change the resulting constructed final alignment, shown at the bottom of Figure 3.20(c), is not optimal anymore.

A different source of non-optimal alignments stems from the fact that sub-levels are not considered in individual alignment steps. Figure 3.21 provides an example. As depicted in Figure 3.21(a), this example compares three calls to function `A` in *Process A* with two calls to function `A` in *Process B*. The difference between both processes is the middle function call `A` that is deleted in *Process B*. The Needleman–Wunsch algorithm detects this change mathematically correctly, shown in Figure 3.21(b). The algorithm cannot differentiate between the four calls from `A` to `a` in *Process A*. The used implementation of the Needleman–Wunsch algorithm selects the first call from `A` to `a` as gap. This choice might not reflect the intuitive result expected by the analyst, but nevertheless represents an optimal solution. The HA algorithm aligns all `A` function calls in the second sub-alignment step, as shown in Figure 3.21(c). Since the algorithm disregards the sub-levels (child nodes) of aligned nodes, all `A` function calls are considered as equivalent. In this case, the HA algorithm aligns the first two `A` function calls of each process. The third function call to `A` in *Process A* is aligned to a gap. This choice results in a non-optimal alignment as the second function call to `A` in *Process B* is not aligned to its counterpart in *Process A* anymore. Compared with the optimal alignment shown in Figure 3.21(b), the constructed final alignment depicted at the bottom in Figure 3.21(c) shows more gap states and one diff state.

Yet, besides the described examples, there is also the possibility that the HA algorithm computes an optimal alignment. Depending on the distribution of changes in the call tree, the HA result error may vary. Therefore, the next section describes a method to quantify the error introduced by the HA approach.

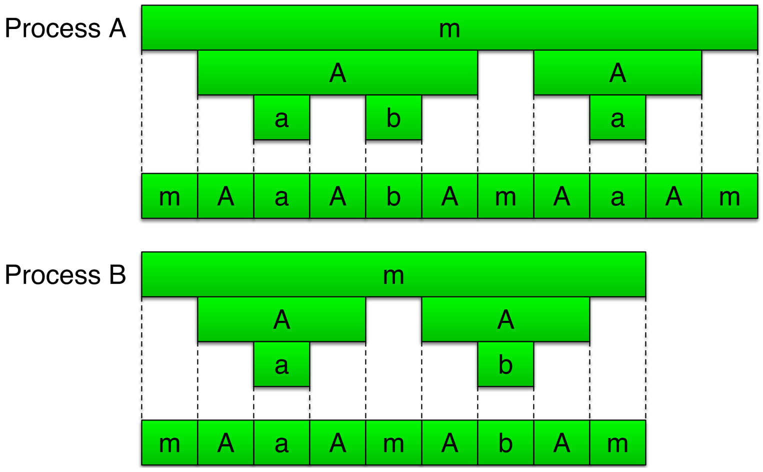
3.4.2 Comparison of Hierarchical and Flat Alignments

The hierarchical alignment algorithm is not guaranteed to compute optimal alignments. A way of measuring the error introduced by this heuristic approach is to compare alignment scores. Therefore the alignment for two processes is computed with a flat alignment algorithm and the HA algorithm, respectively. Then, based on the following scoring scheme, the score of each alignment can be calculated by adding up all states.

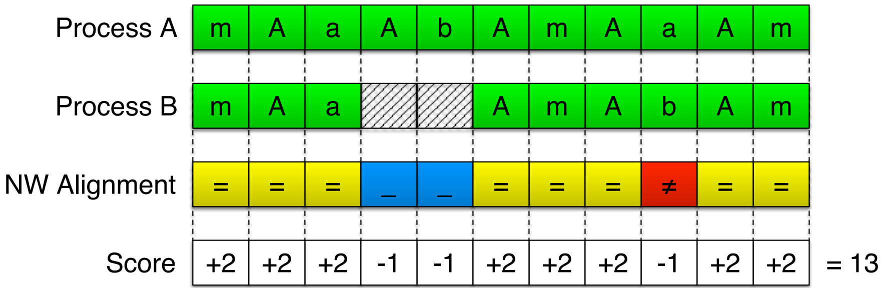
$$\begin{aligned} \text{Match Score:}\ & 2 \\ \text{Mismatch Score:}\ & -1 \\ \text{Gap Score:}\ & -1 \end{aligned}$$

However, the score number derived directly from both alignments is unsuitable for comparison. Depending on the changes between the input processes, the scores even for similar alignment results may differ. Figure 3.22 demonstrates how direct scores for flat and hierarchical alignments differ. The correct alignment score for the example shown in Figure 3.22 is 13. Figure 3.22(b) demonstrates how this score is derived from the Needleman–Wunsch alignment. The flat and the hierarchical algorithm produce optimal alignments for this example. Consequently, both alignments should yield the same alignment score. However, taking the score directly from the HA, shown in the middle of Figure 3.22(c), yields a too low number. Taking the score from the flat sequence of the HA, shown in the bottom of Figure 3.22(c), yields a too high number.

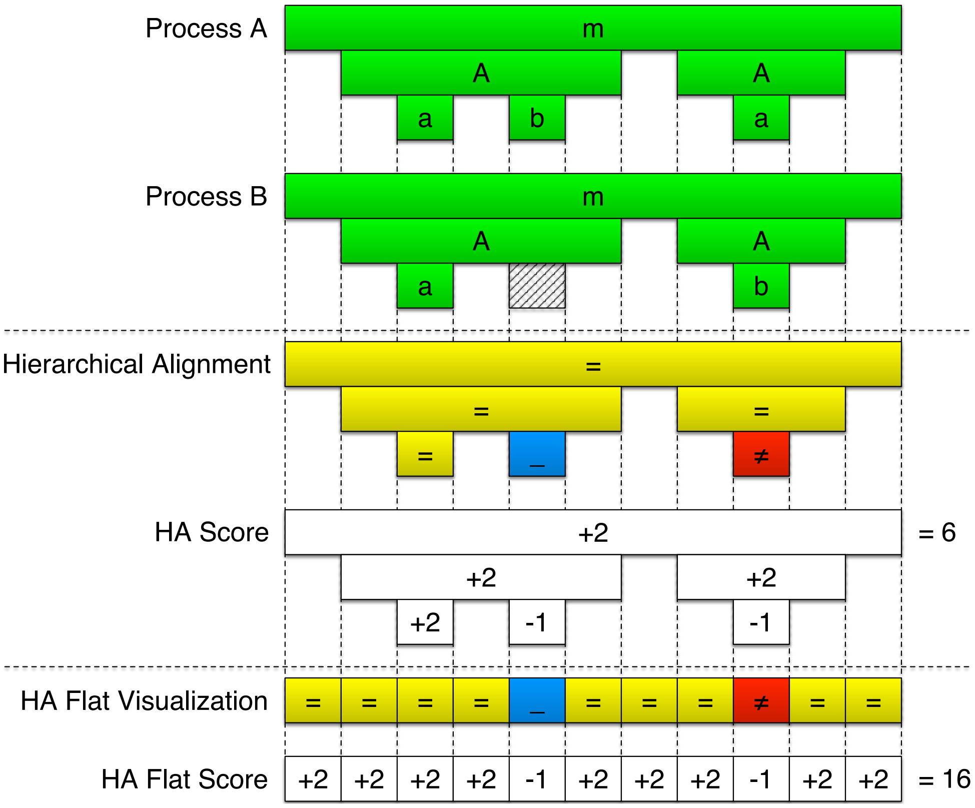
In order to provide a suitable measure for the comparison of alignments, the extraction of the score from the HA requires adaption. First, to be comparable with the Needleman–Wunsch alignment result, the flat sequence of the HA needs to be considered. The difference causing dissimilar scores between the Needleman–Wunsch alignment result and the flat sequence of the HA result is the representation of *Gap* states. If gaps occur below the top call level, each gap always results from two *Gap* states in the



(a) Two input processes. Differences occur on the last call level.



(b) Optimal alignment of the flat input sequences computed with the Needleman–Wunsch (NW) algorithm. The alignment achieves a score of 13.



(c) Optimal alignment result computed with the hierarchical approach (HA). Taking the score from the HA yields only a score of 6. Taking the score from the flat visualization of the HA yields a score of 16.

Figure 3.22: Scores for flat and hierarchical alignments may differ and are not directly comparable.

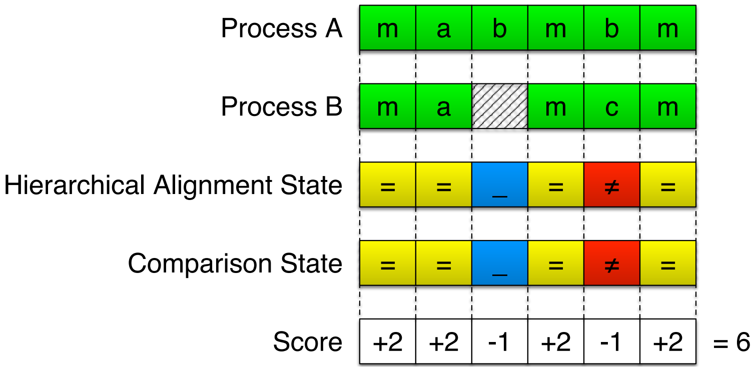


Figure 3.23: HA score correction. In case of differences on the top call level no correction is necessary. Flat and hierarchical alignment scores are equal.

Needleman–Wunsch algorithm. This is due to the part of the code that calls a removed function (i.e., *Gap*). This part in the parent node of the *Gap* is also removed along with the removed function. In the example in Figure 3.22(a) the first function `A` of *Process A* is executed in three parts. *Process B* executes the same function in only two parts. The reason is that this function `A` does not call function `b` on *Process B*. Consequently, the missing third part of function `A` on *Process B* needs to be represented by a *Gap* state. This results in the following adaption requirement for the extraction scores from flat HA sequences. For each *Gap* below the top call level, an additional *Gap* needs to be selected for the state occurring before the *Gap* state. With this adaption the score extracted from the HA result equals the score extracted from the NW alignment.

The following three examples demonstrate the extraction of comparable alignment scores.

No adaptations are necessary in the case of gaps and differences occurring on the top call level. The scores extracted from the HA algorithm and the Needleman–Wunsch algorithm can be compared directly. Figure 3.23 illustrates this case with the comparison of the following two processes. *Process A*: `m a b m b m` and *Process B*: `m a m c m`. The HA result achieves a score of 6. The Needleman–Wunsch alignment result for the same sequences is shown as follows:

```
        m   a   b   m   b   m
        |   |       |   -   |
        m   a   -   m   c   m
Score: +2  +2  -1  +2  -1  +2 = 6
```

The flat algorithm also yields an alignment score of 6. For changes on the top call level, both alignment results are directly comparable.

The situation is different when changes occur on intermediate call levels. As described above, corrections for the *Gap* costs need to be applied for the HA result. Figure 3.24 demonstrates the required corrections with an example comparing *Process A*: `m A a A b A m B c B m` and *Process B*: `m C d C m`. As illustrated in Figure 3.24 with a shaded field, the first comparison state also needs to be counted as *Gap*. This field marks the calling code on the parent node of the large *Gap* series that represents the missing function `A` on *Process B*. With applied correction the HA results in a score of -5. The alignment result for this example calculated with the Neddleman-Wunsch algorithm is as follows:

```
        m   A   a   A   b   A   m   B   c   B   m
                                |   -   -   -   |
        -   -   -   -   -   -   m   C   d   C   m
Score: -1  -1  -1  -1  -1  -1  +2  -1  -1  -1  +2 = -5
```

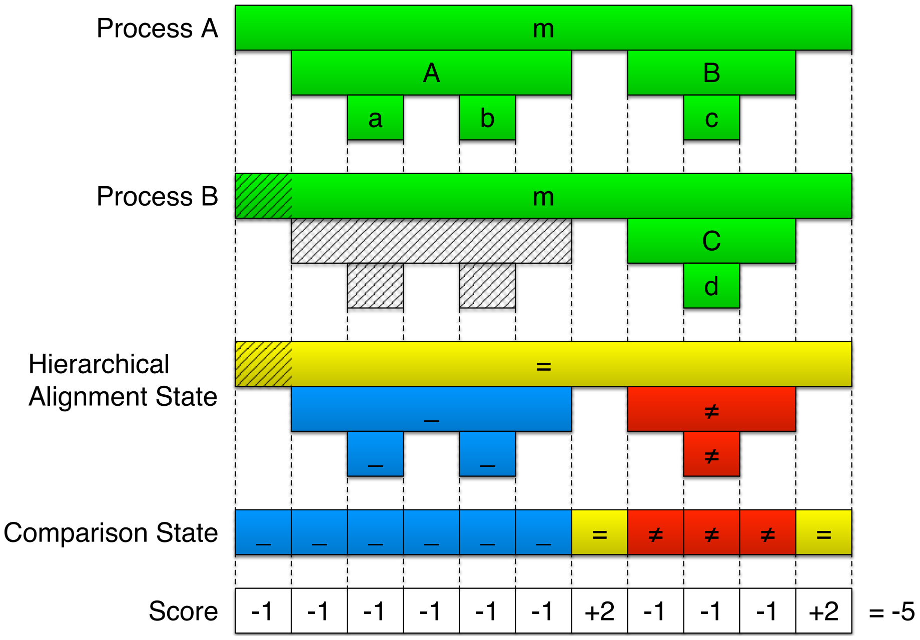


Figure 3.24: HA score correction for differences on the middle call level. Gap states require an additional gap state on the above call level to mark the missing calling code. With applied correction, the HA yields a score of -5 for the example.

Also the alignment from the Needleman–Wunsch algorithm yields a score of -5. When applying the gap corrections, the scores of both alignment algorithms agree.

In case of changes on the bottom levels, or the leaf nodes, of the call tree, the correction procedure works accordingly to changes on middle levels. Figure 3.25 shows an example comparison of *Process A*: `m A a A b A m A a A m` with *Process B*: `m A a A m A c A m`. This example matches the situation shown in the beginning in Figure 3.22. With applied corrections the HA algorithm now results in an alignment with the score of 13. This is consistent with the result of the Needleman–Wunsch algorithm as shown below.

```
        m  A  a  A  b  A  m  A  a  A  m
        |  |  |        |  |  |  -  |  |
        m  A  a  -  -  A  m  A  c  A  m
Score: +2 +2 +2 -1 -1 +2 +2 +2 -1 +2 +2 = 13
```

The introduced corrections to the calculation of the HA alignment score result in a comparable alignment score. The described method provides a measure to quantify the error introduced by the HA algorithm heuristic.

3.5 Evaluation

This section evaluates the performance and accuracy of the introduced alignment algorithms. It starts with an analysis of dynamic programming algorithms using multiple flat alignment cases. Then, a case study with real-world applications evaluates both the performance of the hierarchical approach in contrast to flat alignments and the error between flat and hierarchical alignments.

The case study in this section uses the following two real-world applications. AMG2006 [34], a parallel algebraic multigrid solver for linear systems arising from problems on unstructured grids. ParaDiS [7], a simulation that models the dynamics of dislocation lines as they interact and move in response to the forces imposed by external stress and inter-dislocation interactions.

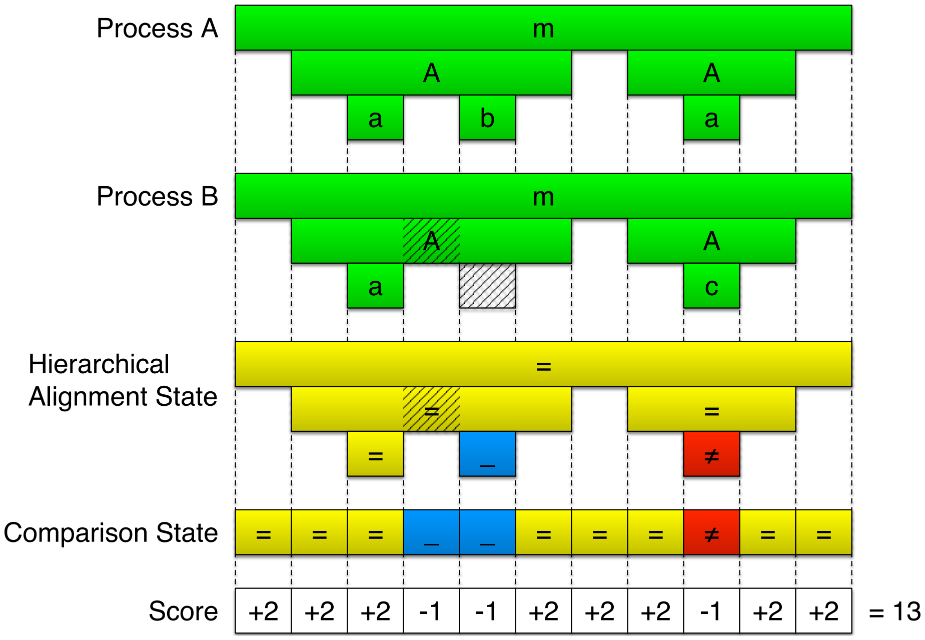


Figure 3.25: HA score correction for differences on the bottom call level. According to changes on the middle call levels, gap states on bottom call levels need an additional gap on the above call level. With applied correction the HA yields a score of 13 for the shown example.

All performance measurements have been conducted on an HPC system with the following setup. One compute node is equipped with two Intel Xeon E5-2680 v3 processors. Each CPU provides 12 cores and runs at 2.50 GHz with hyper-threading disabled. The available memory of a node ranges from 64 GB to 256 GB. For the alignment benchmarks, one node has been reserved exclusively with only one core running the benchmark actively. All reported results in this section are always the median of ten repeated measurement runs.

3.5.1 Performance of Flat Alignment Algorithms

To evaluate the performance of the described dynamic programming algorithms, input sequences with rising lengths are compared. In this evaluation, the following sequence lengths have been used: 100, 1 k, 10 k, 100 k, 1 M.

The alignment algorithms use a dynamic programming matrix to store computed results. Algorithms with quadratic memory complexity, such as the Needleman–Wunsch algorithm, need to hold the complete matrix in memory. Table 3.1 shows the memory consumption of such algorithms with respect to the input sequence length. Both input sequences are assumed to have the same length.

Table 3.1: Memory requirement for the Needleman–Wunsch algorithm.

SEQUENCE LENGTH	MEMORY REQUIREMENT
100	~39 kB
1 k	~4 MB
10 k	~381 MB
100 k	~37 GB
1 M	~3725 GB

Following the results of Table 3.1, it is not possible to compute sequences longer than 100 k elements with the available main memory of the test system. Algorithms with linear memory complexity, such as the Hirschberg algorithm, are not restricted by that quadratic factor and are capable of computing sequence lengths beyond 1 M elements.

Since the performance of Ukkonen's algorithm depends on the similarity of the input sequences, this evaluation uses five different input sets. Each input set provides a fixed similarity, ranging from completely similar to completely dissimilar sequences. The individual input sets and the related benchmark results are described in the following.

Equal In this test case both sequences are completely similar.

```
a a a a a a a a a a a a a a a a a a a a
| | | | | | | | | | | | | | | | | | | |
a a a a a a a a a a a a a a a a a a a a
```

Half-Equal In this test case one constant sequence is compared with an alternating sequence. Both sequences share 50% similarity.

```
a a a a a a a a a a a a a a a a a a a a
| – | – | – | – | – | – | – | – | – | –
a b a b a b a b a b a b a b a b a b a b
```

Small Equal Blocks This test case sequences consist of repetitions of a small block of high similarity followed by a large block of dissimilarity. In total both input sequences share 30% similarity.

```
a a a a a a a a a a a a a a a a a a a a
| | | – – – – – – – | | | – – – – – – –
a a a b b b b b b b a a a b b b b b b b
```

Large Equal Blocks This test case sequences consist of repetitions of a large block of high similarity followed by a small block of dissimilarity. In total both input sequences share 70% similarity.

```
a a a a a a a a a a a a a a a a a a a a
| | | | | | | – – – | | | | | | | – – –
a a a a a a a b b b a a a a a a a b b b
```

Different In this test case both input sequences are completely different. They share no similarity.

```
a a a a a a a a a a a a a a a a a a a a
– – – – – – – – – – – – – – – – – – – –
b b b b b b b b b b b b b b b b b b b b
```

Tables 3.2, 3.3, 3.4, 3.5, and 3.6 summarize the benchmark results for all input sets and sequence lengths.

The Needleman–Wunsch and Hirschberg algorithms are independent from the input data similarity. Consequently, both algorithms show similar runtimes across all input data test sets. With rising sequence lengths, the Needleman–Wunsch and Hirschberg algorithm show the expected quadratic increase in runtime. Figure 3.27 illustrates this behavior. The value range is plotted on a logarithmic scale in Figure 3.27. Thus, the visible linear increase translates to quadratic rise on a linear scale.

Compared with the Needleman–Wunsch algorithm runtimes, the Hirschberg algorithm shows a considerable higher runtime. This is caused by the numerous re-computations necessary to achieve linear memory complexity.

Table 3.2: Alignment times for equal sequences.

	SEQUENCE LENGTH				
ALGORITHM	100	1,000	10,000	100,000	1,000,000
Needleman–Wunsch	<1ms	4ms	562ms	54s 938ms	`Out of Memory`
Hirschberg	<1ms	29ms	2s 869ms	287s 932ms	8h 11min
Ukkonen	<1ms	<1ms	<1ms	2ms	21ms

Table 3.3: Alignment times for half-equal sequences.

	SEQUENCE LENGTH				
ALGORITHM	100	1,000	10,000	100,000	1,000,000
Needleman–Wunsch	<1ms	4ms	563ms	54s 946ms	`Out of Memory`
Hirschberg	<1ms	29ms	2s 897ms	289s 123ms	8h 8min
Ukkonen	<1ms	3ms	149ms	12s 956ms	21min 11s

Table 3.4: Alignment times for small equal blocks sequences.

	SEQUENCE LENGTH				
ALGORITHM	100	1,000	10,000	100,000	1,000,000
Needleman–Wunsch	<1ms	4ms	565ms	54s 833ms	`Out of Memory`
Hirschberg	<1ms	29ms	2s 880ms	289s 915ms	8h
Ukkonen	<1ms	3ms	233ms	22s 980ms	40min 13s

Table 3.5: Alignment times for large equal blocks sequences.

	SEQUENCE LENGTH				
ALGORITHM	100	1,000	10,000	100,000	1,000,000
Needleman–Wunsch	<1ms	4ms	563ms	54s 819ms	`Out of Memory`
Hirschberg	<1ms	29ms	2s 906ms	290s 982ms	8h 6min
Ukkonen	<1ms	1ms	45ms	4s 378ms	7min 37s

Table 3.6: Alignment times for completely dissimilar sequences.

	SEQUENCE LENGTH				
ALGORITHM	100	1,000	10,000	100,000	1,000,000
Needleman–Wunsch	<1ms	4ms	563ms	54s 811ms	`Out of Memory`
Hirschberg	<1ms	29ms	2s 870ms	290s 426ms	8h
Ukkonen	<1ms	4ms	344ms	34s 273ms	60min 51s

Ukkonen's algorithm benefits from high similarity between input sequences. In the test case with equal sequences, the algorithm only needs to compute the main diagonal. This results in only linear increase of runtime with respect to sequence lengths. Even sequences with lengths of 1 M elements can be aligned in a few milliseconds. With rising dissimilarity, Ukkonen's algorithm computes the alignment slower. The increase in runtime is the direct result of the dissimilarity between the input sequences. The dissimilarity forces the algorithm to compute more cells in the dynamic programming matrix. Yet, compared to the two other algorithms, Ukkonen's algorithm still computes the result noticeably faster. Even for completely dissimilar input sequences Ukkonen's algorithm computes the alignment considerably faster than the other two algorithms. The reason is that this implementation of Ukkonen's algorithm detects that both sequences are completely dissimilar. In this special case the algorithm can stop the divide-and-conquer process and align both sequences completely on the main diagonal of the dynamic programming matrix. Since the related sequence pairs (on the main diagonal) are not equal, the algorithm cannot directly reach down the main diagonal but also needs to compute cells around the main diagonal. However, the number of computed cells is still considerably lower than in the case of the other two algorithms. The Needleman–Wunsch algorithm always computes the complete dynamic programming matrix. The Hirschberg algorithm also computes the complete matrix with many cells computed repeatedly. Table 3.7 compares the number of computed cells in the dynamic programming matrix for an example alignment of two sequences of 10 characters in length. In the first case, both input sequences are completely similar. In the second case, both input sequences are completely dissimilar.

Table 3.7: Number of computed dynamic programming matrix cells for an example alignment of two character sequences of length 10.

ALGORITHM	EQUAL INPUT SEQUENCES	DISSIMILAR INPUT SEQUENCES
Needleman–Wunsch	121	121
Hirschberg	304	304
Ukkonen	11	61

Discussion of Benchmark Results The quadratic memory requirement of the Needleman–Wunsch algorithm renders its use for the alignment of large sequences impossible due to the limited size of available main memory. Sequences with lengths of about 100k elements already need special HPC systems for the alignment. Aligning sequences with more than 1M elements is impossible even on most HPC systems. Sequences constructed from real-world applications easily contain more than 1M elements. This forbids the use of the Needleman–Wunsch algorithm for the alignment of application sequences.

The Hirschberg and Ukkonen's algorithm do not present that limitation. By applying a divide-and-conquer strategy they require only linear memory complexity to compute an alignment. However, this comes at the cost of recomputing erased values. Thus, especially the Hirschberg algorithm runs slower than the Needleman–Wunsch algorithm. With runtimes of more than eight hours for sequences with 1M elements, it is not feasible to use this algorithm for the alignment of real-world application sequences.

The Needleman–Wunsch and Hirschberg algorithm's performance is independent from the similarity of the input sequences. As shown in the five test cases, the alignment times of both algorithms stay almost constant for input sequences with changing similarities. Ukkonen's algorithm presents different behavior. This algorithm performs faster for sequences with higher similarity. The measurements in five test cases support this thesis. The higher the similarity of the input sequences, the shorter the runtime of the algorithm. Figure 3.26 compares the alignment times of the Needleman–Wunsch algorithm and Ukkonen's algorithm across all test cases. Shown are the alignment times for sequences with 10,000 elements length.

In general all algorithms show a quadratic increase in runtime with rising sequence lengths. Figure 3.27 depicts the alignment times for all three algorithms on a logarithmic scale. Shown are the times required to align completely different input sequences. The quadratic increase in runtime results in very high alignment time for long input sequences.

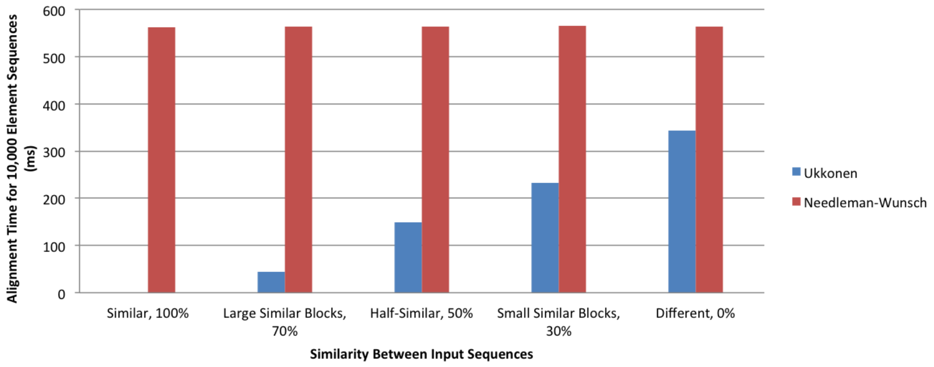


Figure 3.26: Alignment times for changing input similarities.

For sequences sharing very high similarity, Ukkonen's algorithm has the potential to align even large sequences in an acceptable time. However, this assumption cannot be guaranteed for the comparison of sequences constructed from real-world applications. Additionally, the length of such sequences may easily exceed 1M elements. This might also result in unacceptable alignment times using Ukkonen's algorithm.

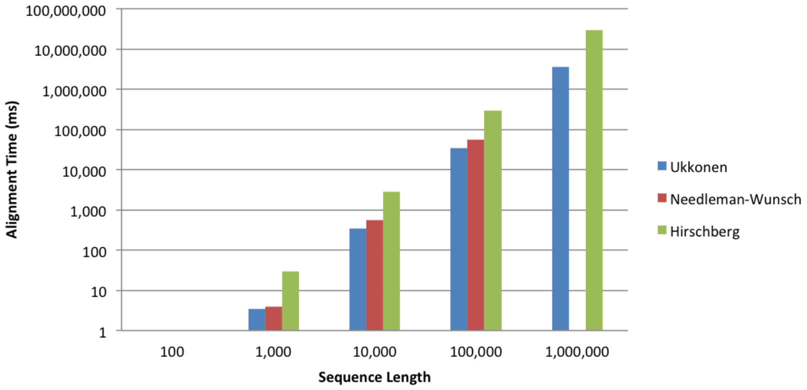


Figure 3.27: Alignment times for dissimilar input sequences.

Summary Dynamic programming algorithms allow a detailed detection of structural similarities and differences between application processes. Yet, although the dynamic programming approach satisfies the functionality requirement, the quadratic time complexity forbids an alignment of long sequences in a reasonable time. A method to speed up the alignment is needed.

3.5.2 Performance of the Hierarchical Alignment Algorithm

This section contrasts the performance of the dynamic programming algorithms with the performance of the hierarchical approach for the comparison of real-world applications. Therefore, four test cases have been created using the two applications AMG2006 [34] and ParaDiS [7].

To provide a measure for similarity, the input sequences have been aligned using the hierarchical approach. Then, the percentage of equal areas in the alignment indicates the similarity between the input sequences.

For each test case a table provides an overview of how the hierarchical approach splits up the input sequences into sub-sequences. Each sub-alignment step is counted in the table and grouped by the length of its input sub-sequences. In the case that the length of the two input sub-sequences differs, the longer sequence is used as grouping criterion.

AMG2006 In this test case, sequences from two different versions of the application AMG2006 [34] have been compared. The two input sequences are constructed from the first process of each application version. The lengths of the two sequences are 51,205 and 35,618, respectively. Both sequences share a similarity of 60% equal areas.

Table 3.8: Alignment times for AMG2006 processes.

ALGORITHM	ALIGNMENT TIME
Needleman–Wunsch	$9s$ $902ms$
Hirschberg	$55s$ $235ms$
Ukkonen, with speedup techniques	$687ms$
Ukkonen, without speedup techniques	$711ms$
Hierarchical Alignment, with speedup techniques	$35ms$
Hierarchical Alignment, without speedup techniques	$40ms$

This first test case requires the alignment of relatively short sequences. This allows to conduct an alignment with the Needleman–Wunsch algorithm. All runtimes of the three flat alignment algorithms are in the range of the alignment times already measured in the previous benchmarks with artificial input sequences. Both input sequences share high similarity that results in considerably faster runtime of Ukkonen's algorithm compared to Needleman–Wunsch and Hirschberg. Using speedup techniques allows to achieve a small performance improvement for Ukkonen's algorithm. The runtime decreases by 3.4% with enabled speedup techniques.

Using the hierarchical approach speeds up the alignment performance considerably. The hierarchical alignment only needs 5.6% of the time required by Ukkonen's algorithm. The usage of speedup techniques additionally decreases the runtime of the hierarchical alignment by 12.5%.

Table 3.9: Number of sub-alignments during the hierarchical alignment of AMG2006.

SUB-SEQUENCE LENGTH	NUMBER OF ALIGNMENTS
0 - 100	4,682
101 - 1,000	7
1,001 - 5,000	1

As shown in Table 3.9, the hierarchical approach splits up the input sequences into sub-sequences with primarily less than 100 elements length. Due to the short length of the input sequences, all algorithms are capable to produce an alignment in less than one minute for this test case.

ParaDiS The second test case is a comparison of processes from two versions of the application ParaDiS [7]. The lengths of the input sequences are 1,953,973 and 2,379,035, respectively. The similarity between both sequences is about 40%. The first process of each application version is compared in this test.

Table 3.10: Alignment times for ParaDiS processes.

ALGORITHM	ALIGNMENT TIME
Needleman–Wunsch	`Out of Memory`
Hirschberg	$>24h$
Ukkonen, with speedup techniques	$3h$ $13min$
Ukkonen, without speedup techniques	$3h$ $19min$
Hierarchical Alignment, with speedup techniques	$11s$ $291ms$
Hierarchical Alignment, without speedup techniques	$11s$ $571ms$

The length of the input sequences forbids an alignment with the Needleman–Wunsch algorithm. The Hirschberg algorithm requires more than 24 hours to compute an alignment. Even Ukkonen's algorithm with active speedup techniques runs for more than three hours to compute the alignment.

The hierarchical alignment algorithm needs only about 11 seconds for the alignment. This presents a significant performance improvement over the flat dynamic programming algorithms. This is possible by splitting up the input sequence into 268,692 sub-sequences, see Table 3.11.

Table 3.11: Number of sub-alignments during the hierarchical alignment of ParaDiS.

SUB-SEQUENCE LENGTH	NUMBER OF ALIGNMENTS
0 - 100	267,622
101 - 1,000	822
1,001 - 5,000	132
5,001 - 10,000	116

All flat dynamic programming algorithms cannot compute the alignment in reasonable time. Only the hierarchical approach is applicable for the comparison of such long application sequences.

ParaDiS with Increased Detail This test case uses the application ParaDiS [7] again. In contrast to the previous test, the application processes have been measured with increased detail. The measurement data of this test case includes more application activities, and thus, results in longer input sequences. The two sequences have lengths of 22,486,844 and 20,387,720, respectively. In this test, the first process is compared with the second process of the same application. Both input sequences share about 90% equal areas after the hierarchical alignment.

Table 3.12: Alignment times for ParaDiS processes (increased detail version).

ALGORITHM	ALIGNMENT TIME
Needleman–Wunsch	`Out of Memory`
Hirschberg	$>24h$
Ukkonen, with speedup techniques	$2h$ $16min$
Ukkonen, without speedup techniques	$2h$ $20min$
Hierarchical Alignment, with speedup techniques	$22s$ $392ms$
Hierarchical Alignment, without speedup techniques	$24s$ $277ms$

Contrary to the previous case, Ukkonen's algorithm performs faster for this comparison. This is due to the considerably higher similarity of the input sequences. This allows a more efficient computation of Ukkonen's algorithm.

Table 3.13: Number of sub-alignments during the hierarchical alignment of ParaDiS (increased detail).

SUB-SEQUENCE LENGTH	NUMBER OF ALIGNMENTS
0 - 100	4,973,475
101 - 1,000	926
1,001 - 5,000	70
5,001 - 10,000	164
>10,000	53

As shown in Table 3.13, the hierarchical approach mainly needs to align sequences with less than 100 elements length.

ParaDiS Compared with AMG2006 This test case demonstrates the comparison of completely unrelated application processes. In this test the first process of ParaDiS [7] is compared with the first process of AMG2006 [34]. The length of both input sequences are 1,953,973 and 35,618, respectively. Both sequences share no similarity and show 0% similarity after the hierarchical alignment.

Table 3.14: Alignment times for the comparison of ParaDiS with AMG2006.

ALGORITHM	ALIGNMENT TIME
Needleman–Wunsch	`Out of Memory`
Hirschberg	$34min$ $33s$
Ukkonen, with speedup techniques	$7min$ $10s$
Ukkonen, without speedup techniques	$7min$ $9s$
Hierarchical Alignment, with speedup techniques	$1s$ $147ms$
Hierarchical Alignment, without speedup techniques	$1s$ $145ms$

Due to the input sequence length, applying the Needleman–Wunsch algorithm exceeds available main memory of the test system. The hierarchical alignment needs about one second for the comparison of both processes. This time is spent primarily for traversing the larger call tree of the first process.

Table 3.15: Number of sub-alignments during the hierarchical alignment of ParaDiS with AMG2006.

SUB-SEQUENCE LENGTH	NUMBER OF ALIGNMENTS
0 - 100	1
101 - 1,000	1

Table 3.13 shows the number of sub-alignments computed during hierarchical alignment. Since both call trees are unrelated, the algorithm can already stop computing alignments on top call levels.

Summary For the alignment of sequences constructed from real-world applications, only the hierarchical alignment algorithm provides acceptable runtimes. The hierarchical approach effectively separates the input sequences into small sub-sequences. Although not occurring in the evaluated case studies, there is still the possibility that long sub-sequences need to be aligned. This might be the case if functions are called very often in a loop. In such case the fast-alignment-heuristic can be applied to that sub-sequence. The heuristic allows a fast alignment time at the price of decreased accuracy. The use of the hierarchical alignment algorithm satisfies the functional and temporal requirements for process comparison.

3.5.3 Accuracy of the Hierarchical Alignment Algorithm

In this case study the gap-correction-method, described in Section 3.4.2, is applied to evaluate the error introduced when using the hierarchical alignment (HA) algorithm for comparing real-world applications.

Therefore, alignment results are taken from the four test cases of the previous section. For alignments computed with Ukkonen's algorithm and the HA algorithm scores are calculated and compared to quantify the HA result error.

AMG2006

$$\begin{aligned} \text{Ukkonen's algorithm alignment score: } & 97,957 \\ \text{Hierarchical alignment algorithm alignment score: } & 87,871 \\ \text{Error: } & 10.3\% \end{aligned}$$

For the comparison of processes of AMG2006 [34] the scores of the two algorithms differ by about 10%. This low error suggests that the alignment result of the HA algorithm is still applicable for process comparison.

ParaDiS

$$\begin{aligned} \text{Ukkonen's algorithm alignment score: } & 909,907 \\ \text{Hierarchical alignment algorithm alignment score: } & 800,597 \\ \text{Error: } & 12\% \end{aligned}$$

In case of ParaDiS [7] the introduced error by the HA approach is similar to the error seen for AMG2006. With about 12% error, the HA algorithm produces applicable results for the comparison of ParaDiS processes.

ParaDiS with Increased Detail

$$\begin{aligned} \text{Ukkonen's algorithm alignment score: } & 77,243,841 \\ \text{Hierarchical alignment algorithm alignment score: } & 77,192,042 \\ \text{Error: } & 0.07\% \end{aligned}$$

In case of comparing ParaDiS [7] processes measured with increased detail, the HA introduces only very few errors. With 0.07% difference the results of the two algorithms are almost equal.

ParaDiS Compared with AMG2006

$$\begin{aligned} \text{Ukkonen's algorithm alignment score: } & -3,913,725 \\ \text{Hierarchical alignment algorithm alignment score: } & -3,978,636 \\ \text{Error: } & 1.6\% \end{aligned}$$

Also when comparing completely different processes, like in the case of aligning processes of ParaDiS [7] with processes of AMG2006 [34], the introduced error is very low. The HA approach result differs by only 1.6% from the result computed by Ukkonen's algorithm.

Summary For the comparison of processes, the HA heuristic provides a large performance improvement over flat alignment algorithms. The introduced error by the heuristic is relatively small in all tested comparison cases. This underlines the applicability of the HA approach for process comparison.

4 Alignment-Based Comparison Metrics for Processes

The previous chapter introduced a hierarchical alignment algorithm for the structural comparison of event streams of processes. Yet, efficient alignment algorithms alone are not sufficient to help users understand the differences between processes. In order to compare the performance of processes, their timing behavior needs to be considered as well. Therefore, supporting the alignment approach, this section contributes a set of novel alignment-based metrics that quantify the differences between processes, both in terms of differences in the event stream and timing differences across events. Further, it introduces visualization techniques that highlight and facilitate understanding of the sources of the differences. A case study demonstrates the effectiveness of the introduced approach by showing automatically detected performance and code differences across different versions of one benchmark and two real-world applications.

4.1 Similarity Metric

The *Similarity Metric* describes the similarity between two traces (event streams). This metric is based on the alignment algorithms described in Chapter 3. The alignment algorithms work with the following scoring scheme, which is now used to derive a metric stating the similarity of two traces:
Match Score: $\sigma_{equal} = 2$, Mismatch Score: $\sigma_{diff} = -1$, Gap Score: $\sigma_{gap} = -1$.

Using the raw score, however, is problematic. In an alignment, each function pair, dependent on its state—equal, different, or gap—represents the score defined in the scoring scheme. The sum of all scores, $Score_{actual}$, results in positive scores for equal areas and penalizes differences and gaps with negative scores. Thus, the higher the total score, the higher the similarity between the processes. However, the actual value of the total score also depends on the length of the compared sequences, M and N. For example, in case of two different comparisons: one comparison of sequences with lengths of hundreds of pairs and one with sequences of length of tens of pairs. Even if the longer sequences contain some differences and the shorter sequences are completely equal, the comparison of the longer sequences would achieve a higher score. This renders the total score impractical as a metric for the direct comparison of the similarity of multiple process pairs.

In every comparison of two traces, there is a maximal and minimal total score that could be achieved. The maximal score $Score_{max}$, if both sequences are completely equal, would be as follows.

$$Score_{max} = \sigma_{equal} \cdot \max(M, N) \tag{4.1}$$

Using penalties of -1 for gaps and differences, with every difference or gap in the comparison, the total score decreases as an equal (positive) scoring pair is replaced by a difference/gap (negative) scoring pair. This results in a theoretical minimum score $Score_{theoretical\_min}$ that aligns both sequences with gaps.

$$Score_{theoretical\_min} = \sigma_{gap} \cdot (M + N) \tag{4.2}$$

However, this case would never occur in practice since the alignment algorithms try to maximize the score. Even if two sequences are totally different, the algorithm would always align some parts of the two sequences with differences instead of only using gaps, reducing the penalty from -2 for each two gaps to -1 for a difference. Therefore, to compute a functional minimum score, gaps are only inserted for missing events in the shorter trace. This results in a minimum score $Score_{min}$ that aligns both sequences with differences and necessary gaps in case of unequal sequence length.

$$Score_{min} = \sigma_{diff} \cdot \min(M, N) + \sigma_{gap} \cdot |M - N| \tag{4.3}$$

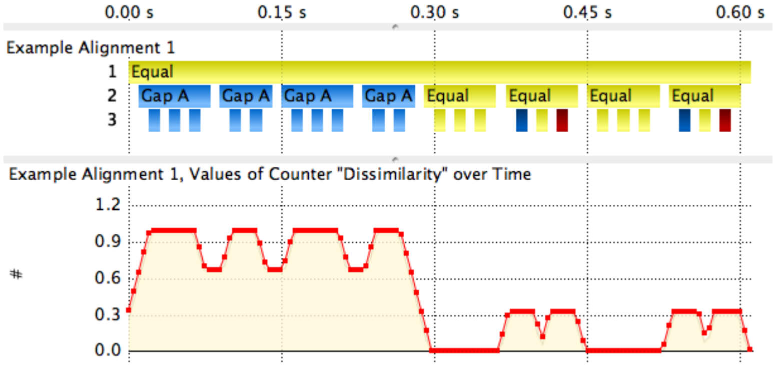


Figure 4.1: Dissimilarity timeline.

Using these minimum and maximum scores as a value range, it is possible to derive a similarity metric. The first step is to compute how close the actual total score of an alignment is to its maximum,

$$\mathit{Ratio} = \frac{Score_{actual}}{Score_{max}}; \mathit{Ratio} = [-0.5, 1]. \tag{4.4}$$

The above equation gives a ratio in a range between $[-0.5, 1]$. To define a more intuitive metric, the value is scaled to a range between $[0, 1]$. Thus, the metric *Similarity* is defined as

$$\mathit{Similarity} = \frac{\mathit{Ratio} + 0.5}{1.5}; Similarity = [0, 1]. \tag{4.5}$$

Similarity presents a means to objectively evaluate and compare the similarity of processes. $Similarity = 1$ means the processes are completely equal whereas $Similarity = 0$ means they are completely different.

4.2 Dissimilarity Timeline Metric

The goal for the *Dissimilarity Timeline* metric is to give an indication of how the similarity between two traces changes over time. This metric is useful for identifying regions of the trace that exhibit high dissimilarities for further inspection. Additionally, a visualization of the metric along with the traces can help pinpointing periodic differences in the traces.

To compute the metric, a computed alignment is sampled over time. This process produces a series of data values at equidistant time points indicating similarity for locations in the aligned traces. Once the alignment and similarity data values are computed, a sliding window technique is applied to sum up the score under the window. In the following case study a window width of 10% of the entire alignment length produces good results for all experiments. This width balances temporal granularity with the ability to capture context information around points with increased changes avoiding misleading “spikes of differences.” To sum up the values under the window several strategies are possible. For instance, one could sum up the total score of all pairs, or just add the score of all equal pairs. The latter would result in a metric representing the similarity of the sequences at the window position. Since the general goal is to compare relatively similar applications and to detect the dissimilarities, a slightly different approach is used. For the metric the score of all difference and gap pairs under the window is added up, which results in a metric representing the dissimilarity of the traces over time.

To make the *Dissimilarity Timeline* metric more useful and comparable between alignments, it is normalized to a value range of $[0, 1]$. The maximum possible dissimilarity of 1 occurs in the case that all pairs under the window are difference or gap areas. In the opposite case, all pairs under the window are equal, the metric has the value 0.

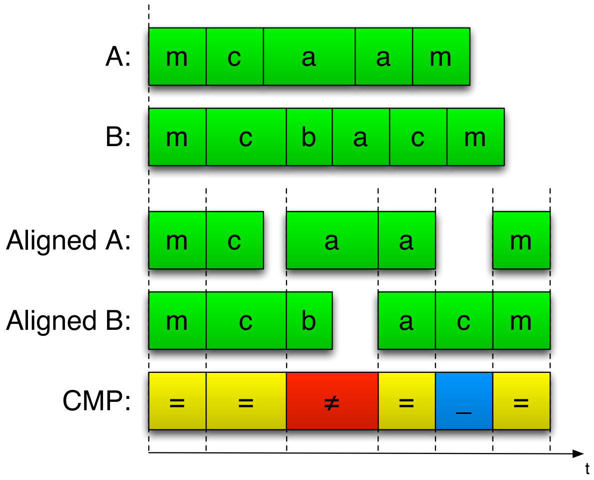


Figure 4.2: Artificial timing introduced by the alignment.

Figure 4.1 shows a *Dissimilarity Timeline* visualization for an example alignment. Sampled measurement points are depicted as red dots. The timeline visualization makes it easy to detect areas with high dissimilarity. For example, there are four *Gap A* areas (blue) that result in high dissimilarity at the beginning of the alignment shown at the top. However, the areas that are completely equal (all yellow) have a dissimilarity of 0. The dissimilarities are easy to pick out of this example alignment. In contrast, as demonstrated in Section 4.5, it is very challenging to pick out the dissimilarities of real application traces and alignments without the aid of the *Dissimilarity Timeline* metric.

4.3 Runtime Skew Timeline Metric

An important aspect of performance analysis is an understanding of the behavior of applications with respect to time. Event traces are especially useful for this purpose, because they retain the time-stamp of each event occurrence. However, it is tedious for users to gain this understanding manually. It involves attempting to line up iterations from traces and visually determine the timing differences between them. If the behavior of the application changes over time, then this process is even more complicated. The goal of the *Runtime Skew Timeline* metric is to aid the user in this process. The insight gained from this metric is valuable for analyzing performance differences across traces, including the benefit (or lack thereof) of changes to source code for performance improvements and for evaluating the effect of compiler options for performance optimization.

It is important to note that the alignment of processes introduces artificial virtual timings into the visualization of the traces. In other words, functions that appear to line up according to time in the visualization did not necessarily occur at the same time relative to the beginning of the execution. These artificial virtual timings arise because the alignment algorithm does not account for event timings, and only considers function names. For instance, the introduction of gap areas changes the apparent runtime of events. Also, aligning short running functions with long running functions alters the displayed runtime. In such cases, successive function calls are shifted back in time by the difference between the durations of the aligned functions. See Figure 4.2 for an example. Hence, looking at an arbitrary pair of functions does not allow one to draw conclusions about the real runtime behavior of the processes. This is another reason for developing the *Runtime Skew Timeline* metric, so that the user can understand the relative timing behavior across the aligned traces.

Figure 4.3 depicts a visualization of the *Runtime Skew Timeline* metric for a simple example alignment. In the alignment both processes are equal except for four functions only executed in process B. These are shown as gap areas in blue in the figure. These four extra functions delay process B by 20 ms

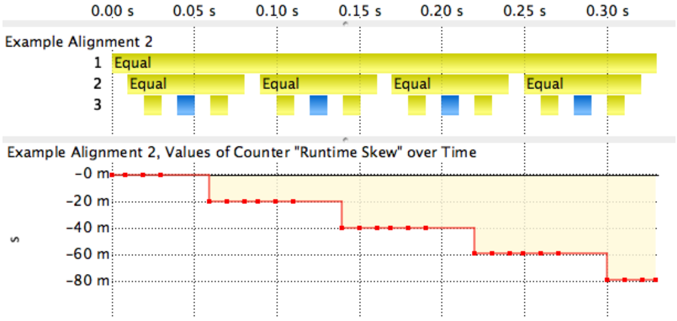


Figure 4.3: Runtime skew timeline.

per function. Since function events contain individual time stamps, it is possible to directly compare the invocation times of aligned function calls. The difference between these invocation times gives the runtime skew between both processes at the time of the function calls. For the simple example in Figure 4.3, the results are straightforward. The value for the metric increases with time, because process B is delayed by additional 20 ms in each iteration. However, in more complicated situations the timeline shows its real strengths and enables an easy detection of faster running code areas, possibly on both processes, or gradual or abrupt speed changes.

4.4 Function Time Difference Table

In addition to the *Runtime Skew Timeline* metric, the *Function Time Difference* table gives relative timing information. The time differences for all aligned invocations of each function in the traces are summed to generate the table. The result shows how much overall time was gained or lost within each function. Generally speaking, there is no guarantee that a particular function will exclusively either gain or lose time over the execution. It is likely that a particular function will show varying behavior over time, sometimes faster and sometimes slower compared to the aligned function in the other trace. This table clearly presents this information. For each function, it shows the number of times that it was faster (No. $+$) or slower (No. $-$) in the trace of process A than in process B. It also shows the overall time gained ($\Delta+$) or lost ($\Delta-$) for each function. An example of the *Function Time Difference* table is given in Table 4.2.

4.5 Case Studies

This section demonstrates the effectiveness of the introduced metrics by applying them in a series of case studies, covering BT from the NAS Parallel Benchmarks (NPB) and two real-world applications, AMG, a widely used Algebraic Multigrid solver, and ParaDiS, a dislocation dynamics code simulating crack propagation in materials.

Experiments have been conducted on two systems. The first is Sierra, a Linux cluster with 1944 dual-processor, Intel Xeon 5660 compute nodes. Each node has 24 GB RAM and a total of 12 cores each running at 2.8 GHz. The second system is Hera, a Linux cluster with 864 quad-processor, AMD Quad-Core Opteron nodes. The 13,824 cores run at 2.3 GHz and each node has 32 GB RAM. Intel compilers version 12.1 and MVAPICH2 as MPI library have been used on both systems.

The trace analysis and comparison prototype tool is built on top of the OTF trace library [75]. The library is used to capture the initial traces and then compare two respective OTF traces by applying alignment methods. The resulting differential trace is again written in OTF format using the OTF library and visualized using the trace visualizer Vampir [19]. The visualization of compared traces in Vampir required additional functionality that provides users with a visualization of the similarities, differences,

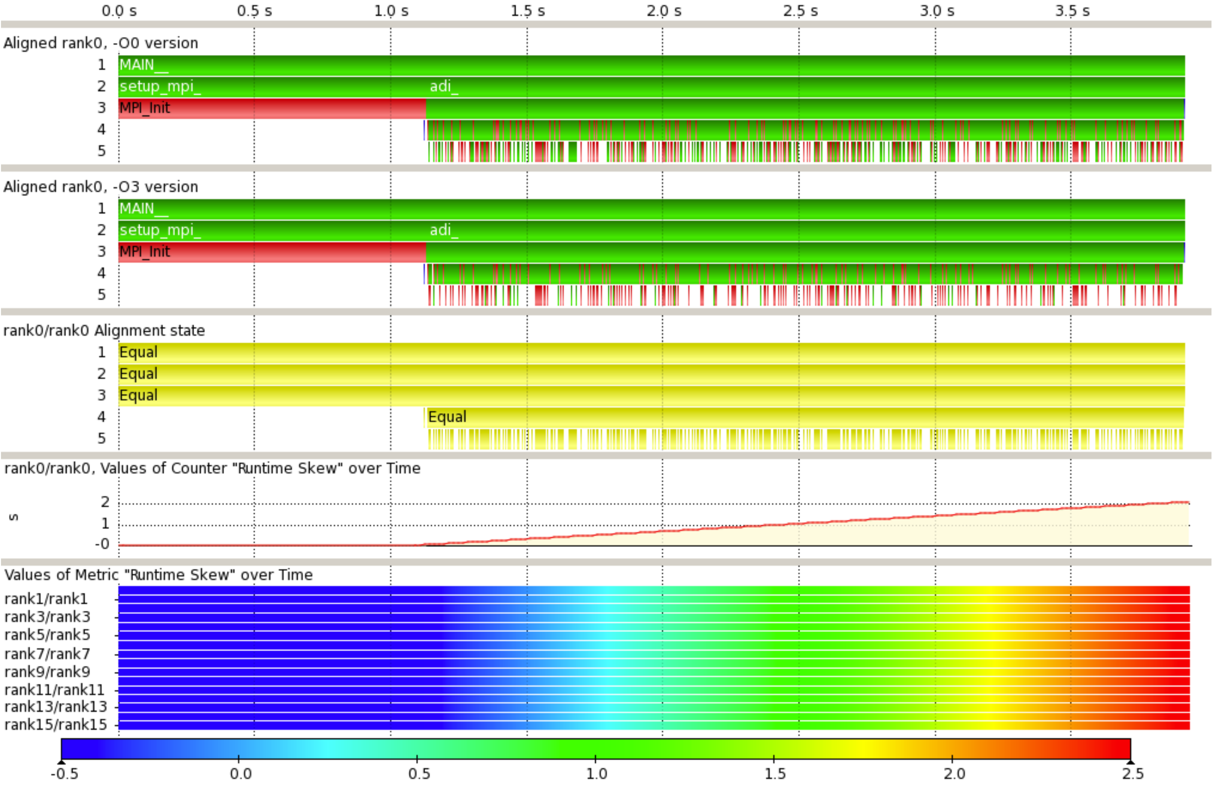


Figure 4.4: Comparison of two 16 process BT runs built with different compiler optimization levels (`O0` vs. `O3`).

and gaps across traces, as well as visualizations of the timing differences. Therefore, a hierarchical display of the trace differences based on the call tree has been implemented. This hierarchical display facilitates understanding of trace differences, because it simplifies the identification of the root of those differences. For example, if additional function calls are made in one trace and not the other, it is easy to locate the enclosing function to further investigate the changes. Additionally, the visualization differentiates the origin of gaps by introducing two gap states (*GAP A* and *GAP B*), which indicate if a gap occurs in trace A or B.

4.5.1 NAS Parallel Benchmark BT

The first case study is a comparison using BT from the NAS Parallel Benchmarks, version NPB-3.3.1-MPI [8]. The benchmark was run with 16 processes using Class W inputs. The first run was built with deactivated compiler optimization, i.e., using `O0`, while the second was built using the optimization level `O3`. The tests were conducted on the Linux cluster Hera.

Figure 4.4 shows the results of the comparison. The first two timelines show `rank 0` of the `O0` and the `O3` runs, respectively. The numbers on the left side indicate the call level, i.e., sub-functions are drawn one level below the calling function, e.g., `MAIN__` (level 1) calls `setup_mpi_` (level 2). The third timeline indicates the alignment result and shows that both processes are completely equal, which means the change in compiler optimization did not change the structure of the code. [1]

Although not shown, the similarity between the ranks is also reflected by the *Similarity Metric*. This metric value is 1 for all 16 comparisons. Hence, both runs are completely equal with respect to their function structure. The *Dissimilarity Timeline* would show the same result.

---

[1] In this case inlining may have been deactivated due to the attached tracing monitor for the measurement. This is due to the compiler's behavior when applying instrumentation for the tracing library and cannot be influenced by the user.

Because different compiler options were used, the runtime behavior is different for the runs. As expected, the optimized code (`O3`) is faster. Since artificial virtual timings are introduced for trace alignment, it is not directly visible that one run is faster than the other from looking at the aligned traces. Thus, the user can gain understanding of the performance differences in the aligned traces by looking at the values for the *Runtime Skew* metric, shown in the last two timelines in Figure 4.4. The top timeline gives the *Runtime Skew* for the comparison of `rank 0` for both executions and the bottom timeline gives the *Runtime Skew* for all rank-wise comparisons in a color-coded bar. Here, the blue/cold colors indicate low skew, while red/warm colors indicate higher skew. Initially, during the execution of `MPI_Init`, both code versions run with the same speed. The reason is that `MPI_Init` is a library call, and thus, is not affected by the compiler optimization level during the build phase of the application. After `MPI_Init`, the *Runtime Skew* metric shows that the optimized code version runs consistently faster and finishes about 2 seconds earlier than the non-optimized version. This behavior is the same across all processes.

4.5.2 AMG2006

AMG2006 [34] is a parallel algebraic multigrid solver for linear systems arising from problems on unstructured grids. In this case study the default version of AMG is compared with an optimized version that performs less coarsening. This results in more overall work, but avoids a lot of expensive communication that would have been necessary at the coarsest levels. Both compared versions solved a Laplace problem using 64 processes on 4 nodes. The measurements were conducted on the Linux cluster Hera.

Figure 4.5 shows the unaligned traces for `rank 0` from the executions of both versions of AMG. The optimized version runs faster and finishes about 1.25 seconds earlier than the original version. However, it is difficult to see the reasons for the performance differences from looking at the unaligned traces. With the help of the alignment analysis techniques, this task becomes straightforward. Figure 4.6 shows the comparison of the traces using the improved alignment visualization. The top and the middle timeline show the aligned traces for `rank 0` for both versions. The bottom timeline shows the alignment state between both traces. The optimized version saves considerable work in the initialization, identified by the blue gap area at 7.0 – 7.6 seconds in the beginning of the application run. Also, the optimized version saves work in each computation step, indicated by the repeating blue gap areas at 9.0 – 12.7 seconds starting at the middle of the application run.

These dissimilarities are the same for all 64 processes as shown by the *Dissimilarity Timeline* visualized as a set of color-coded bars in the top of Figure 4.7. Below the *Dissimilarity Timeline* visualization is the alignment state timeline from Figure 4.6 for easier comparison. Blue/cold colors mean processes are equal, while red/warm colors indicate differences. The differences agree with the alignment of the comparison of `rank 0`: the red area indicates the big gap in the initialization and the green areas from the middle to the end of the trace indicate the smaller gaps occurring at each compute iteration. The differences are almost the same throughout all processes, except for a few outliers indicated by horizontal green lines. The slightly higher dissimilarity in these processes is also highlighted using the *Similarity* metric (not shown). Most process pairs achieved similarity values between 0.60 – 0.63, while the few outliers only achieved similarity values between 0.46 – 0.49.

While the above metrics clearly show the structural differences and the overall change in performance, more detailed timing comparisons require the *Runtime Skew Timeline* metric, shown in the bottom two timelines in Figure 4.7. The upper graph depicts the runtime skew for the comparison of `rank 0` from each trace and the color-coded timeline depicts the runtime skew for all 64 process comparisons. The optimized version achieves a large speed gain in the initialization by avoiding unnecessary work. However, in the later stages of the initialization (after the blue gap), it performs slower than the original version. Yet, the speed gain in the beginning outweighs this slow down. During the iterations of the main body of the code, the optimized version performs faster again. In this case the speed gains are not consistent from iteration to iteration. The gains level off in the middle of the execution and rise again at the end. Additionally, the color-coded *Runtime Skew Timeline* shows that this behavior is consistent

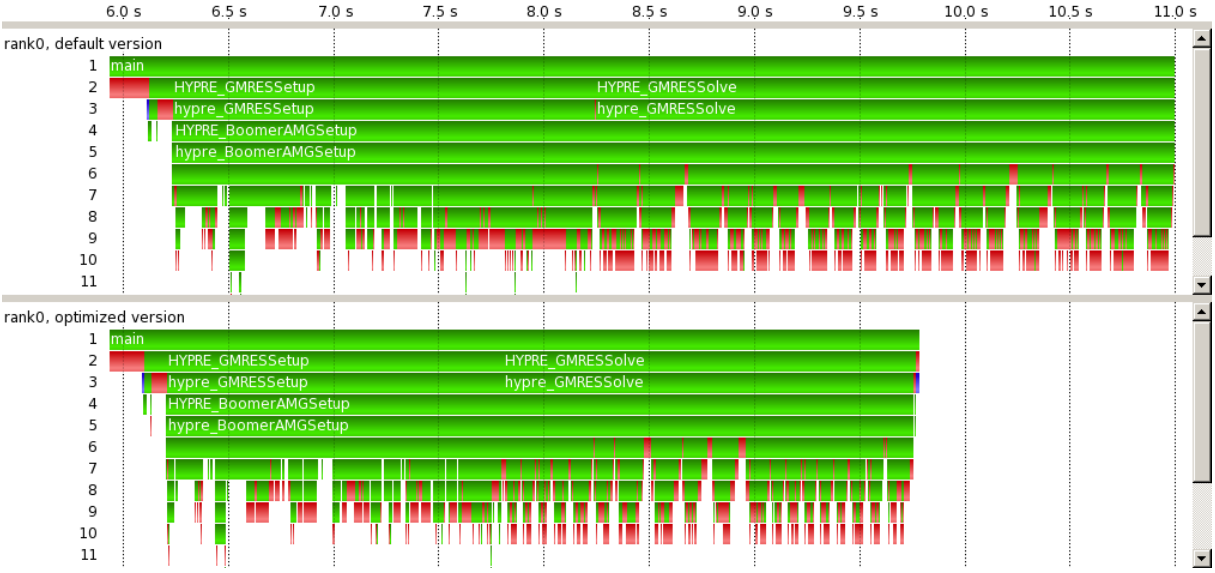


Figure 4.5: Unaligned rank 0 processes of the default and optimized AMG version.

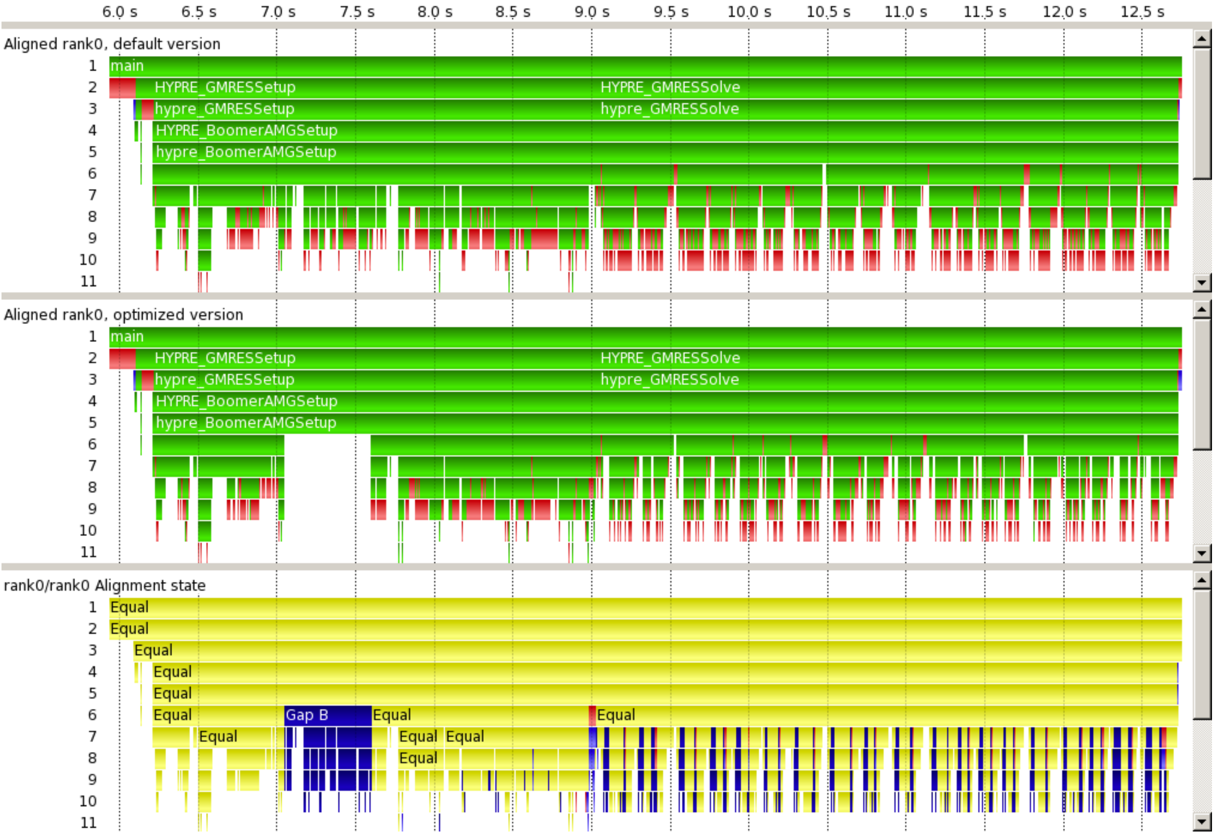


Figure 4.6: Aligned rank 0 processes of the default and optimized AMG version.

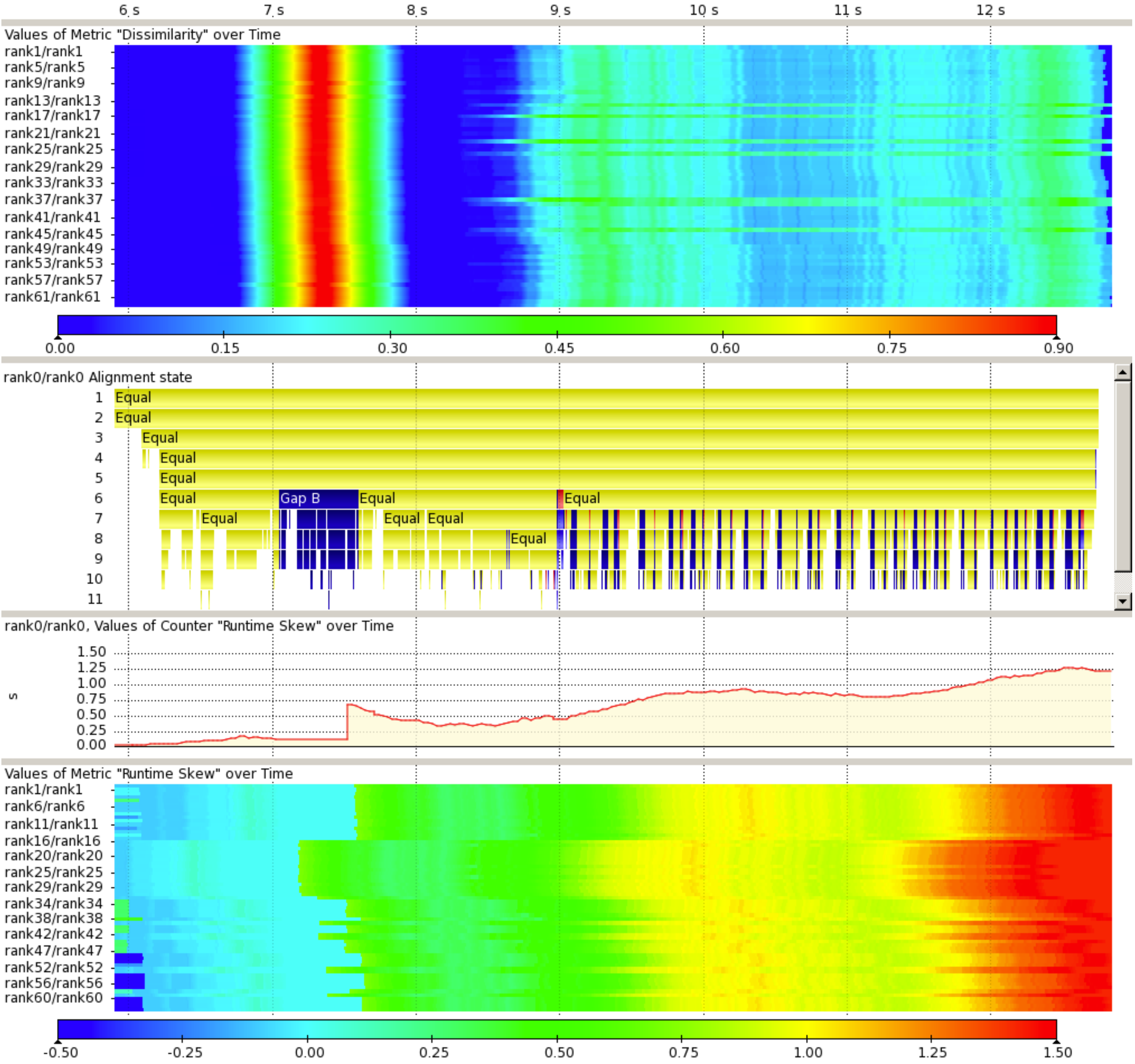


Figure 4.7: Similarity and runtime skew analysis between the default and optimized AMG version.

across all processes, but shows that performance on one node (second block of lines from the top) is slightly shifted. These observations would be very difficult to make without the support of the introduced metrics and visualizations.

4.5.3 ParaDiS

The Parallel Dislocation Simulator (ParaDiS) [7] models the dynamics of dislocation lines as they interact and move in response to the forces imposed by external stress and inter-dislocation interactions. The dislocations are divided into bounded regions called domains. Each process computes on one domain, with an adaptive load balancer periodically redistributing dislocations between domains.

This case study compares two versions of ParaDiS: v2.2.3 vs. v2.3.5.1. The release dates of the versions are about two years apart. Hence, version 2.3.5.1 includes bug fixes, improvements and corrections to the computations, as well as advanced load balancing. Both compared versions solved the same example problem, located in the code package: “tests/fmm_8cpu.ctrl”, using 8 processors. The measurements were conducted on the Linux cluster Sierra.

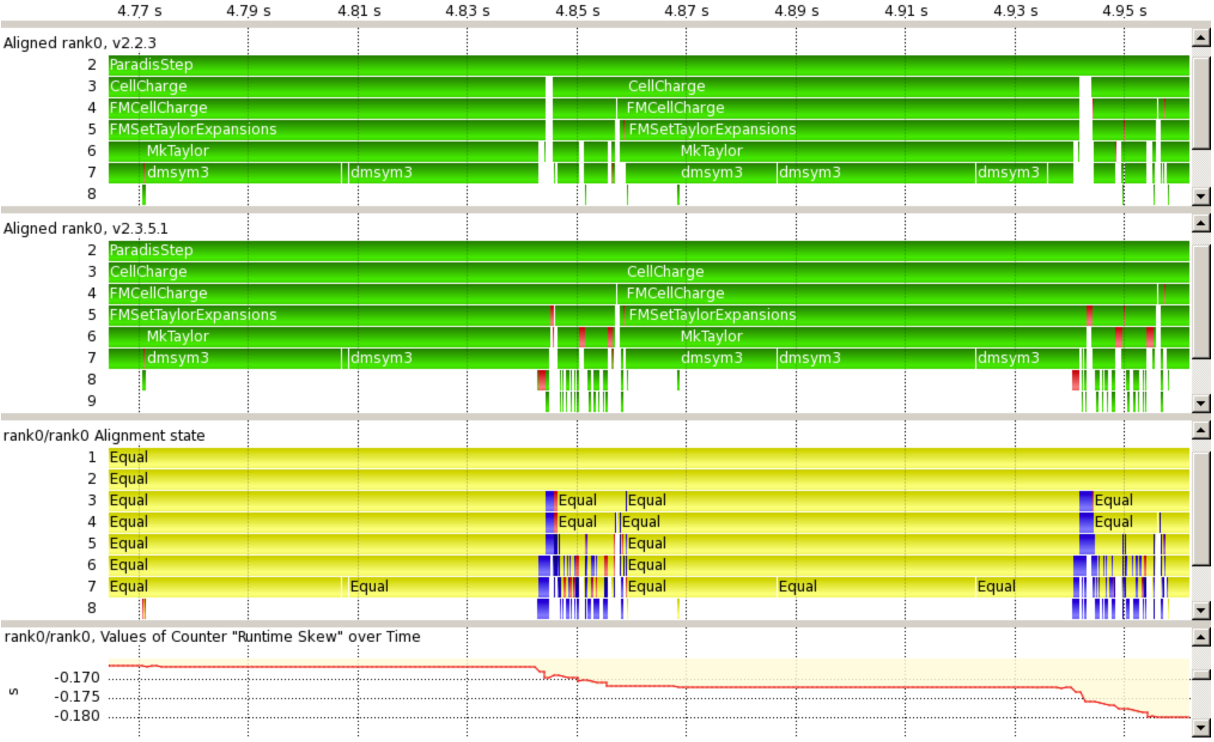


Figure 4.8: Comparison of two iterations of ParaDiS versions 2.2.3 and 2.3.5.1.

Figure 4.8 shows a detailed comparison of two representative iterations of both versions. In the beginning of the iterations, the executed function structure is the same. However, at the end of the iterations, blue gap areas in the bottom timeline indicate new functions. These functions can be seen in the direct comparison of the aligned timelines of both code versions (top and middle timelines in the figure). These new function calls are the result of added and extended functionality in version 2.3.5.1. As shown in the *Runtime Skew Timeline* at the bottom of Figure 4.8, this change comes at the cost of higher runtime. The runtime difference for the complete comparison is depicted in Figure 4.9. As in the case of BT, the runtime behavior is consistent across the whole execution. Version 2.3.5.1 runs consistently slower than the older version 2.2.3.

Added functionality is not the only cause of differences. The bottom timeline in Figure 4.9 shows that dissimilarity varies across the application. This variation is caused by the changes to the load balancer in version 2.3.5.1. In ParaDiS the executed function structure depends on the workload. The higher the load the more functions are executed in one iteration. Figure 4.9 also shows that process pairs for ranks 1 and 6 are more similar than the other compared processes. This is reflected in the *Similarity* metric as well, see Table 4.1 The comparisons of `rank 1` and `rank 6` achieve higher values than the other processes.

Table 4.2 shows the top of the *Function Time Difference* table for the comparison of `rank 0` from both ParaDiS versions. The function `MkTaylor` shows the most inconsistent behavior in terms of time consumption. This function, due to added functionality, runs slower in the new version for 114,230 invocations. The overhead time caused by this function adds up to 71.586ms. Yet, this function is not always slower. In 49,817 cases the execution of the new `MkTaylor` function was faster than its counterpart in the old version. The faster execution leads to a speed gain of 41.542ms. In normal profiles these times would have been evened out resulting in a single and potentially misleading reading of 30.044ms time lost in `MkTaylor` in the new version. While this particular result for `MkTaylor` most likely is caused by load distribution differences from the new load balancer, it is valuable performance information that can be obtained using the introduced alignment analysis techniques for trace comparisons.

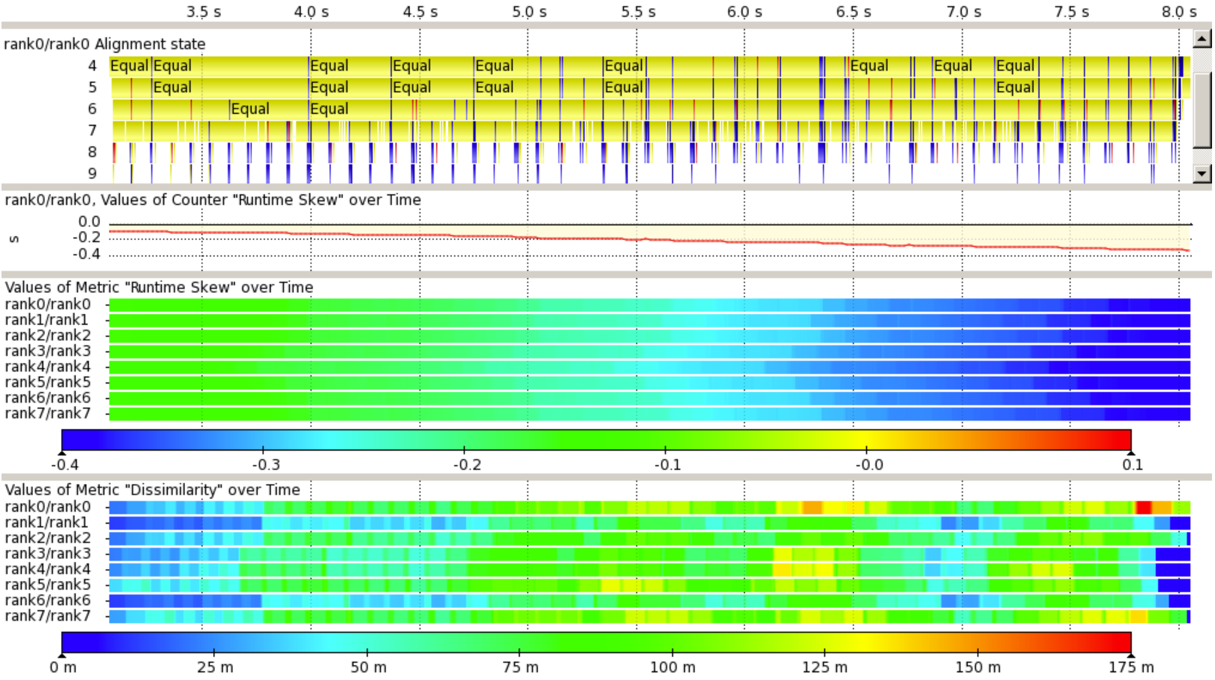


Figure 4.9: Similarity and runtime skew analysis between ParaDiS versions 2.2.3 and 2.3.5.1.

Table 4.1: Similarity between ParaDiS processes.

PROCESS PAIR	SIMILARITY METRIC
`rank 0 – rank 0`	0.450597
`rank 1 – rank 1`	0.561114
`rank 2 – rank 2`	0.471564
`rank 3 – rank 3`	0.485480
`rank 4 – rank 4`	0.475750
`rank 5 – rank 5`	0.456011
`rank 6 – rank 6`	0.546265
`rank 7 – rank 7`	0.436554

Table 4.2: Function time difference table for the rank0/rank0 comparison of ParaDiS.

FUNCTION NAME	NO. +	Δ +	NO. -	Δ -
`MkTaylor`	49817	$41ms\ 542\mu s$	114230	$71ms\ 586\mu s$
`ComputeForces`	315896	$21ms\ 938\mu s$	46280	$4ms\ 92\mu s$
`MPI_Waitall`	1035	$5ms\ 274\mu s$	1178	$3ms\ 517\mu s$
`LocalSegForces`	76952	$3ms\ 191\mu s$	63599	$8ms\ 446\mu s$
`dmsym3`	31685	$3ms\ 29\mu s$	51451	$2ms\ 636\mu s$
`FMSetTaylorExpa...`	121361	$71ms\ 545\mu s$	49217	$1ms\ 898\mu s$
`GetNeighborNode`	92813	$2ms\ 568\mu s$	35606	$1ms\ 814\mu s$
`HandleCollisions`	24886	$1ms\ 792\mu s$	26497	$2ms\ 13\mu s$
`MPI_Isend`	3420	$1ms\ 966\mu s$	3514	$1ms\ 744\mu s$
`GetNodeFromTag`	66112	$1ms\ 644\mu s$	32924	$1ms\ 484\mu s$
`SegSegForce`	770	$1ms\ 319\mu s$	93282	$20ms\ 88\mu s$

5 Methods for Structural Comparison of Multiple Processes

This chapter discusses techniques for the comparison of multiple processes. The analysis of multiple processes poses the challenge of analyzing large data volumes [58]. To cope with such large data sets, this chapter introduces a two step approach. First, a scalable method clusters structurally similar processes. Then, subsequent multiple sequence alignment methods perform a detailed comparison of grouped processes. The introduced alignment-based approach compares similar processes in detail and identifies their structural differences. The results of both structural comparison steps are valuable information that enables subsequent performance analysis methods.

5.1 Pre-Clustering Structurally Similar Processes

Clustering techniques divide data into groups (clusters). To facilitate user understanding, the identified groups should capture the natural structure of the data. This section describes a scalable clustering algorithm that reliably identifies groups of structurally similar processes from a large number of processes. The resulting groups can be used for information reduction and selection of processes for subsequent comparison analyses. For instance, hybrid applications that may use MPI processes, threads, and accelerators at the same time, pose the problem of comparing the performance of processes with different execution structure. Ignoring the structure and performing purely time-based analyses over all processes can complicate the identification of performance problems. As performance information of all processes is evaluated together, some processes may cover critical performance problems in others. For instance the mix of MPI processes with accelerator streams might result in hard to analyze performance data, due to their unrelated execution structure. Using structural pre-clustering allows to analyze distinct groups of processes independently. Reducing large numbers of processes to a few meaningful groups alleviates analysis effort for the user and provides a condensed overview of an application's execution.

A good criterion for structural comparison of processes is information on the invocation of functions, disregarding timing. This information is available in almost any type of performance profile. Information on function invocation can be represented in *call trees*. These trees represent merged call stacks of all functions that a process invokes. Differences between the call trees of two processes provide a measure of similarity. Comparing call trees, however, is computationally expensive and small differences close to the root of a call tree yield low similarity, even if large parts of the trees are equal.

Thus, this work uses a simplification of the information contained in a call tree, the so-called *function pairs*, made up out of the caller/callee relation in the call tree. The function pairs are a set of pairs, where for each pair the first function calls the second one. The number of function pairs is independent of application runtime or the process count and is limited to the squared number of existing functions. For many applications, the number of function pairs is less than 1,000, which makes them a powerful approach for comparison. Based on the function pairs, this work introduces a similarity measure for comparing processes.

Scalable structural process clustering not only requires a similarity measure based on function pairs, but also requires a scheme that is efficient even for large process counts. Since a pair-wise comparison of all processes is highly inefficient, this work introduces a fast hash-based clustering approach to store and compute groups of similar processes. This approach removes the need for pair-wise process comparison.

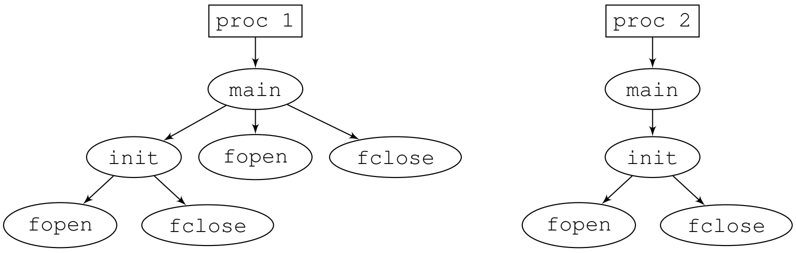


Figure 5.1: Call trees of two example processes.

5.1.1 Determining Similarity

All clustering methods require a similarity or distance metric as a basis for building clusters. This section outlines such definitions of similarity for the purpose of process clustering. It begins with general definitions followed by the development of a similarity measure for the structure of processes.

Structural Similarity of Processes

The structural information for a process is determined by examining the functions that the process invokes during its execution. This information allows to group processes according to their structure. To extract the structural information and perform the grouping, the following definitions are used. Let $\mathcal{P}$ denote the set of all processes and let individual processes be labeled $P, P_1, P_2, P_3, \ldots$. Additionally, for illustrative purposes, names like `process 1` and `thread 3` refer to individual processes in $\mathcal{P}$. Let $\mathcal{F}$ denote the set of the functions that any process invokes. Individual functions are labeled as $F, F_1, F_2, F_3, \ldots$ or for illustrative purposes with names like `main` and `fopen`. 2^A refers to the power set of a set A, which is the set of all subsets of A.

The mapping $\mathit{funcs} : \mathcal{P} \rightarrow 2^{\mathcal{F}}$ assigns each process the set of functions that it invokes during an execution. Formally: $\mathit{funcs}(P) := \{F \in \mathcal{F} \mid F \text{ is called at least once by } P\}$. Processes that invoke many common functions are considered as structurally similar. Using the Jaccard index [66], the structural similarity of two processes is expressed as the number of functions that they have in common, divided by the number of functions that either process calls:

$$\mathit{funcsim}(P_1, P_2) := \frac{|\,\mathit{funcs}(P_1) \cap \mathit{funcs}(P_2)\,|}{|\,\mathit{funcs}(P_1) \cup \mathit{funcs}(P_2)\,|} \tag{5.1}$$

If both sets are empty, i.e., neither process calls a function, the result is defined to be 1. Consequently, $funcsim$ maps a pair of processes to a real number between 0 and 1. 1 expresses total similarity and 0 expresses complete dissimilarity. Therefore, *funcsim* presents a measure for the structural similarity between two processes. Information on function invocation can be represented in call trees. As an example, Figure 5.1 illustrates call trees of two processes. In the example, the sets of invoked functions are identical, i.e.:

$$\begin{aligned}\mathit{funcs}(\texttt{proc 1}) &= \mathit{funcs}(\texttt{proc 2})\\ &= \{\texttt{main, init, fopen, fclose}\}\end{aligned}$$

Consequently, similarity according to *funcsim* is 1 for the example shown in Figure 5.1.

Structural Similarity Measure

Given the high degree of difference between the two call trees in Figure 5.1, a similarity measure, like *funcsim* (Equation 5.1), that indicates exact similarity between them is undesirable. Improvement options for the similarity measure include consideration of supplied arguments, global state, and possibly even external influences, such as incoming messages. Incorporating these factors is hard due to the sheer size of this influential state space. A suitable way to refine the measure is to use pairs of functions that represent the *caller-callee* relation. Therefore, the function *funcs* is extended with the function *pairs* $: \mathcal{P} \rightarrow 2^{\mathcal{F} \times \mathcal{F}}$. Formally: $\textit{pairs}(P) := \{(F_1, F_2) \in \mathcal{F} \times \mathcal{F} \mid F_1 \text{ calls } F_2 \text{ on } P\}$. The key advantages of considering the caller-callee relationship are that it retains structural information in the form of functions and their context in the form of the relation. As this refinement still relies on a set-based representation, it allows to directly retrieve a similarity measure based on caller-callee pairs:

$$\textit{pairsim}(P_1, P_2) := \frac{\mid \textit{pairs}(P_1) \cap \textit{pairs}(P_2) \mid}{\mid \textit{pairs}(P_1) \cup \textit{pairs}(P_2) \mid} \tag{5.2}$$

In the following, $F_1 \rightarrow F_2$ is used to denote a caller-callee pair (F_1, F_2). Functions that are a source in the call stack, such as `main`, have no caller. Such functions use a virtual root function called ϵ to include these source functions in *pairs*, i.e., $\epsilon \rightarrow \texttt{main} \in \textit{pairs}(\texttt{proc 1})$.

Continuing the example in Figure 5.1, the function pair sets *pairs* of the two processes exactly contain the directed edges of their corresponding call trees, i.e.:

$$\begin{aligned}
\textit{pairs}(\texttt{proc 1}) = \{&\epsilon \rightarrow \texttt{main},\ \texttt{main} \rightarrow \texttt{init},\\
&\texttt{main} \rightarrow \texttt{fopen},\ \texttt{main} \rightarrow \texttt{fclose},\\
&\texttt{init} \rightarrow \texttt{fopen},\ \texttt{init} \rightarrow \texttt{fclose}\}\\
\textit{pairs}(\texttt{proc 2}) = \{&\epsilon \rightarrow \texttt{main},\ \texttt{main} \rightarrow \texttt{init},\\
&\texttt{init} \rightarrow \texttt{fopen},\ \texttt{init} \rightarrow \texttt{fclose}\}
\end{aligned}$$

Based on the results of *pairs* it is possible to directly compute the similarity between processes 1 and 2, which is $\textit{pairsim}(\texttt{proc 1}, \texttt{proc 2}) = \frac{4}{6}$. Processes that have a similarity of 1 form groups of *similar processes*.

The design of this metric bases on the expectation that: first, a single execution will exhibit a reasonably low number of function pairs; and second, that the number of function pairs remains about constant when application scale increases. This expectation results from the fact that a wide range of HPC applications rely on a single executable that is executed by multiple threads, processes, or GPGPU devices. Consequently, the variation in the structural behavior observed within a single execution or between multiple executions is limited by the statically compiled executable. Existing approaches [3, 6] for parallel call stacks support this expectation.

Building the Similarity Matrix

Common clustering approaches require a *similarity matrix* holding the distances between objects. The structural similarity metric *pairsim* (Equation 5.2) represents such a distance measure for processes. This metric states the similarity or distance between processes. To calculate all distances of a similarity matrix for n processes, $0.5\, n\, (n-1)$ pairwise comparisons are needed [57]. The resulting similarity matrix M is a symmetric matrix. The entries on the main diagonal are filled with 1, as each process is completely similar to itself. The upper triangle can be mirrored into the lower triangle since the calculation of the similarity is a commutative operation: $S_{n,m} = S_{m,n}$, with n and m being the two subscript indices for a matrix S.

$$M = \begin{array}{r|cccccc|}
 & \text{Proc 1} & \text{Proc 2} & \text{Proc 3} & \text{Proc 4} & \cdots & \text{Proc n} \\
\text{Proc 1} & 1 & S_{1,2} & S_{1,3} & S_{1,4} & \cdots & S_{1,n} \\
\text{Proc 2} & S_{1,2} & 1 & S_{2,3} & S_{2,4} & \cdots & S_{2,n} \\
\text{Proc 3} & S_{1,3} & S_{2,3} & 1 & S_{3,4} & \cdots & S_{3,n} \\
\text{Proc 4} & S_{1,4} & S_{2,4} & S_{3,4} & 1 & \cdots & S_{4,n} \\
\vdots & \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\
\text{Proc n} & S_{1,n} & S_{2,n} & S_{3,n} & S_{4,n} & \cdots & 1
\end{array} \tag{5.3}$$

Each row in the matrix represents the data point for the respective process. This data set is suitable as input for a number of common clustering algorithms. The matrix spans an n-dimensional space for n processes.

5.1.2 Grouping Structurally Similar Processes

This section first discusses two common clustering algorithms and then introduces a new approach for partitional clustering. In partitional clustering the data set is divided into non-overlapping subsets (clusters). Each data point is assigned to exactly one subset. Finally, this section introduces a method of arranging the clustered process groups hierarchically with respect to their structural similarity.

Common Clustering Algorithms

The following describes two widely used clustering methods. Their advantages and disadvantages for the clustering of processes are discussed.

K-Means, K-Medoids *K-Means* [51,86,88] and *K-Medoids* [71,72] are the most prominent partition algorithms. They partition the data set into a number of Voronoi cells. The number of the cells (clusters) k needs to be known a priori. To guide the choice of k the *silhouette coefficient* [123] allows to measure cluster quality and compare different clusterings for various numbers of k.

Algorithm 3 shows the pseudo-code of the K-Medoids algorithm. In contrast to the K-Means algorithm, K-Medoids chooses data points as centers and works with an arbitrary matrix of distances between data points. It is more robust to noise and outliers as K-Means, because it minimizes a sum of pairwise dissimilarities instead of a sum of squared Euclidean distances.

Both algorithms have a tendency to produce clusters of the same size, which might not be appropriate for all data sets. Additionally, the positions of the initial start clusters influence the clustering result. The requirement of defining the number of clusters a priori impedes the search for the natural layout of the underlying data set. To find the best number of clusters, the analyst needs to perform and compare a possibly large number of different clusterings.

DBSCAN The *density-based spatial clustering of applications with noise* (DBSCAN) [32] algorithm partitions the data based on the density between the data points. High-density areas build clusters of arbitrary shape. Consequently, the algorithm automatically determines the number of clusters. Additionally, DBSCAN provides a notion of noise. Data points in low-density areas are classified as noise and omitted from the clustering. Different densities in one data set cannot be handled well by DBSCAN. Algorithm 4 gives the pseudo-code of the DBSCAN algorithm.

The algorithm has the advantage of automatically identifying the natural number of clusters. However, it has difficulties capturing clusters with different densities. For the algorithm, the user needs to define the maximum distance between points eps. Depending on the data set and situation, small changes of this distance can result in completely different clustering results.

It is possible to use both algorithms for the clustering of processes. In fact, K-Means/K-Medoids as well as DBSCAN already have been used for similar purposes [36, 43]. However, the results of both algorithms are very sensitive to the chosen input parameters. This makes it hard to acceptably quantify the quality of a clustering result for subsequent comparison steps. Also, both algorithms require a similarity matrix. Like shown in Section 5.1.1 the calculation of a similarity matrix for large numbers of processes quickly hits computational limits. Especially if the employed similarity measure is a computationally expensive operation. To alleviate these disadvantages, the next section introduces an alternative hash-based clustering method.

Algorithm 3: K-Medoids clustering algorithm

```
Input: Number of clusters $k$, set of processes $D$.
Output: A set of $k$ clusters $C$.
foreach $C_i \in C$
    Initialize cluster medoid $\mu_{C_i}$ with an unused process from $D$.
repeat
    Classify the remaining $|D| - k$ processes. Each process $p \in D$ is put in the cluster
      containing the most similar cluster medoid.
    foreach $C_i \in C$
        Recompute cluster medoid $\mu_{C_i}$.
until there is no change in medoids
return $C = C_0, C_1, \ldots, C_{k-1}$
```

Algorithm 4: DBSCAN clustering algorithm

```
Input: Set of processes $D$, maximum dissimilarity between processes $eps$, minimum number of
       processes in one cluster $minPs$.
Output: Process clusters $C$.
Function DBSCAN(D, eps, minPs)
    $C = 0$
    foreach unvisited process $P$ in $D$
        mark $P$ as visited
        $neighborPs$ = regionQuery(P, eps)
        if sizeof(neighborPs) < minPs-1
            mark $P$ as NOISE
        else
            $C$ = next cluster
            expandCluster(P, neighborPs, C, eps, minPs)
    return $C$
Function expandCluster(P, neighborPs, C, eps, minPs)
    add $P$ to cluster $C$
    foreach point P' in neighborPs
        if P' is not visited
            mark $P'$ as visited
            $neighborPs'$ = regionQuery(P', eps)
            if sizeof(neighborPs') >= minPs-1
                $neighborPs$ = $neighborPs$ joined with $neighborPs'$
        if P' is not yet member of any cluster
            add $P'$ to cluster $C$
Function regionQuery(P, eps)
    return all processes within $P'$s eps-neighborhood
```

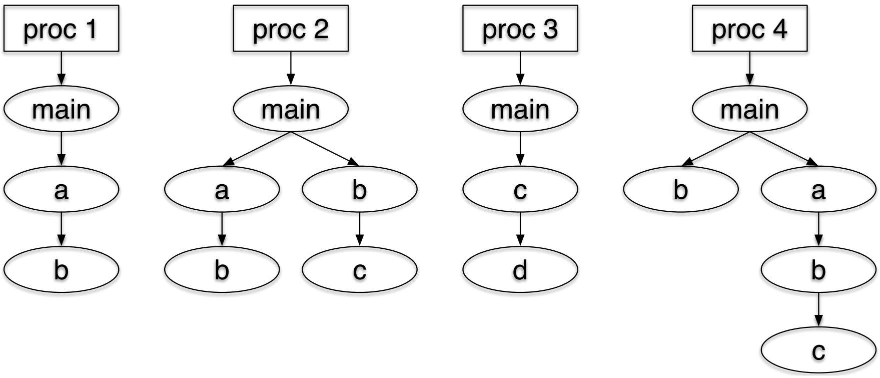


Figure 5.2: Call trees of four example processes.

Fast Hash-Based Clustering

Efficient clustering of processes requires a method that does not need the computation of a similarity matrix. Meaningful clustering results that do not depend on input parameters also present easier to evaluate results.

This approach works with a cryptographic hash function to scalably compute natural clusters of structurally similar processes. A cryptographic hash function computes a fixed size bit string (hash) from an arbitrary sized input. The same input data produces equal output hashes, whereas even small changes in the input data cause substantial changes in the output hash. The design of cryptographic hash functions renders the search for two different input data sets that produce the same output hash infeasible. Such a case, a so-called *hash collision*, is theoretically possible, but should never occur in practice. Additionally, an important design criterion of cryptographic hash functions is the capability to compute the output hash as fast as possible. These properties allow a fast and memory efficient grouping of processes. The cryptographic hash function employed in this work is *SHA-3* [12], the newest member of the Secure Hash Algorithm (SHA) family. The chosen version is SHA3-512.

The goal of the hash-based clustering method is, that the hash function produces equal hashes for structurally similar processes. The output hash of the hash function is then used as key in a map structure that holds all processes. As similar processes share the same key, the map automatically groups similar processes.

Algorithm 5 lists the pseudo-code of the hash-based clustering algorithm. The following example demonstrates the algorithm by clustering the processes shown in Figure 5.2.

The similarity definition introduced in Section 5.1.1 describes the structural similarity of processes. In order for the hash approach to work, it needs to be guaranteed, that similar processes produce the same hash input. This is achieved in three steps:

Extraction of Structural Data Applying the definition *pairs*, introduced in Section 5.1.1, the example processes contain the following structural information:

$$
\begin{aligned}
&\mathit{pairs}(\texttt{proc 1}) = \{\epsilon \rightarrow \texttt{main},\ \texttt{main} \rightarrow \texttt{a},\ \texttt{a} \rightarrow \texttt{b}\}\\
&\mathit{pairs}(\texttt{proc 2}) = \{\epsilon \rightarrow \texttt{main},\ \texttt{main} \rightarrow \texttt{a},\ \texttt{a} \rightarrow \texttt{b},\ \texttt{main} \rightarrow \texttt{b},\ \texttt{b} \rightarrow \texttt{c}\}\\
&\mathit{pairs}(\texttt{proc 3}) = \{\epsilon \rightarrow \texttt{main},\ \texttt{main} \rightarrow \texttt{c},\ \texttt{c} \rightarrow \texttt{d}\}\\
&\mathit{pairs}(\texttt{proc 4}) = \{\epsilon \rightarrow \texttt{main},\ \texttt{main} \rightarrow \texttt{b},\ \texttt{main} \rightarrow \texttt{a},\ \texttt{a} \rightarrow \texttt{b},\ \texttt{b} \rightarrow \texttt{c}\}
\end{aligned}
$$

Algorithm 5: Hash clustering algorithm

```
Input: Set of processes $D$.
Output: Process clusters $C$.
$C = 0$
foreach process $P$ in $D$
    extract structural data $P_{struct}$ from $P$
    get sorted structural data $P_{struct-sorted}$ from $P_{struct}$
    get serialized structural data $P_{struct-string}$ from $P_{struct-sorted}$
    compute hash $P_{hash}$ from $P_{struct-string}$
    insert $P_{hash}$ and $P$ into $C$
        if $C$ already contains $P_{hash}$
            append $P$ to list of processes already associated with key $P_{hash}$
        else
            add key $P_{hash}$ to $C$ and associate $P$ with it
return $C$
```

Sorting of Structural Data The extracted function pairs are sorted, using an algorithm with $\mathcal{O}(n\ log(n))$ complexity, to ensure the same input data for similar processes. The hash-based clustering method arranges all pairs in alphabetical order. The sorted structural information of the example processes looks as follows:

$$\textit{sorted-pairs}(\texttt{proc 1}) = \{\texttt{a} \rightarrow \texttt{b},\ \texttt{main} \rightarrow \texttt{a},\ \epsilon \rightarrow \texttt{main}\}$$
$$\textit{sorted-pairs}(\texttt{proc 2}) = \{\texttt{a} \rightarrow \texttt{b},\ \texttt{b} \rightarrow \texttt{c},\ \texttt{main} \rightarrow \texttt{a},\ \texttt{main} \rightarrow \texttt{b},\ \epsilon \rightarrow \texttt{main}\}$$
$$\textit{sorted-pairs}(\texttt{proc 3}) = \{\texttt{c} \rightarrow \texttt{d},\ \texttt{main} \rightarrow \texttt{c},\ \epsilon \rightarrow \texttt{main}\}$$
$$\textit{sorted-pairs}(\texttt{proc 4}) = \{\texttt{a} \rightarrow \texttt{b},\ \texttt{b} \rightarrow \texttt{c},\ \texttt{main} \rightarrow \texttt{a},\ \texttt{main} \rightarrow \texttt{b},\ \epsilon \rightarrow \texttt{main}\}$$

Serialization of Structural Data The cryptographic hash function requires one input stream. Therefore it is necessary to serialize the sorted function pairs into one input string. For this purpose, the algorithm iterates over the list of sorted function pairs and prints this information to a text string. To mark the caller-callee relationship, the algorithm inserts the symbol "→". This symbol must not occur in the structural information of the example processes. The input strings for the example processes are shown below:

$$\texttt{string proc 1}: \text{“}\texttt{a} \rightarrow \texttt{b main} \rightarrow \texttt{a}\ \epsilon \rightarrow \texttt{main}\text{”}$$
$$\texttt{string proc 2}: \text{“}\texttt{a} \rightarrow \texttt{b b} \rightarrow \texttt{c main} \rightarrow \texttt{a main} \rightarrow \texttt{b}\ \epsilon \rightarrow \texttt{main}\text{”}$$
$$\texttt{string proc 3}: \text{“}\texttt{c} \rightarrow \texttt{d main} \rightarrow \texttt{c}\ \epsilon \rightarrow \texttt{main}\text{”}$$
$$\texttt{string proc 4}: \text{“}\texttt{a} \rightarrow \texttt{b b} \rightarrow \texttt{c main} \rightarrow \texttt{a main} \rightarrow \texttt{b}\ \epsilon \rightarrow \texttt{main}\text{”}$$

The serialized data stream presents a suitable input for the cryptographic hash function. Equal input streams result in the same hash value. Indicated below is the generation of hash values for the example processes.[1]

$$\textit{hash}(\texttt{string proc 1}) = 1ca6371c$$
$$\textit{hash}(\texttt{string proc 2}) = a6207684$$
$$\textit{hash}(\texttt{string proc 3}) = 98bc7d73$$
$$\textit{hash}(\texttt{string proc 4}) = a6207684$$

---

[1] The shown hash values have only illustrative purposes and are not real computed values from the SHA-3 algorithm.

	$\epsilon \to F_1$	$F_1 \to F_2$	$F_1 \to F_3$
P_1	×	×	
P_2	×		×
P_3	×	×	×
P_4	×		×

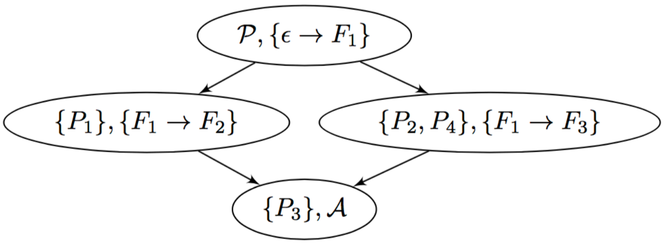

$\mathcal{P}, \{\epsilon \to F_1\}$
$\{P_1\}, \{F_1 \to F_2\}$
$\{P_2, P_4\}, \{F_1 \to F_3\}$
$\{P_3\}, \mathcal{A}$

Figure 5.3: Incidence relation table (left) and resulting concept lattice (right) for four example processes.

The computed hash value builds the basis for the final clustering step. Similar processes are clustered automatically by using the hash value as key for a map structure. Processes are stored in the map according to their hash key. Table 5.1 illustrates this step for the example processes. `proc 2` and `proc 4` are clustered together due to their equal hash value. `proc 1` and `proc 3` stay in own clusters as they are structurally different.

Table 5.1: Hash map with the clustering result for the example processes.

HASH KEY	PROCESSES
1ca6371c	`proc 1`
a6207684	`proc 2`, `proc 4`
98bc7d73	`proc 3`

Advantages of this approach are the very fast computation of the hash function and that processes can be added sequentially to the clustering map. The algorithm requires no direct comparison of structural information between processes. This prevents the computation of a similarity matrix and drastically reduces memory requirements. The approach groups equal processes, according to the similarity definition in Section 5.1.1. This produces easily understandable clustering results. The computed clusters contain highly structural similar processes. The grouping result is suitable as starting point for the successive analysis methods introduced in Section 5.2.2. How to extract relations between found clusters and arrange them in a hierarchical fashion is described in the next section.

Hierarchical Arrangement of Process Clusters

This section briefly explains the structural clustering approach introduced in the paper *Structural Clustering: A New Approach to Support Performance Analysis at Scale* [141]. More detailed descriptions along with a performance evaluation and an applicability study of the method can be found in the paper.

Section 5.1.1 introduces set-based structural similarity definitions that allow to use *concept lattices* in order to store and hierarchically arrange process clusters. A concept lattice is based on a *formal context*, which is a triple (O, A, I), where O is a set of objects, A a set of attributes, and $I \subseteq O \times A$ an incidence relation. The incidence relation associates each object with a set of attributes. For process clustering, the processes represent the set of objects and the function pairs represent the attributes. A formal context defines a *concept lattice* by specifying clusters, and a partial order on them. A concept lattice can be represented as a directed acyclic graph where clusters are nodes and the order on them determines the edges. A concept lattice has the property that each object (process) and attribute (function pair) is contained in the concept lattice exactly once. Consequently, the nodes of the lattice provide a process clustering, since the above property guarantees that each process belongs to exactly one cluster.

Figure 5.3 illustrates this formalism using an example. The incidence relation can also be described as a table (left side in Figure 5.3). In the example, every process has the function pair $\epsilon \to F_1$, i.e., processes share F_1 as their main function. Processes P_1 and P_3 have the additional pair $F_1 \to F_2$, i.e., for these processes F_1 calls F_2. Finally, P_2, P_3 and P_4 also have the pair $F_1 \to F_3$, i.e., for these processes F_1 calls F_3. The right side of Figure 5.3 illustrates the resulting concept lattice. It reads as:

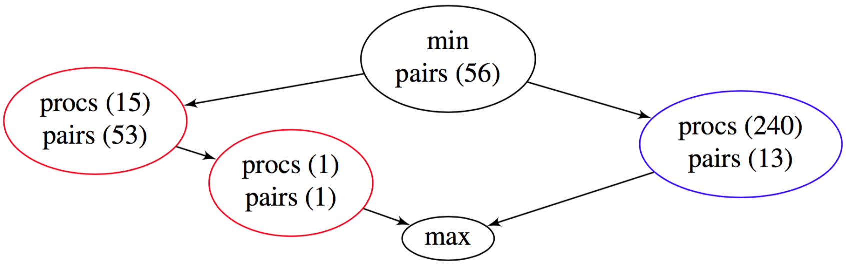


Figure 5.4: Concept lattice for an execution of the BT kernel with 16 MPI processes (red) and 15 OpenMP threads (blue) per process.

- Process sets subsume those that are reachable following edges downwards;
- Function sets subsume those that are reachable following edges upwards;
- The top node indicates that all processes share the function pair $\epsilon \to F_1$;
- The bottom node signifies that only P_3 has all function pairs, and in particular that it includes all function pairs that P_1, P_2 and P_4 exhibit;
- P_1 is different from P_2 and P_4; and
- P_2 and P_4 have the same function pairs.

Figure 5.4 shows the concept lattice for an execution of the real-world application BT [8] with 16 MPI processes, which use 15 OpenMP threads each. Every node contains a set of processes and a set of function pairs. Since the names of individual processes or their function pairs are most accessible to developers of the target application, only the number of processes or function pairs belonging to a node are specified, e.g., `procs` (15) for a set of 15 processes. Even with 256 overall processes—MPI processes and OpenMP threads—-the concept lattice for this example remains small. All processes share 56 common function pairs. OpenMP threads (right) exhibit different structural behavior than MPI processes (left). All MPI processes have the same function pairs, except for process 0, which invokes one additional pair.

The result of using the concept lattice is a classification of all processes into groups with structurally identical behavior, along with a relation between these groups. In practice, this method generates a low number of groups for many application types and therefore provides a compact representation for structural information that can be computed scalably for large-scale executions.

Concluding, both the hash-based and concept lattice approach compute the same clustering result. Additionally to the clustering, the concept lattice approach adds a hierarchical order to the grouping results.

5.2 Alignment of Clustered Processes

Processes grouped in a cluster can still exhibit structural differences. The reason is that the underlying data used to compute the similarity metric—the call tree representing all call relations occurring during a process's execution—only contains an aggregated representation of the structural information. Consequently, the detailed execution of function calls during runtime can still differ between grouped processes. In order to compare the actually executed function call structures between processes, multiple sequence alignment (MSA) methods are employed. Like shown in the previous two chapters, alignment methods are capable of identifying detailed structural differences between processes. MSA methods perform such comparisons between multiple processes. The following sections introduce this comparison approach. The MSA methods are applied as an additional analysis step on top of the clustering analyses described in the previous section.

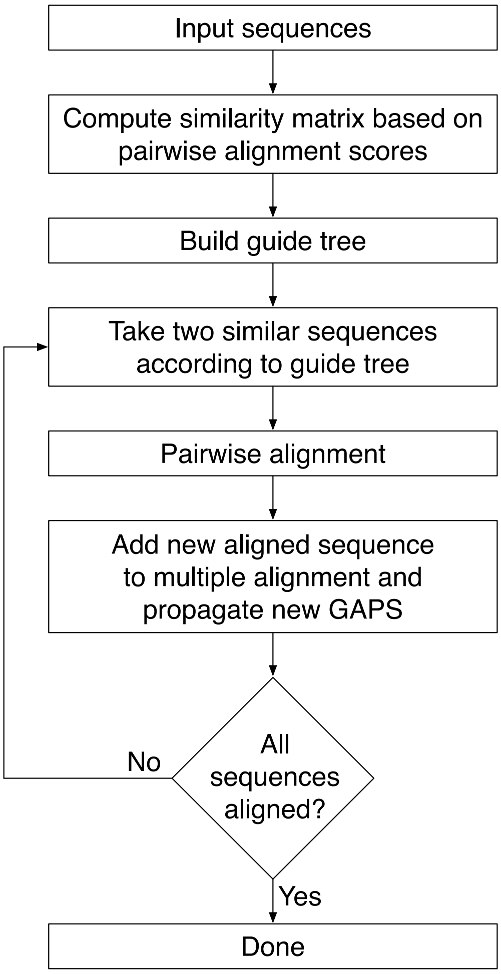

Input sequences
Compute similarity matrix based on pairwise alignment scores
Build guide tree
Take two similar sequences according to guide tree
Pairwise alignment
Add new aligned sequence to multiple alignment and propagate new GAPS
All sequences aligned?
No
Yes
Done

Figure 5.5: The progressive alignment scheme.

5.2.1 Multiple Sequence Alignment

Multiple sequence alignment (MSA) methods are capable of aligning more than two sequences. In computational biology MSA is used to align RNA, DNA, or protein sequences with the purpose of analyzing evolutionary relationships. With help of MSA constructed phylogenetic trees indicate how sequences (or the species they represent) are related and show their common ancestors. For the comparison of parallel applications, MSA methods are interesting because they allow a detailed comparison of multiple processes in one step. By aligning a group of processes, MSA identifies equal areas that are shared by all compared processes and detects exact positions of differences. This information allows to aggregate the performance data of many processes into one representative processes with augmented performance information of all processes.

The naïve approach for solving the MSA problem works in principle identical to the dynamic programming approach for pairwise alignment. The difference is that each additionally aligned sequence adds a new dimension to the dynamic programming matrix. Instead of a two-dimensional matrix in case of a pairwise alignment, the MSA builds a k-dimensional matrix for k sequences. Thus, the search space increases exponentially with increasing k and is additionally dependent on the sequence lengths. The computational complexity to solve an MSA naïvely is $\mathcal{O}(n^k)$, where n is the sequence length (assuming that all sequences have the same length) and k is the number of sequences. Finding the optimal solution for the MSA problem is proven to be NP-complete [138].

As both the time needed to compute all matrix entries as well as the memory required to store the matrix exceed the capabilities of available computing hardware, heuristic solutions to the MSA problem have been proposed. A widely used heuristic is the progressive alignment approach, employed by the Clustal family [57, 132]. Compared to other MSA heuristics, like iterative or probabilistic methods, the progressive method is the fastest but also least correct approach. The source of higher inaccuracy in progressive methods is that initial errors are never corrected and are propagated throughout the complete multiple alignment. Yet, as the number of aligned sequences can get very large for the comparison of parallel processes, the performance of the MSA method is the major selection criterion.

The workflow of a progressive alignment is depicted in Figure 5.5. In principle, the algorithm takes a pairwise alignment and then successively adds new sequences to this alignment result. This work refers to the MSA result, the alignments of the input sequences, as *multiple sequence alignment block*. First, the MSA block holds only the alignments of the initial two sequences. Then, the algorithm progressively adds new alignments to the MSA block. When the alignments of all sequences are added, the MSA block represents the final MSA result. Adding new sequences is done by performing a pairwise alignment with one sequence taken from the MSA block and one new sequence. This step aligns the new sequence to the MSA block. During a pairwise alignment step, one change can occur that has influence on the complete MSA block. If new Gaps are inserted in the sequence taken from the MSA block, this Gap needs to be inserted into all sequences of the MSA block. This situation, informally speaking described with “once a Gap, always a Gap”, is the major source of error propagation during the progressive alignment. To keep the error propagation as low as possible, it is important that the most similar sequences are compared first. Therefore, progressive alignment algorithms first construct a so-called *guide tree* that orders sequences according to their similarity. To determine the similarity between sequences, MSA methods build a similarity matrix using pairwise alignment scores as distance metric. During the progressive alignment scheme the guide tree selects the most similar (remaining) sequences for the alignment. This procedure ensures that similar sequences are aligned first. The progressive alignment is finished after all sequences are added to the MSA block.

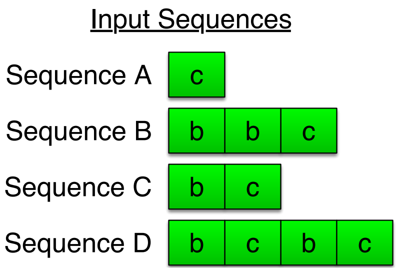
5.2.2 Hierarchical Alignment Algorithm for Multiple Processes

This section introduces an MSA approach for the alignment of multiple process sequences or traces. The goal of the resulting alignment is the detailed analysis of structural similarities and differences between the aligned processes. The employed MSA method works by progressively adding sequences to an MSA block. Each addition of a process’s sequence requires one pairwise alignment step. As shown in Chapter 3, the times required for aligning complete process sequences might render the MSA approach infeasible. Therefore, the progressive MSA approach is placed on top of a hierarchical scheme, similar to the hierarchical approach described in Section 3.3.1.

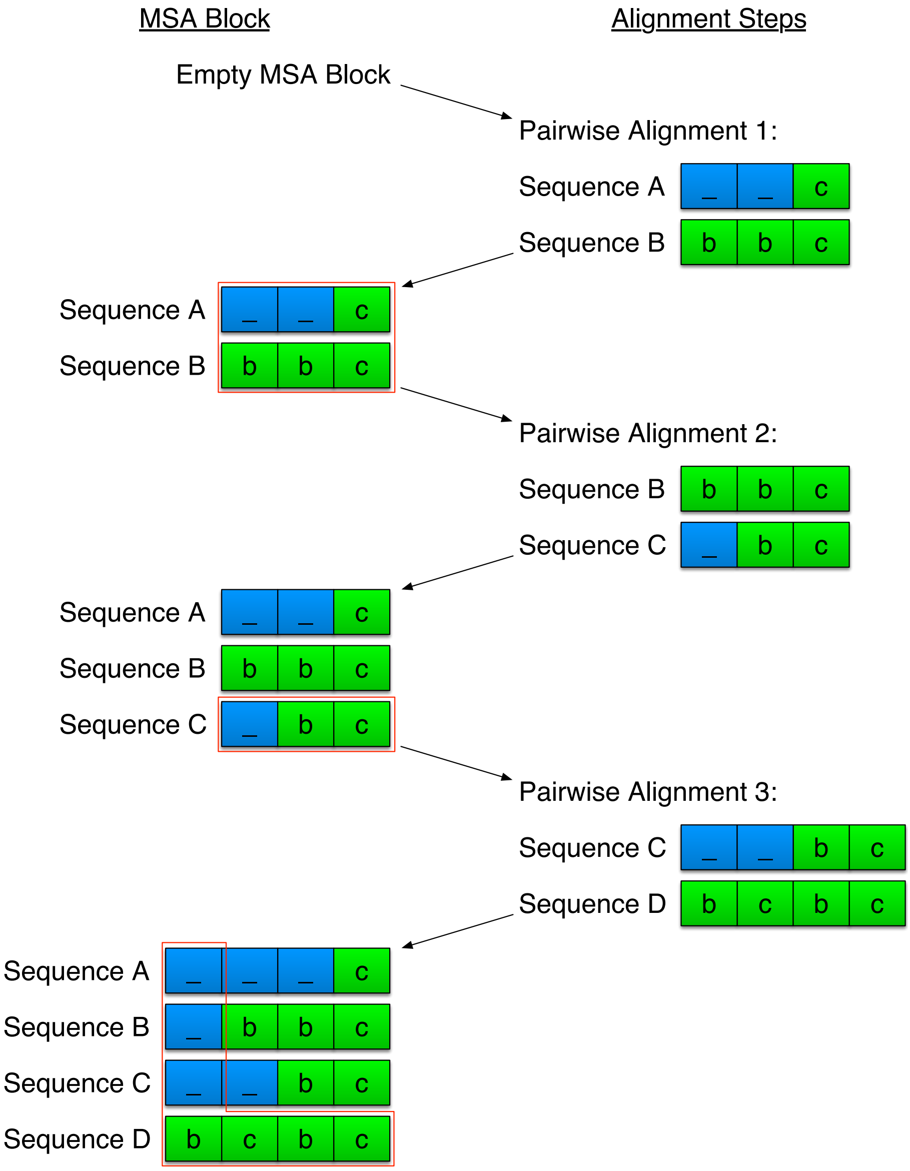
Progressive Alignment of Multiple Process Sequences

This section demonstrates the construction of an MSA block that contains the alignment of multiple sequences. The key idea of building the complete MSA block is to progressively add new sequences by using pairwise alignment methods. During this process, introduced Gap states need to be distributed to all sequences already aligned in the MSA block.

Figure 5.6 demonstrates the progressive multiple sequence alignment process. The illustrated example uses four sequences, shown in Figure 5.6(a). The goal of the MSA is to align all four example sequences. As shown in Figure 5.6(b), the MSA process requires three pairwise alignment steps. The first step aligns `Sequence A` with `Sequence B`. With that alignment both sequences build the initial MSA block. Then `Sequence C` is taken and aligned to one sequence of the MSA block, in this example to `Sequence B`. The resulting MSA block contains all three sequences. To complete the MSA block, `Sequence D` needs to be added by an alignment with one sequence from the block. In this example `Sequence D` is aligned with `Sequence C`. During this alignment step, a new Gap state is introduced



(a) The four input sequences used in this MSA example.



(b) Three individual pairwise alignment steps are required for progressively adding all input sequences to the multiple sequence alignment block. The MSA block at the bottom represents the final result of the multiple sequence alignment.

Figure 5.6: This example demonstrates the computation of a multiple sequence alignment (MSA) using four input sequences.

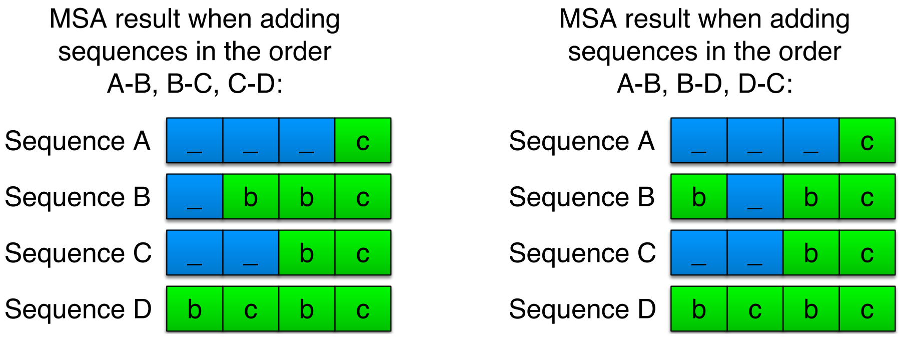


Figure 5.7: Results of an MSA can differ depending on the order in that sequences are added to the MSA block. This figure contrasts two possible results for the sequences given in Figure 5.6(a).

into a sequence from the MSA block (first Gap in `Sequence C`). This Gap needs to be distributed to all sequences currently in the MSA block. The result of this step is shown at the bottom of Figure 5.6(b). This bottom MSA block also represents the final MSA result, as no additional sequences need to be added.

The result of the MSA depends on the order in that sequences are added to the MSA block. Also the individual process within the MSA block that is selected for the pairwise alignment with a new sequence has an influence on the final MSA result. In the above example, sequences are added using the following pairwise comparisons, `Sequences:` `A-B`, `B-C`, and `C-D`. However, it is possible to arrive at a different result by adding sequences in the order: `A-B`, `B-D`, and `D-C`. Figure 5.7 contrasts the resulting MSA blocks for both options.

The two results in Figure 5.7 differ by the alignment of the first character `b` in `Sequence B`. The result shown on the right side in the figure represents the better MSA of the sequences given in Figure 5.6(a). The key to achieve high quality MSA results is to always align the most similar sequences in the pairwise alignment steps. In the initial version of the example, demonstrated in Figure 5.6(b), `Sequence D` has been aligned with `Sequence C`. In the alternative version leading to the result shown on the right side in Figure 5.7, `Sequence D` has been aligned with `Sequence B`. As `Sequence B` is more similar to `Sequence D` than `Sequence C` to `Sequence D`, the alternative version yields the better MSA result.

In order to facilitate high quality MSA results, a guide tree determining the right order of pairwise comparisons is necessary for the alignment process. The next section describes the computation of this tree.

Fast Guide Tree Construction

The so-called *guide tree* determines the order in that sequences are added to the MSA block. It selects the sequence pairs for the consecutive alignment steps. MSA algorithms applied in computational biology use UPGMA [57] or Neighbor-Joining [132] methods to construct the guide tree. Naïve implementations of both methods run with $\mathcal{O}(n^3)$ time complexity.

The most time spent during the complete MSA calculation is required for the computation of the pairwise similarities used to construct the guide tree [57]. Therefore, the construction method of the guide tree requires an adaption to be suitable for the progressive alignment of process sequences. Using pairwise alignments to compute the distance between processes is an expensive operation by itself. This combined with the number of required comparisons quickly renders the computation of a similarity matrix infeasible. As described in Section 5.1.1, even a similarity matrix holding only 1,024 processes already requires 523,776 comparisons. Assuming for example an alignment time of 11,291 ms for one comparison, like measured in Section 3.5.2 for the application ParaDiS, would result in more than 68 days of computation time to fill the similarity matrix.

Algorithm 6: Max2FreqHashing algorithm

```
Input: Two sequences $A$ and $B$.
Output: Similarity score $Sim_{score}$.
$Sim_{score}$ = 0
select the two most frequent characters $A_1$, $A_2$ from $A$
select the two most frequent characters $B_1$, $B_2$ from $B$
count character $A_1$ occurrences in $A$
count character $A_2$ occurrences in $A$
count character $B_1$ occurrences in $B$
count character $B_2$ occurrences in $B$
from $A_1$, $A_2$, $B_1$, $B_2$ select the characters $C$ that occur in $A$ and $B$; $C := \{A_1, A_2\} \cap \{B_1, B_2\}$
foreach $x$ in $C$
    take character $x$ occurrences and add value to $Sim_{score}$
return $Sim_{score}$
```

To compute the similarity between two process sequences quickly, a more efficient method than alignment is required. This work uses the *MaxKFreqHashing* method proposed by Seker [126] to approximate the similarity between two sequences. To compute the similarity, this approach first counts the occurrences of the K most frequent characters in each sequence. The following examples show the result of this function counting the most two, $K = 2$, characters (Max2FreqHashing):

`research` : 2 “`r`” and 2 “`e`”
`sequence` : 3 “`e`” and 1 “`s`”

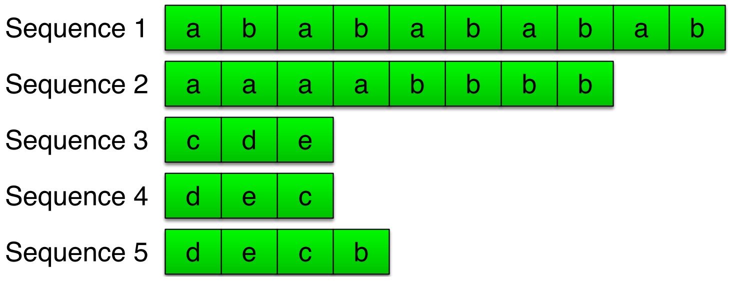
Then, to retrieve a similarity score, the counts of the characters occurring in both sequences are summed up. For the example above, the similarity between `research` and `sequence` is 5. The character “`e`” occurs 5 times in both sequences, “`r`” and “`s`” are only occurring in one sequence and do not contribute to the similarity score.

Algorithm 6 lists the Max2FreqHashing approach. This algorithms requires $\mathcal{O}(n\ log(n))$ time complexity, as the algorithm needs to iterate over the sequences and to sort the results. Consequently, using the Max2FreqHashing algorithm in place of dynamic programming methods significantly accelerates the computation of the similarity matrix. Max2FreqHashing algorithm calculates, especially for long process sequences, an approximation of their structural similarity in acceptable time.

The following example illustrates the complete process from computing the similarity matrix to determining the series of sequence pairings used to build the MSA. This approach retrieves the series of pairings directly from the similarity matrix, rather than additionally constructing a guide tree structure. The computed series of pairings represents the same information that is included in a guide tree. In principle it is possible to construct a guide tree for visualization. This however, would serve only illustrative purposes. Consequently, in the following the term *guide tree* refers only to the series of sequence alignments computed during the MSA block construction, without indicating an actual tree structure. This example aligns the input sequences depicted in Figure 5.8.

Using the Max2FreqHashing algorithm to approximate the similarity between the input sequences results in the following similarity matrix:

$$M = \begin{array}{r|cccc} & \text{Sequence 1} & \text{Sequence 2} & \text{Sequence 3} & \text{Sequence 4} \\ \hline \text{Sequence 2} & 18 & & & \\ \text{Sequence 3} & 0 & 0 & & \\ \text{Sequence 4} & 0 & 0 & 4 & \\ \text{Sequence 5} & 6 & 5 & 4 & 4 \end{array}$$



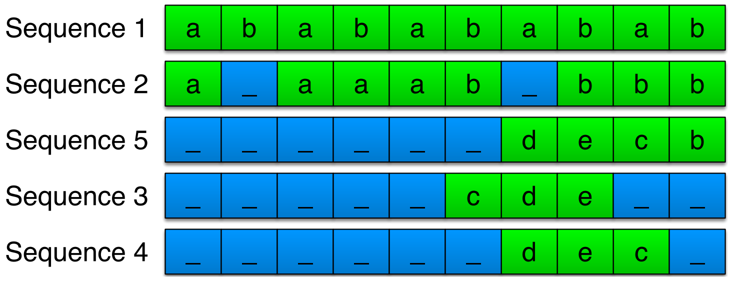
Sequence 1	a	b	a	b	a	b	a	b	a	b
Sequence 2	a	a	a	a	b	b	b	b		
Sequence 3	c	d	e							
Sequence 4	d	e	c							
Sequence 5	d	e	c	b						

Figure 5.8: Input sequences for the guide tree construction example.

Sequences that share high similarity are added to the MSA first. Therefore, the comparisons stored in the similarity matrix are sorted according to their similarity value. Table 5.2 shows the sorted pairwise comparisons with the highest similarity values. The order in that sequences are added to the MSA is defined by following comparisons from top to bottom in Table 5.2. First, `Sequence 1` and `Sequence 2` are aligned and build the initial sequences of the MSA block. Next, `Sequence 1` and `Sequence 5` are aligned. This adds `Sequence 5` to the MSA block. The next comparison in Table 5.2 is `Sequence 2` with `Sequence 5`. Since both sequences are already contained in the MSA block, this comparison is unnecessary. The next comparison is `Sequence 3` with `Sequence 4`. This comparison needs to be postponed, as one sequence is required to be contained in the MSA block. The next comparison according to Table 5.2 is `Sequence 5` with `Sequence 3`. This alignment adds `Sequence 3` to the MSA. Now, since `Sequence 3` has been added to the MSA block, the alignment of `Sequence 3` and `Sequence 4` can be computed. This step adds the last missing sequence, `Sequence 4`, to the MSA block. Consequently, for this example, the guide tree reflects the following series of alignments: `Sequence 1-2`, `Sequence 1-5`, `Sequence 5-3`, and `Sequence 3-4`. Adding sequences in the described order produces the MSA result shown in Figure 5.9.

Table 5.2: Table with ordered results from the similarity matrix.

SIMILARITY SCORE	SEQUENCES TO COMPARE
18	`Sequence 1 - Sequence 2`
6	`Sequence 1 - Sequence 5`
5	`Sequence 2 - Sequence 5`
4	`Sequence 3 - Sequence 4`
4	`Sequence 5 - Sequence 3`
4	`Sequence 5 - Sequence 4`



Sequence 1	a	b	a	b	a	b	a	b	a	b
Sequence 2	a	_	a	a	a	b	_	b	b	b
Sequence 5	_	_	_	_	_	_	d	e	c	b
Sequence 3	_	_	_	_	_	c	d	e	_	_
Sequence 4	_	_	_	_	_	_	d	e	c	_

Figure 5.9: MSA result achieved by adding sequences using a guide tree approach.

Using the Max2FreqHashing algorithm to compute the similarity between sequences lowers the time required for the computation of the similarity matrix. Nevertheless, the large number of processes in parallel applications can render the computation of a similarity matrix infeasible for a sequential implementation. In such cases, a faster heuristic not requiring the computation of all pairwise similarities between processes is necessary. The following describes such a method. The heuristic is based on the MaxKFreqHashing algorithm and counts the three most frequent characters, $K = 3$, Max3FreqHashing. First, all sequences are searched to find the three most frequent characters (top characters) occurring in the input data. Then, sequences are ranked according to their number of top characters. The sequence containing the most occurrences of top characters acts as the "main sequence". All remaining sequences are aligned to the main sequence in the order of their top character occurrences.

Using the example sequences given in Figure 5.8, Table 5.3 lists the number of occurrences of individual characters.

Table 5.3: Number of occurrences of characters in the sequences given in Figure 5.8.

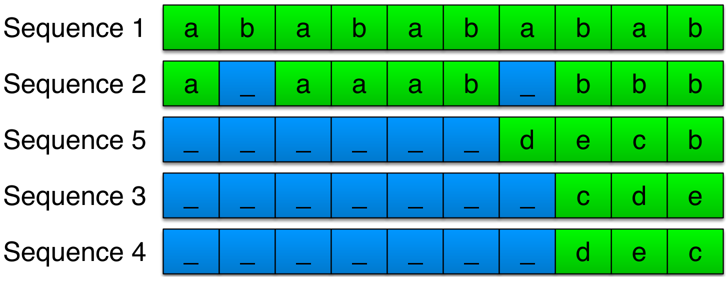
Occurrences	Character
10	`b`
9	`a`
3	`c`
3	`d`
3	`e`

Following Table 5.3, the resulting three top characters for this example are: `b`, `a`, and `c`. Table 5.4 shows the example sequences ranked according to their number of top characters.

Table 5.4: Sequences of Figure 5.8 ranked according to the number of their top characters.

Occurrences of top characters	Sequence
10	`Sequence 1`
8	`Sequence 2`
2	`Sequence 5`
1	`Sequence 3`
1	`Sequence 4`

The sequence containing the highest number of top characters is `Sequence 1`. Thus, this sequence acts as the *main sequence* for this example. Applying the ranking in Table 5.4, sequences are added to the MSA using the following comparisons: `Sequence 1-2`, `Sequence 1-5`, `Sequence 1-3`, and `Sequence 1-4`. Figure 5.10 presents the resulting MSA when adding sequences in the described order.



Sequence 1	a	b	a	b	a	b	a	b	a	b
Sequence 2	a	_	a	a	a	b	_	b	b	b
Sequence 5	_	_	_	_	_	_	d	e	c	b
Sequence 3	_	_	_	_	_	_	_	c	d	e
Sequence 4	_	_	_	_	_	_	_	d	e	c

Figure 5.10: MSA result achieved by simply adding sequences using pairwise alignments with `Sequence 1`.

Comparing the results shown in Figure 5.9 and Figure 5.10, it is visible that the first result represents the higher quality MSA. The result shown Figure 5.9 aligns the calls to the functions *d* and *e* correctly. This highlights the importance of using a guide tree approach for the MSA result quality. In the second result, shown in Figure 5.10, the employed heuristic added all sequences by pairwise comparisons with `Sequence 1`. This heuristic might reduce MSA result quality. However, it also enables MSA for large numbers of processes, that would not have been comparable otherwise.

Algorithm 7: Hierarchical MSA Algorithm.

```
Input: A group of processes $P$.
Output: Merged call tree $mergedCallTree$ of all input processes.
$mergedCallTree$ = 0
Function computeMergedCallTree($P$)
    // Get call trees from processes
    $callTrees$ = getCallTree($P$)
    // Start hierarchical MSA with root nodes
    compareNodes(getRootNodes($callTrees$))
    // Return merged call tree
    return $mergedCallTree$
Function compareNodes($nodes$)
    // Compare functions of input nodes
    $state$ = compareFunctions($nodes$)
    // Add new node with result state to merged call tree
    addNewNode($mergedCallTree, state$)

    // If all nodes have no child-nodes the comparison stops here
    if $false$ = haveChildNodes($nodes$)
        return // All nodes are leaf nodes, done

    // Compare child-nodes of input nodes

    // Generate sequences from child-node calls of input nodes
    $subSequences$ = genChildNodeSequences($nodes$)
    // Compute multiple sequence alignment of sequences
    $subAlignment$ = calculateMultipleSequenceAlignment($subSequences$)
    // Iterate over sub-alignment
    foreach $alignedchildNodes$ in $subAlignment$
        // Continue hierarchical MSA with aligned child-nodes
        compareNodes($alignedchildNodes$)
    return // done
```

Hierarchical Scheme

Progressive pairwise alignment steps are necessary to build a multiple sequence alignment. Thus, similarly to the situation described in Section 3.3, large sequence lengths may inhibit the computation of an MSA in an acceptable time. As shown in Section 3.5, flat alignments of function sequences from real-world applications may easily require computation times in the range of hours. Therefore, an adaption of the MSA method is necessary to ensure fast computation of multiple sequence alignments of application processes. The key to speed up the computation is to split up complete process sequences into shorter segments. Analogous to the pairwise hierarchical alignment algorithm, the hierarchy of processes can be exploited to accelerate multiple sequence alignments. By augmenting the MSA approach with a hierarchical scheme, individual pairwise alignments are executed with significantly shorter sequences.

The pseudo-code given in Algorithm 7 describes the hierarchical MSA alignment algorithm in detail. The call trees of the input processes are traversed in parallel. During the traversal one merged call tree that contains the alignment information of the input processes is generated. Each traversal step first compares the states of the current nodes and generates the respective merged node in the resulting merged call tree. Then it aligns all child-nodes of the current nodes using MSA. The MSA result aligns

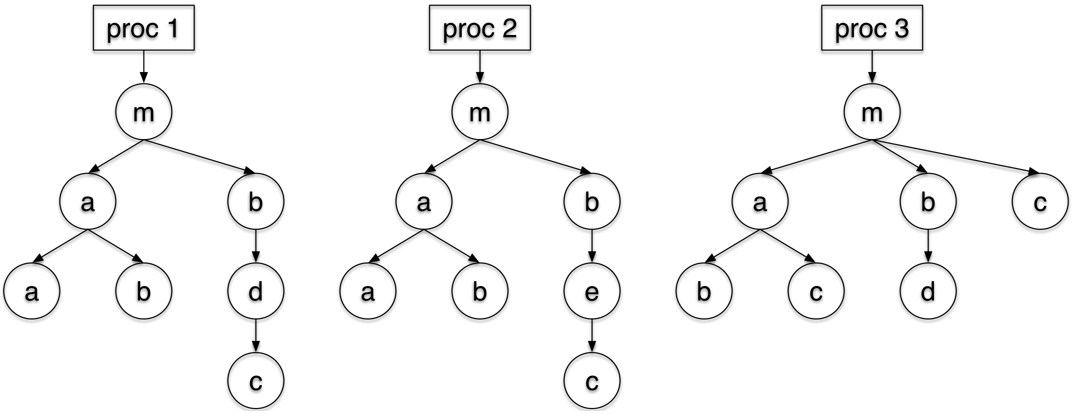


Figure 5.11: Input processes for the hierarchical MSA scheme example.

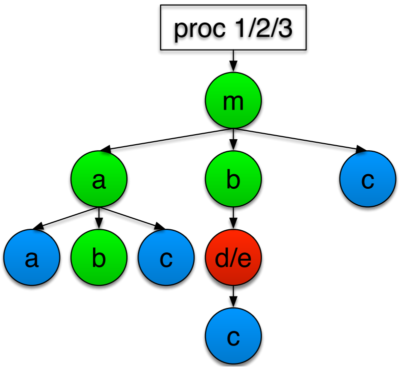


Figure 5.12: The merged call tree for the example processes, computed by the hierarchical MSA algorithm.

related child-nodes, and thus, defines the node groups for subsequent traversal steps. The construction of the resulting merged call tree is finished when the complete input call trees are traversed.

In order to demonstrate the hierarchical MSA scheme with an example, Figure 5.11 provides three input call trees. Figure 5.13 illustrates the individual comparison steps of the hierarchical MSA algorithm for the three example processes. The hierarchical MSA algorithm starts with the root nodes of the input call trees. The first step compares all child-nodes of the root nodes, resulting in the merged call tree shown in step two. The second step aligns the child-nodes of the three nodes for function `m`. This detects a gap in the first two processes and results in the merged call tree shown in step three. The algorithm continues the traversal of the call trees following the alignment result shown in step three. First, step three, compares the child-nodes of the aligned `a` nodes. This step detects three gap states and results in the merged call tree shown in step four. As all nodes of step four are leaf-nodes, no deeper traversal is necessary. Thus, the algorithm continues traversal following the results shown in step three. Step four compares the child-nodes of the aligned `b` nodes. This comparison merges two different states—`d` and `e`—into one node. Step five compares the child-nodes of the merged node created in step four. This comparison detects a gap in process three and results in the call tree shown in step six. As all remaining nodes (node `c` in step three and step six) are leaf nodes, the traversal of the input call trees is complete. The resulting merged call tree for this example is shown in Figure 5.12. Colors in the figure indicate the alignment state of individual nodes. Green nodes are equal across all input processes. Blue nodes represent a gap, indicating at least one missing process. Red nodes indicate different functions merged together. For this example, the merged call tree in Figure 5.12 identifies four equal nodes. Four nodes exhibit a gap state, indicating that at least one process is missing. One node is marked as different as it contains two different functions.

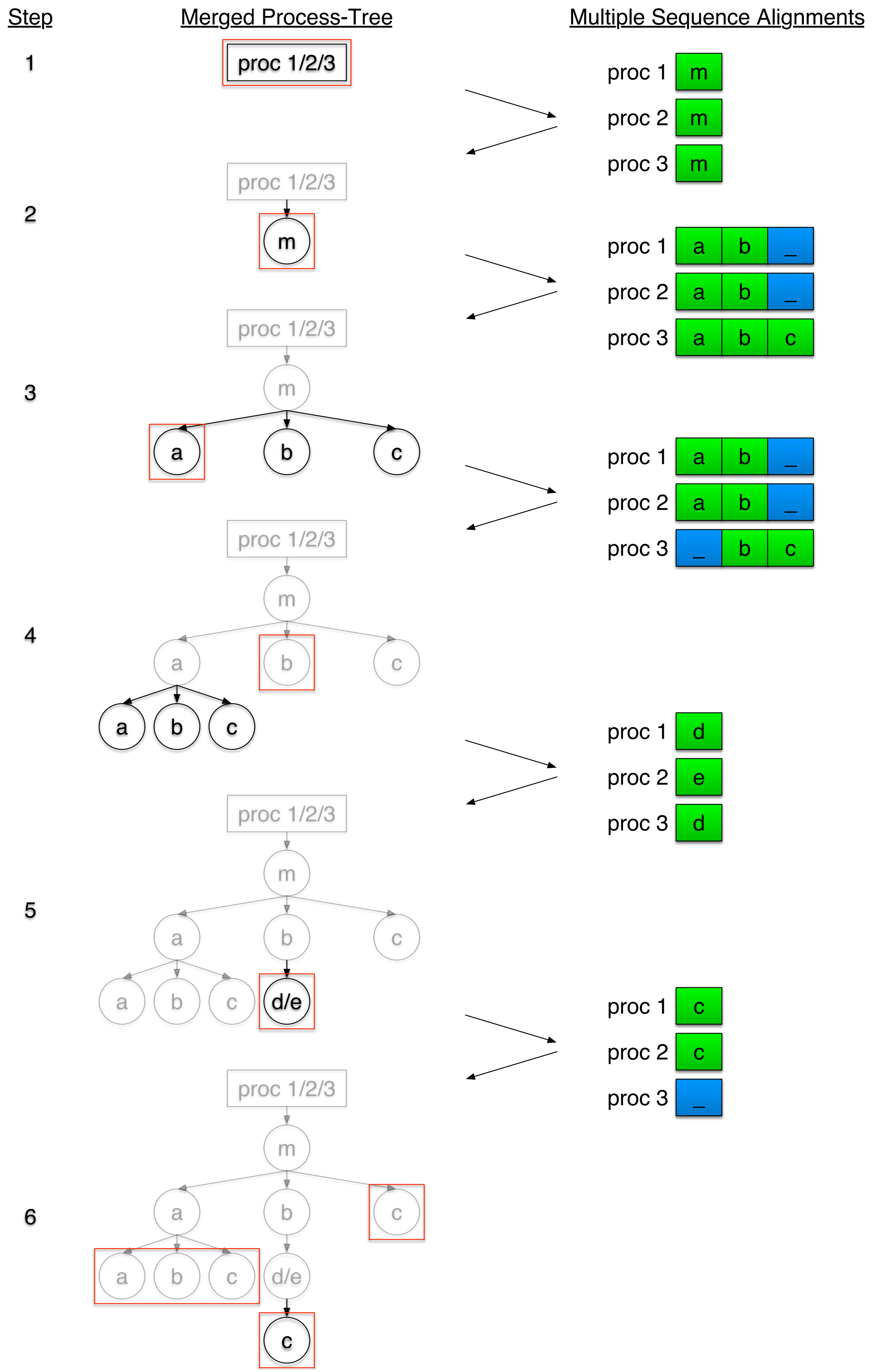


Figure 5.13: Hierarchical MSA algorithm scheme for the example call trees shown in Figure 5.11. Red rectangles indicate the nodes used for the individual MSA steps. After step six no further MSA are necessary as all remaining nodes are leaf nodes.

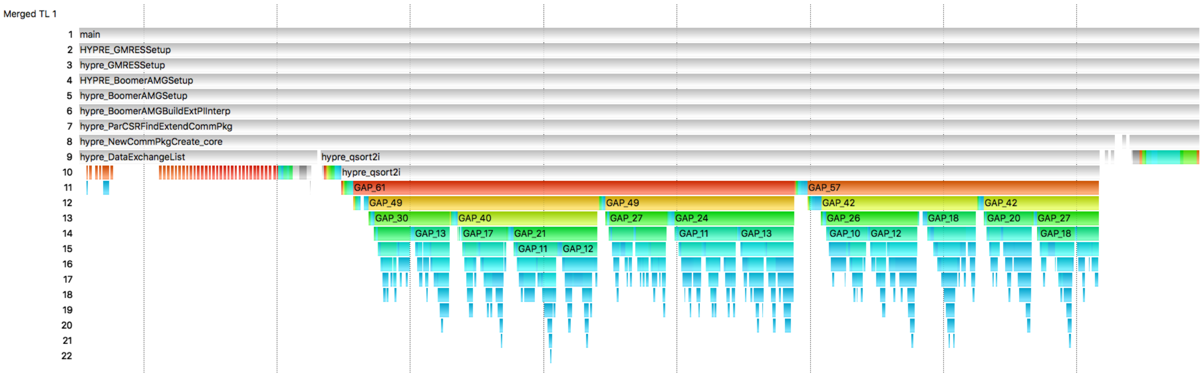


Figure 5.14: Portion of the merged call tree of 64 AMG2006 processes. The colored parts in the center show the recursive execution of a sort function. Starting from call-level 11, more and more processes stop execution of their sorting (indicated by colors changing from red to blue).

The hierarchical MSA algorithm has the capability to compare the processes of a parallel application. The resulting merged call tree reveals detailed structural differences and similarities between the input processes. The following two examples showcase merged call trees for two real-world application runs. The colors in Figures 5.14 and 5.15 indicate the alignment state and the number of processes contributing to individual nodes. White nodes depict equal structure. All merged processes execute the same function in such nodes. Colored nodes depict differences in the structure between processes, while the individual node color indicates the number of merged processes for this node. Red—warm—colors indicate a high number of processes that contribute to the individual node. Blue—cold—colors indicate a low number of processes that contribute to the individual node.

Figure 5.14 shows a portion of the merged call tree generated from a 64 process run of the application AMG2006 [34]. The figure depicts a part of the initialization. The center of Figure 5.14 shows the recursive execution of a sort function. First, visible at call-level 10 in Figure 5.14, all processes execute the sort function `hypre_qsort2i`. Then, beginning at call-level 11, more and more processes have finished their sorting and stop calling the sort function. Finally, at call-levels 19–22, only a few remaining processes perform sorting steps. For the parallel execution of the sort function, Figure 5.14 shows the structural differences between all application processes in one merged call tree.

Figure 5.15 shows the merged call tree for an eight process run of the application ParaDiS [7]. The figure shows a few computation steps during the execution of the application. The colored parts of the merged call tree highlight parts of the application that are not executed by all processes. Blue areas are executed by one process only. What process actually runs during blue areas changes throughout the execution and is dependent on the individual workload of the processes. This is not directly visible in Figure 5.15, as the chosen visualization approach only encodes the number of processes and does not indicate individual processes.

Concluding, the merged call tree combines the structural information of multiple processes in one graph, while highlighting similarities and differences at the same time. As the merged call tree contains a large amount of information, more than the demonstrated visualizations are possible. This data structure provides the potential to design new powerful use case specific visualization approaches.

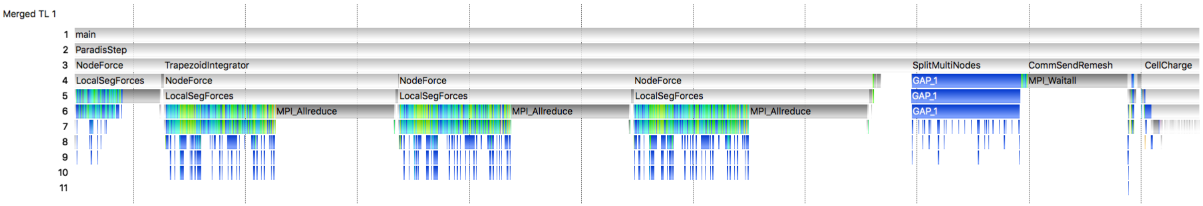


Figure 5.15: Portion of the merged call tree of 8 ParaDiS processes showing a few computation steps. Colored nodes indicate areas that are not executed by all application processes. Especially blue parts are only executed by one process.

Scalability Considerations

This chapter introduces an approach for the structural comparison of multiple parallel processes. The approach consists of several steps.

The first part groups structurally similar processes. This is done by a hash-based clustering algorithm. The first step of this algorithm, the extraction of the structural data, requires a complete traversal of all call trees of the application processes. Most steps of the algorithm (extraction and serialization of the structural data, computation of the hash value, and insertion of the hash value in the cluster list) require at most linear time, $\mathcal{O}(n)$, with respect to the number of entries n in the largest process call tree $n = max(n_1, ..., n_p)$, with the number of processes p. The sorting of the structural data requires $\mathcal{O}(m\ log(m))$ time complexity, with respect to the number of entries m in the structural data list. Since this list consists of the different functions or function pairs occurring in a call tree, the number of entries in this list is usually much smaller than the number of entries in the call tree, $m << n$. The overall worst-case time complexity of the hash-based clustering is $\mathcal{O}(n\ log(n))$. The expected time complexity for the clustering of processes is $\mathcal{O}(n)$.

Then, the hierarchical MSA algorithm compares the clustered processes. Therefore the algorithm traverses the process call trees in parallel. This traversal requires linear time complexity, $\mathcal{O}(n)$.

During the traversal MSA are computed for child-node sequences. The computation of an MSA consists of two parts. The first part computes the guide tree. This computation is based on the MaxKFreqHashing algorithm, that runs with $\mathcal{O}(n\ log(n))$ time complexity, with respect to the sequence lengths n. There are two options to compute the guide tree. The first option uses the Max2FreqHashing algorithm to compute a similarity matrix. The worst-case time complexity of this approach is quadratic, $\mathcal{O}(p^2)$, with respect to the number of processes p. The second option is to use the Max3FreqHashing algorithm to compute a sorted table. This approach runs in $\mathcal{O}(p\ log(p))$ worst-case time complexity.

Using the guide tree, progressive pairwise alignments are computed. Each alignment requires quadratic worst-case time complexity, $\mathcal{O}(n^2)$, with respect to the sequence lengths n.

The overall worst-case time complexity of the hierarchical MSA algorithm is quadratic, $\mathcal{O}(n^2)$. However, similarly to the solution described in Section 3.3.2, a fast-alignment-heuristic method can be used to limit impact of the quadratic component in the algorithm. In such case, the construction of the guide tree, requiring $\mathcal{O}(n\ log(n))$ time complexity, dominates the algorithm. Thus, the expected time complexity for the hierarchical MSA algorithm is $\mathcal{O}(k\ log(k))$, with $k = max(n, p)$.

5.3 Performance Evaluation

This section evaluates the performance of both the introduced clustering algorithms and the hierarchical multiple sequence alignment algorithm. The case study in this section consists of the following real-world applications:

- **ParaDiS** [7], "Parallel Dislocation Simulator", models the dynamics of dislocation lines as they interact and move in response to the forces imposed by external stress and inter-dislocation interactions.
- **PEPC** [41], "Pretty Efficient Parallel Coulomb Solver", an N-body simulation using an octree as internal representation. It simulates the interaction between a laser and charged particles.
- **FD4** [83], "Four-Dimensional Distributed Dynamic Data structures", adds dynamic load balancing to the COSMO-SPECS and WRF weather forecast models. The measured application run has been a benchmark that evaluated FD4's scalability.
- **PIConGPU** [20, 21], "Particle-In-Cell on Graphics Processing Units", an implementation of a fully relativistic Particle-In-Cell (PIC) algorithm, which is widely used in computational plasma physics.

All performance measurements have been conducted on an HPC system with the following setup. One compute node is equipped with two Intel Xeon E5-2680 v3 processors. Each processor provides 12 cores and runs at 2.50 GHz with hyper-threading disabled. The available memory of a node ranges from 64 GB to 256 GB. To measure the benchmark runtime one node has been reserved exclusively with only one core running the benchmark actively. All reported results in this section are always the median of ten repeated measurement runs.

5.3.1 Performance of the Clustering Algorithms

This chapter introduces three algorithms applicable as pre-clustering step for grouping of structurally similar processes. Two algorithms, K-Medoids and DBSCAN, are well known and widely used. The novel fast hash-based clustering algorithm introduced in this work presents an additional alternative approach. Goal of the hash-based approach is to prevent the computation of a similarity matrix and to provide an algorithm that is capable of quickly clustering large numbers of processes.

The hash-based approach does not require a similarity matrix. This advantage is important when comparing large numbers of processes, as the memory requirement for a similarity matrix becomes quickly prohibitive. Table 5.5 shows the required memory for a similarity matrix with rising numbers of processes. Starting already with hundreds of thousands of processes, the similarity matrix cannot be stored on common desktop computers anymore. When approaching millions of processes, the storage of the similarity matrix becomes unreasonable even on powerful HPC machines.

The application ParaDiS is used to evaluate the scalability of all three approaches. For this purpose ParaDiS has been run with increasing process numbers. Compared to many other applications, the structural clustering of ParaDiS is relatively expensive. This is due to the varying function call structure between ParaDiS processes. The function call structure of a process depends on its individual workload. This results in a lot of structurally different process groups, what increases clustering effort. Consequently, this benchmark represents an involved clustering case.

For K-Medoids the number of clusters has been set to the number of groups previously computed by the hash-based clustering approach. DBSCAN has been configured to find clusters with at least two members and with a minimum of 70% structural similarity between processes. The clustering results of both algorithms strongly depend on the chosen algorithm parameters. Therefore, result quality has not been evaluated in this benchmark. Moreover, suitable clustering algorithm parameters may depend on the individual application run and cannot be generally defined for the clustering of processes.

Table 5.5: Memory requirement for the similarity matrix computed by K-Medoids and DBSCAN.

Number of Processes	Memory Requirement
128	~32 kB
1,024	~2 MB
16,384	~512 MB
32,768	~2 GB
131,072	~32 GB
262,144	~128 GB
524,288	~512 GB
1,048,576	~2 TB

Table 5.6: Clustering times for ParaDiS.

		Clustering Approach				
No. of Proc.	Size	Hash Clustering		Sim. Matrix	K-Medoids	DBSCAN
8	138 MB	1ms	7 Groups	6ms	<1ms	<1ms
16	164 MB	2ms	15 Groups	20ms	<1ms	1ms
32	216 MB	4ms	27 Groups	62ms	<1ms	1ms
64	334 MB	6ms	42 Groups	281ms	2ms	3ms
128	612 MB	9ms	74 Groups	1s 223ms	13ms	18ms
256	1.1 GB	24ms	93 Groups	4s 409ms	81ms	118ms
512	3.4 GB	40ms	99 Groups	18s 441ms	1s 55ms	947ms
1,024	4.6 GB	80ms	187 Groups	1min 31s	8s 272ms	17s 652ms
2,048	7.1 GB	164ms	216 Groups	6min 55s	1min 24s	2min 49s
4,096	14 GB	340ms	207 Groups	33min 14s	9min 55s	25min 38s

Table 5.6 lists the results of this measurement. The sizes given in Table 5.6 and Table 5.7 represent the compressed OTF trace file size. As both K-Medoids and DBSCAN require the same similarity matrix, its computation is listed separately in the table. The complete time required to compute a DBSCAN or K-Medoids clustering is the sum of the times required to compute the similarity matrix and the respective clustering algorithm. The timing measurements in Table 5.6 represent only the pure algorithm times and exclude times required to load the measurement data into main memory.

Figure 5.16 depicts the hash-based clustering times for ParaDiS. The graph confirms the expected linear increase in runtime. The clustering time increases along with rising process numbers. Additionally, it depends on the size of the measurement data and group count. The number of groups influences the runtime, as new results need to be inserted into the list of clusters. The measurement size is important as the extracted structural data needs to be sorted and serialized before the computation of the hash value.

Figure 5.17 shows a quadratic increase in runtime for the K-Medoids algorithm. K-Medoids and DBSCAN exhibit about the same runtime, as for both algorithms the computation of the similarity matrix dominates the clustering time. Computing the similarity matrix for process numbers larger than 4,096 requires already more than half an hour. This renders the computation of structural clusterings using K-Means or DBSCAN unacceptable. For larger process numbers, even without the time required to compute the similarity matrix, both algorithms require substantially more computation time than the hash-based clustering approach.

To evaluate the clustering performance for higher process numbers, Table 5.7 shows an application study with three real-world applications. The computation of a similarity matrix requires more than one day in all three cases. Thus, it is impractical to obtain DBSCAN or K-Medoid clustering results for applications with this degree of parallelism. Especially since the clustering is intended as pre-clustering step before subsequent analysis techniques. The hash-based clustering of all three applications results in a small number of groups. This approach is capable of grouping application runs with more than

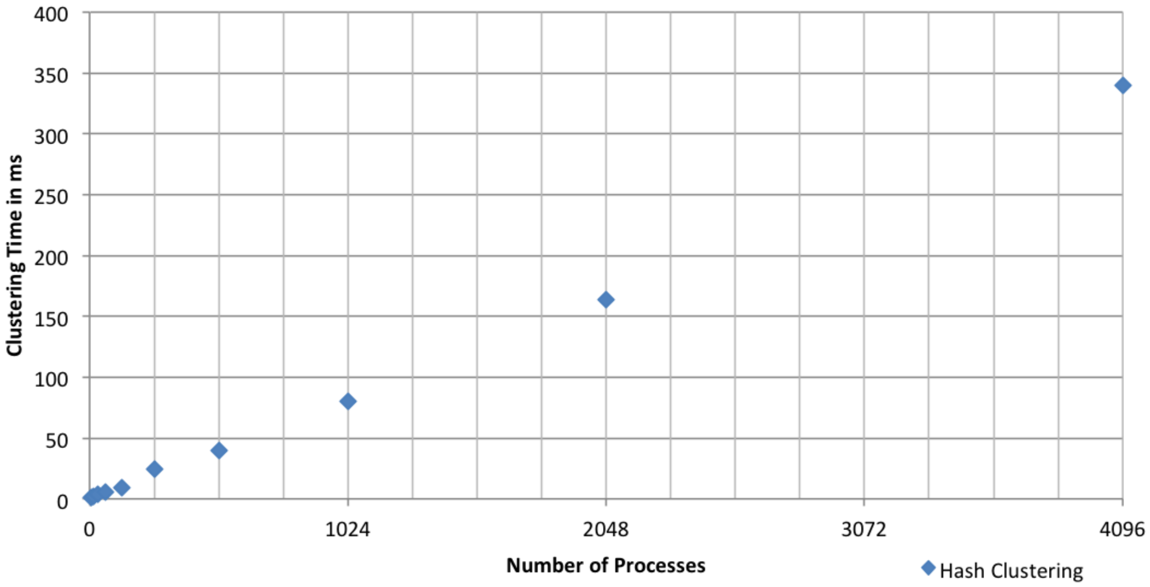


Figure 5.16: Scalability analysis of the hash-based clustering algorithm.

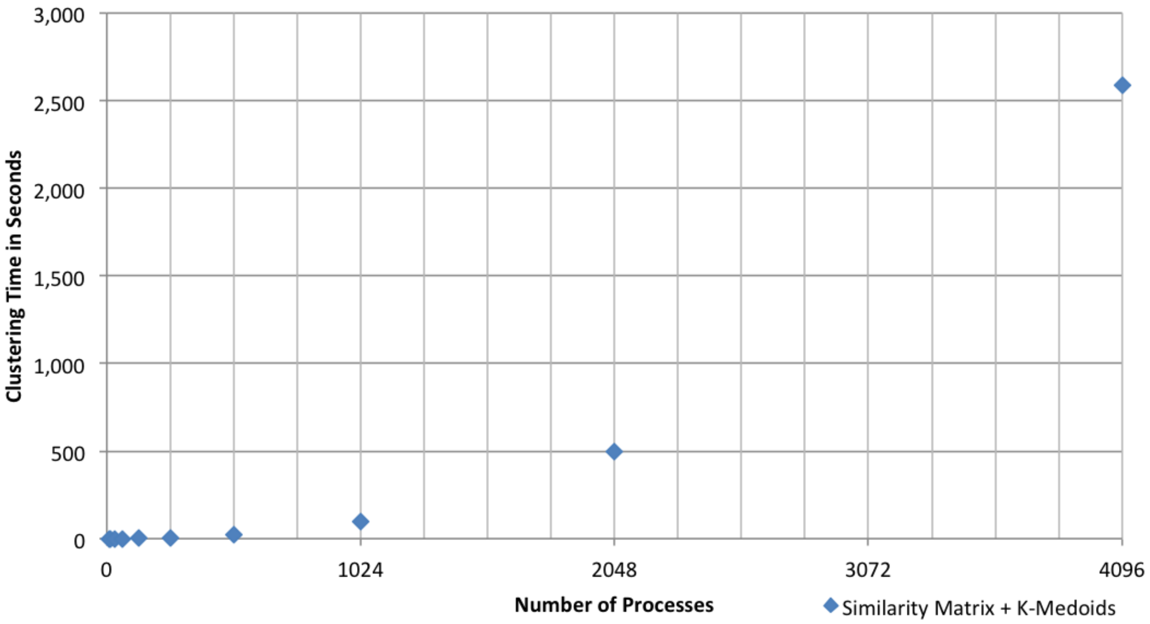


Figure 5.17: Scalability analysis of the K-Medoids algorithm. The displayed time represents the full algorithm time, i.e., similarity matrix + K-Medoids times shown in Table 5.6.

Table 5.7: Clustering time study with three real-world applications.

			Clustering Approach		
Application	Size	No. of Proc.	Hash Clustering		Similarity Matrix
pepc	4.4 GB	16,384	268ms	7 Groups	>24h
PIConGPU	97 GB	18,432	1s 209ms	12 Groups	>24h
fd4	2.6 GB	65,536	555ms	14 Groups	>24h

65,000 processes in about half a second. The clustering of the application PIConGPU requires the most computation time in Table 5.7, due to the size of the measurement data. The sorting of the structural data of the processes increases the algorithm runtime in this case. Nevertheless, this approach can compute the clustering result for PIConGPU in about one second.

The benchmark results demonstrate that the hash-based clustering is an efficient and viable approach for structural pre-clustering of applications. The approach is capable of quickly computing clustering results even for highly parallel applications.

5.3.2 Performance of the Hierarchical Multiple Sequence Alignment Algorithm

The first step in evaluating the multiple sequence alignment (MSA) algorithm's performance is to compare the two guide tree construction approaches. The more accurate approach requires the computation of a similarity matrix. The faster heuristic uses an ordered table without the need for exhaustive pairwise process comparisons. The measurement shown in Table 5.8 compares the guide tree construction times using the application ParaDiS. It presents the scalability of both approaches by measuring construction times for rising process numbers.

Table 5.8: Comparison of the two guide tree construction approaches. Tree generation times are measured for rising process numbers of the application ParaDiS.

Number of Processes	Size	Longest Sequ. Length	Similarity Heuristic	Similarity Matrix
8	138 MB	2,951,567	367ms	1s 649ms
16	164 MB	1,888,072	425ms	4s 277ms
32	216 MB	1,726,667	545ms	10s 636ms
64	334 MB	1,435,297	778ms	30s 148ms
128	612 MB	1,052,519	1s 451ms	1min 53s
256	1.1 GB	824,239	2s 267ms	5min 54s
512	3.4 GB	1,512,833	8s 822ms	44min 30s
1,024	4.6 GB	1,032,019	11s 359ms	1h 54min
2,048	7.1 GB	969,580	16s 999ms	5h 48min
4,096	14 GB	1,063,002	33s 569ms	21h 2min

The function sequence length of a ParaDiS process depends on its individual workload. Depending on the number of processes, the resulting distribution of work between available processes results in varying process sequence lengths. The measurements listed in Table 5.8 always compare complete process sequences. This makes the benchmark more compute intensive than in the case of computing the guide tree as part of the hierarchical MSA algorithm. The hierarchical approach separates complete process sequences into many smaller segments. This significantly accelerates the computation of the guide tree. However, in order to evaluate and compare the performance of both approaches it is reasonable to assess building times for full process sequences.

The left part of Table 5.8 provides the number of processes, the size of the measurement data set, and the length of the longest process sequence in the data set. The right part states the construction times for both approaches.

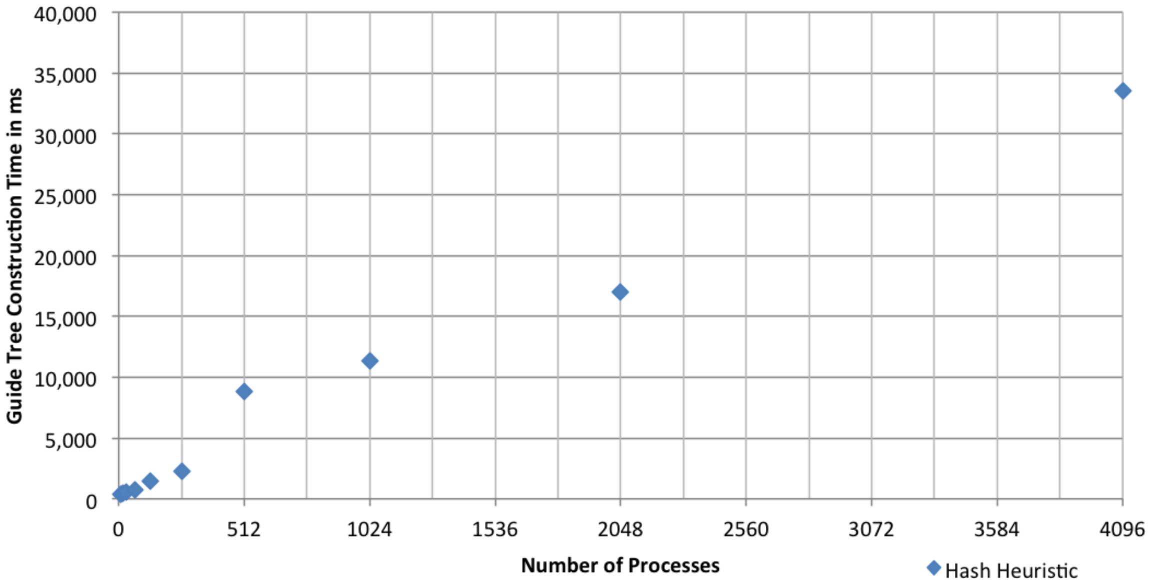


Figure 5.18: Scalability analysis of the heuristic guide tree construction approach.

Figure 5.18 shows the construction times for the faster guide tree approach computing an ordered table. This heuristic iteratively adds new processes to a table and sorts all entries according to the similarity score. The graph in Figure 5.18 shows a linear increase in the construction times for this approach. The value for 512 processes stands a bit out of the linear trend. This is due to the long sequence length for this number of processes. Nevertheless, the construction of the guide tree takes only about half a minute for 4,096 processes with long sequences lengths. This measurement suggests that this method is capable of efficiently constructing guide trees even for higher numbers of processes.

Figure 5.19 presents the calculation times for the similarity matrix based guide tree construction method. The graph shows the characteristic quadratic increase in runtime with rising process numbers. The construction time for 512 processes already takes more than 45 minutes. Using this method for constructing a guide tree is only reasonable with up to about 250 processes.

Concluding, efficiently computing a guide tree for high numbers of processes is infeasible when a similarity matrix is required. To benefit from the higher accuracy, the similarity matrix approach is applicable with up to 250 processes. For higher numbers of processes, the heuristic based on an ordered table allows fast constructions times. Consequently, the MSA algorithm uses a combination of both methods to construct the guide tree. It employs the similarity matrix based approach with up to 250 processes and uses the faster heuristic for higher process numbers.

The second part of this evaluation measures the MSA algorithm's performance using three real-world applications. For this evaluation the complete algorithm is divided into four stages. Table 5.9 presents the measurement results by separately listing the runtimes for the four individual stages of the algorithm. In the first stage, the algorithm detects and groups all completely similar input sequences. An MSA is only necessary between the remaining sequences that exhibit structural differences. In the second stage, the algorithm computes the guide tree that defines the order of alignments for the remaining sequences. The third stage computes the multiple sequence alignment between all remaining sequences according to the guide tree. Finally, the fourth stage takes the MSA information and constructs the resulting merged call trees.

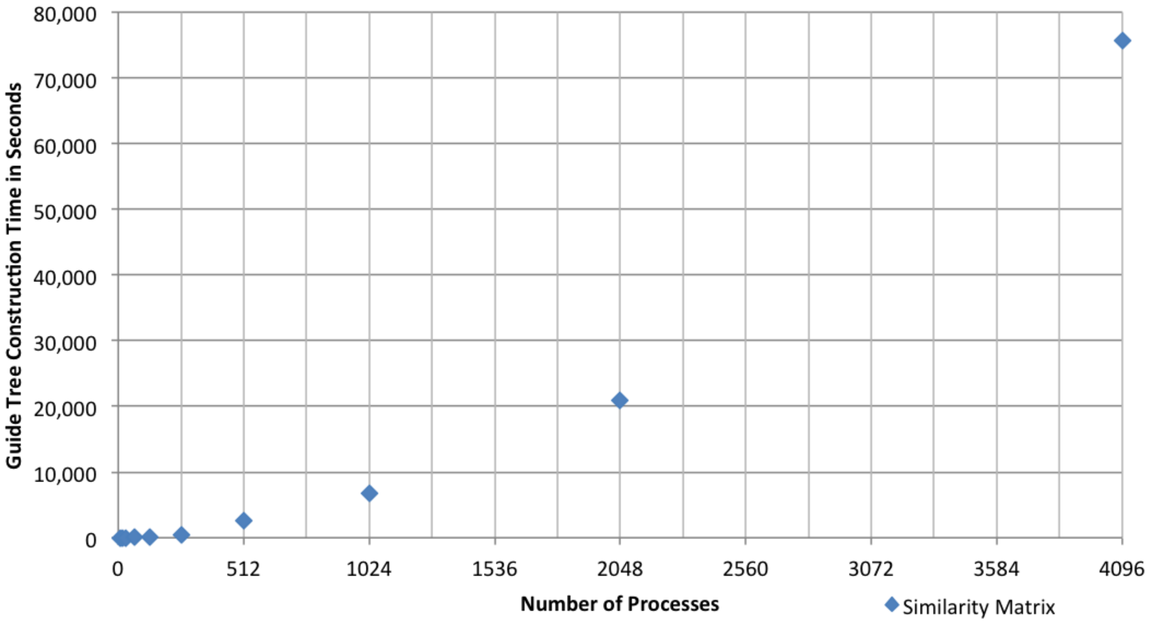


Figure 5.19: Scalability analysis of the similarity matrix based guide tree construction approach.

Table 5.9: Multiple alignment time study with three real-world applications.

APPLICATION	SIZE	NO. OF PROC.	GROUP EQUAL PROC.	BUILD GUIDE TREE	MULT. SEQU. ALIGN.	BUILD MERGED GRAPH	TOTAL
ParaDiS	14 GB	4,096	$2min$ 59s	36s 141ms	51s 22ms	6min 2s	10min 28s
pepc	4.4 GB	16,384	26s 884ms	2s 163ms	3min 6s	1min 18s	4min 53s
fd4	2.6 GB	65,536	1s 517ms	2ms	2ms	3s 36ms	4s 557ms

Computing the MSA and building the merged call trees for the application pepc takes about five minutes. In contrast, the same process takes only about five seconds for the application fd4. The reasons for the higher runtime in case of pepc are the larger amount of measurement data along with the higher number of differences between processes. Many fd4 processes are completely similar. Thus, the MSA algorithm needs to compute less guide trees and MSA alignments than in the case of pepc. Consequently, pepc also results in more complex merged call trees. Similarly, the situation for ParaDiS. This application produces large amounts of measurement data with many structurally different processes. Compared to the other applications, the pre-clustering step computes a high number of process groups for ParaDiS. Since the MSA algorithm builds an individual merged call tree for each group, the time required to construct these merged call trees dominates the total algorithm runtime. Moreover, most processes included in a group are structurally similar. This is visible from the higher runtime required to group equal processes compared to the runtime required to compute guide trees and MSA alignments.

In summary, the measurements show that the MSA algorithm is applicable even for applications with large process numbers. The algorithm is capable of merging detailed call tree information of tens of thousands of processes in only a few minutes. The merged call tree information is valuable for subsequent performance analyses and scalable visualizations of performance data.

6 Visualization Techniques for Event Timelines

The complexity of both high performance computing systems and their parallel software requires performance analysis tools to fully understand application performance behavior. Timeline visualization of event streams has proven to be a powerful approach for the detection of various types of performance problems [65]. It allows to leverage the powerful human visual perception. Clever visualization techniques can facilitate quick understanding of large data volumes. However, visualization of large numbers of process streams quickly hits the limits of available screen resolution. It is also not trivial to design beneficial visualizations [16]. Care needs to be taken during decisions like what part of the information to visualize and what visualization technique to apply [91].

This chapter makes two contributions to advance visualization techniques for event streams. The first section proposes folding strategies for event timelines that consider common questions during performance analysis. The introduced methods facilitate visual scalability and provide powerful overviews of performance data at the same time. Furthermore, the folding strategies improve GPU stream visualization and allow easy evaluation of the GPU device utilization. The second section presents an approach that analyzes runtime imbalances throughout an application run to automatically identify performance critical sections. Intuitive visualizations present the resulting runtime imbalances to the analyst and provide visual guidance to performance hot spots.

6.1 Timeline Folding Methods

This section introduces folding methods for event timelines. By folding multiple event timelines into one result timeline, scalability and usability of timeline visualizations can be greatly improved. Therefore, the required computation as well as possible folding and selection operations for visualizing a timeline are described. First, the basic operations that a timeline visualization involves are described. Second, the basic visualization is extended for folding capabilities. Based on these proposed operations and techniques, potential folding selections for different use cases are introduced in detail. Finally, this section analyzes the time complexity of the introduced techniques.

In the context of trace data, a timeline visualization is a rectangular display that visualizes streams as horizontal lines, which in turn consist of application states that are represented with a color. A stream usually represents a logical processing element such as a process, thread, or graphics processing unit (GPU) stream. Trace viewers such as Vampir [19] then interpret functions or other application events as states of the streams. Additionally, at a given point in time, applications may not execute any activity on the processing element that a stream represents. In such case a so-called *no-state* is inserted as application state.

The upper half of Figure 6.1 illustrates tracing data for a single stream as rectangles on a dotted line. The stream could represent an MPI [103] process and the states a and b could represent functions that the process issues. The tracing data represents these states with *enter* and *leave* events [137], i.e., the first event in the stream is an enter event for a, followed by an enter event for b, followed by a leave event for b, and so forth. Additionally, the example event data contains time regions with no active application state (*no-state*) in the visualization. A timeline visualization must represent this event data with the pixels that are available in its rectangular display. For simplicity it is now assumed that at least one pixel in height is available for each stream. For each stream and each pixel that the width of the display provides, the visualization must associate a state from the event data. The lower half of Figure 6.1 illustrates this task for three available pixels. Each pixel represents an equidistant time range that depends on its position and the time interval for the visualization, i.e., in the example, pixel 0 represents the time interval $[t_0, t_1)$.

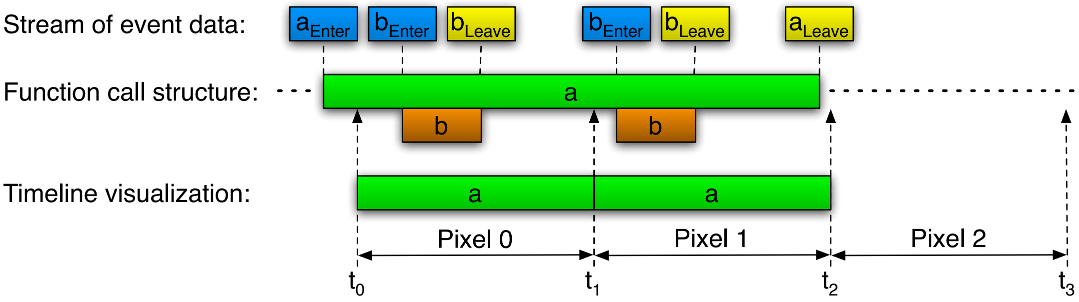


Figure 6.1: Illustration of the state selection for pixels in a timeline chart.

As the figure illustrates, multiple application events can occur within such a time range. Thus, based on the events in the time range of a pixel, a selection operation must associate a state with each pixel.

An obvious approach for this selection is an analysis of the amount of time that is spent in each state. Let $\mathcal{F}$ be the set of all application states with *no-state* $\in \mathcal{F}$, i.e., the example uses $\mathcal{F} = \{a, b, \textit{no-state}\}$. By scanning through all events that occur within the time interval of a pixel p, an analysis can create a mapping $t_p : \mathcal{F} \rightarrow \mathcal{R}$ that assigns a real number for each state to represent how much time is spent for this activity. The visualization could then choose the activity $s \in \mathcal{F}$ as the state for pixel p where s satisfies that for all $s' \in \mathcal{F}$ holds $t_p(s') \leq t_p(s)$. This approach would yield the chosen activities in the lower half of Figure 6.1, since state a is the most active activity for pixel 0, state a most active for pixel 1, and *no-state* is most active for pixel 2.

However, choosing pixel states by the most active state within a pixel comes at a high overhead. For k events in the time range of a pixel, computing t_p requires $\mathcal{O}(k)$ time. In practice, for a timeline visualization of the complete time range of an application's trace, this requires that the visualization analyzes the complete trace. Consequently, this approach is impractical for performance visualization tools such as Vampir.

To overcome this, a more sampling-like approach is often used: To select a state for a pixel with a time interval $[t, t')$, the visualization queries for the first event e with a timestamp $t_e \leq t$. If e is an enter event then its state is still active at time t. Thus, the state for the pixel is the state associated with e. If e is a leave event, then the state of its parent in the call stack is still active at time t. Thus, the state for the pixel is the state associated with the parent event e' of e. If no parent event exists, *no-state* is used instead. In the example, for pixel 0 the first event to the left of t_0 is an enter event for a, thus the sampling chooses a as the state for the pixel. For pixel 1, the first event left of t_1 is a leave event for b, since a is the parent event of b, the state for the pixel is also a. Finally, for pixel 2, a leave event for a is the event left from t_2, since this event has no parent, the state for pixel 2 is *no-state*. For k events in a stream, efficient data structures [17, 137] can support these queries with $\mathcal{O}(\log k)$ time. Consequently, visualization techniques in performance tools use a variation of this sampling approach.

6.1.1 Folding Operations

The proposed folding techniques target a visualization of aggregated streams. Figure 6.2 illustrates a folding of three streams (top) into a single aggregated stream (bottom). Consequently, the folded timeline visualization must both provide a state for each stream of event data, as for a regular timeline visualization, and must then combine the states for each stream into a single result state. Figure 6.2 illustrates this situation. For pixel 0 the sampled timeline visualization provides the sequence (a, a, b) of states for the three event streams. Generalizing this notion, for a sequence $\textit{seq} = (s_0, s_1, \ldots, s_n)$ (with $s_0, s_1, \ldots, s_n \in \mathcal{F}$) a selection operation *select* must select a result state for the folding. The following uses $\textit{count}(s \in \mathcal{F}, \textit{seq})$ to retrieve the occurrence count of s in the sequence *seq*. With respect to use case specific folding the following four operations are proposed:

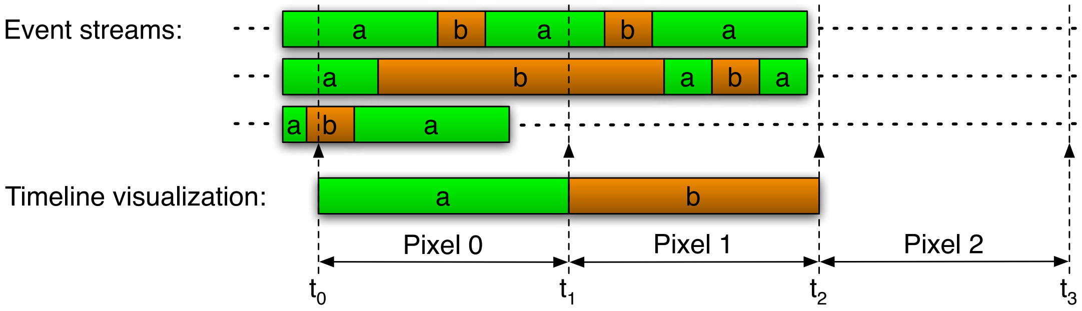


Figure 6.2: Illustration of the folding operation *select*$_{max}$ for three event streams.

$$
\mathit{select}_{max}(\mathit{seq}) := s_i \text{ where for all } k \in \{0,\ldots,n\} : \mathit{count}(s_k,\mathit{seq}) \leq \mathit{count}(s_i,\mathit{seq}) \tag{6.1}
$$

$$
\mathit{select}_{min}(\mathit{seq}) := s_i \text{ where for all } k \in \{0,\ldots,n\} : \mathit{count}(s_k,\mathit{seq}) \geq \mathit{count}(s_i,\mathit{seq}) \tag{6.2}
$$

$$
\mathit{select}_{diff}(\mathit{seq}) := \begin{cases} \mathit{no\text{-}state} & \text{if for all } k \in \{0,\ldots,n\} : s_0 = s_k \\ \mathit{select}_{min} & \text{otherwise, for all } k \in \{0,\ldots,n\} : s_k \neq \mathit{no\text{-}state} \end{cases} \tag{6.3}
$$

$$
\mathit{select}_{idle}(\mathit{seq}) := \begin{cases} \mathit{no\text{-}state} & \text{if for all } k \in \{0,\ldots,n\} : s_k = \mathit{no\text{-}state} \\ \mathit{select}_{max} & \text{otherwise, for all } k \in \{0,\ldots,n\} : s_k \neq \mathit{no\text{-}state} \end{cases} \tag{6.4}
$$

The first operation *select*$_{max}$ is a self-suggesting operation that selects the most frequent state from the sequence. The illustration in Figure 6.2 matches this operation. For pixel 0, it uses state a since this state occurs twice in the input sequence for the selection operation. For pixel 1, both b, a, and *no-state* are valid selections, since each occurs once in the input sequence $(a, b, \mathit{no\text{-}state})$. To resolve such situations a total order $<$ on $\mathcal{F}$ is applied to select the smallest state according to $\mathcal{F}$, e.g., b in the example. Finally, for pixel 2, the operation selects *no-state* from the sequence $(\mathit{no\text{-}state}, \mathit{no\text{-}state}, \mathit{no\text{-}state})$.

6.1.2 Use Case Specific Folding

As the list of introduced selection operations already suggests, some selection operations particularly target specific use cases.

Dominating Behavior The operation *select*$_{max}$ represents dominant behavior across all folded streams and provides a visualization that captures this behavior. This operation provides a general overview of an application's execution and is suitable as default folding operation.

Outlier Detection A common task in performance optimization is the identification of outliers that exhibit different behavior than other threads or processes of a parallel program. Such outlier behavior usually causes wait time of other threads/processes at a subsequent synchronization primitive. Consequently, a common task is the removal of such outlier behavior. In such a situation the operation *select*$_{min}$ immediately highlights outlier behavior across a set of processes or threads. A comparison of the folding result of this operation to the result of *select*$_{max}$ then highlights where outliers exist, as well as which state is dominant for other streams.

The selection operation *select*$_{diff}$ simplifies this comparison further. It replaces states in which all streams exhibit the same state by using *no-state* as their result. This extension indicates regions in time where streams behave equally and highlights regions where behavior diverges. For these differing regions, the operation selects the outlier behavior.

Accelerator Utilization Utilization statistics are a common starting point to evaluate the effectiveness of hybrid applications that utilize GPUs or other accelerators. Since modern accelerators have multiple hardware device streams that can concurrently execute kernels (functions executed on the device), idle times of accelerators exist if no active kernel executes. Timeline folding can effectively support this analysis. The operation $select_{idle}$ serves this purpose and highlights whether any kernel executes on an accelerator device. The operation behaves like $select_{max}$, except that it only returns *no-state* when all input streams are idle. Thus, the operation correctly considers the idle state for accelerator devices.

6.1.3 Time Complexity and Scalability

For k input streams, the four proposed folding operations require a time complexity of $\mathcal{O}(k)$. Consequently, visualizing a folded stream is as expensive as visualizing its input streams without folding. At the same time, the folding allows the visualization of more streams overall, thus increasing the scalability that the timeline visualization provides.

Additionally, by distributing the analysis for the folding, all four proposed operations can be calculated hierarchically. Analysis processes, e.g., as in the parallel analysis of VampirServer [17], can process a subset of the streams that form a folding operation's input. Each analysis process computes a vector of the states that occur across its streams and associates a count with each state to denote how often the state occurs in the streams. For k streams this yields a vector of l occurring states. According to experience for application state analysis [6], $l \ll k$ is expected even for large k, i.e., that streams often exhibit similar states. When multiple analysis processes provide their state-count vectors for a single folding, their vectors can be combined by adding occurrence counts for matching states. This operation is hierarchical, i.e., it maps to a master-slave architecture as in VampirServer as well as to Tree Based Overlay Networks [122] (TBON). A hierarchical processing for $select_{min}(seq)$ and $select_{diff}(seq)$ can additionally use intermediate state-count vectors of size 1 and 2 respectively, without loss of information. These vectors would only include the state with the minimal occurrence count and in the case of $select_{diff}(seq)$ an additional element to store whether all states are equal or not.

6.1.4 Case Studies

This section showcases the introduced folding functionality and the resulting analysis capabilities in detail with two applications.

PIConGPU

The first application analysis provides an example for the use case *Accelerator Utilization*. The analyzed application PIConGPU [20] computes a fully relativistic Particle-In-Cell (PIC) algorithm, which is widely used in computational plasma physics. Additional to MPI-based parallelism on the compute node level, PIConGPU utilizes graphics processing units (GPUs) to compute large-scale plasma simulations. To interface with the GPUs, PIConGPU uses the CUDA programming interface.

Figure 6.3 depicts the starting point of the analysis. It shows a complete run of PIConGPU. The timelines in the figure depict individual processes with respective labels on the left side. The application has been executed on four MPI processes controlling one CUDA device (GPU) each. Each CUDA device provides a set of streams to efficiently handle potentially concurrent activities. The colors in Figure 6.3 represent different functions. CUDA kernels appear in blue, CUDA runtime functions in purple, MPI functions in red, and application functions in green. Communication between processes is indicated by black lines. In the beginning of the simulation, shown in the left half of Figure 6.3, PIConGPU runs initialization code. In the right half of Figure 6.3, PIConGPU performs the computations of the simulation. The execution of CUDA kernels is depicted in blue. At the far right in Figure 6.3, the simulation computations are done and finalization code is executed.

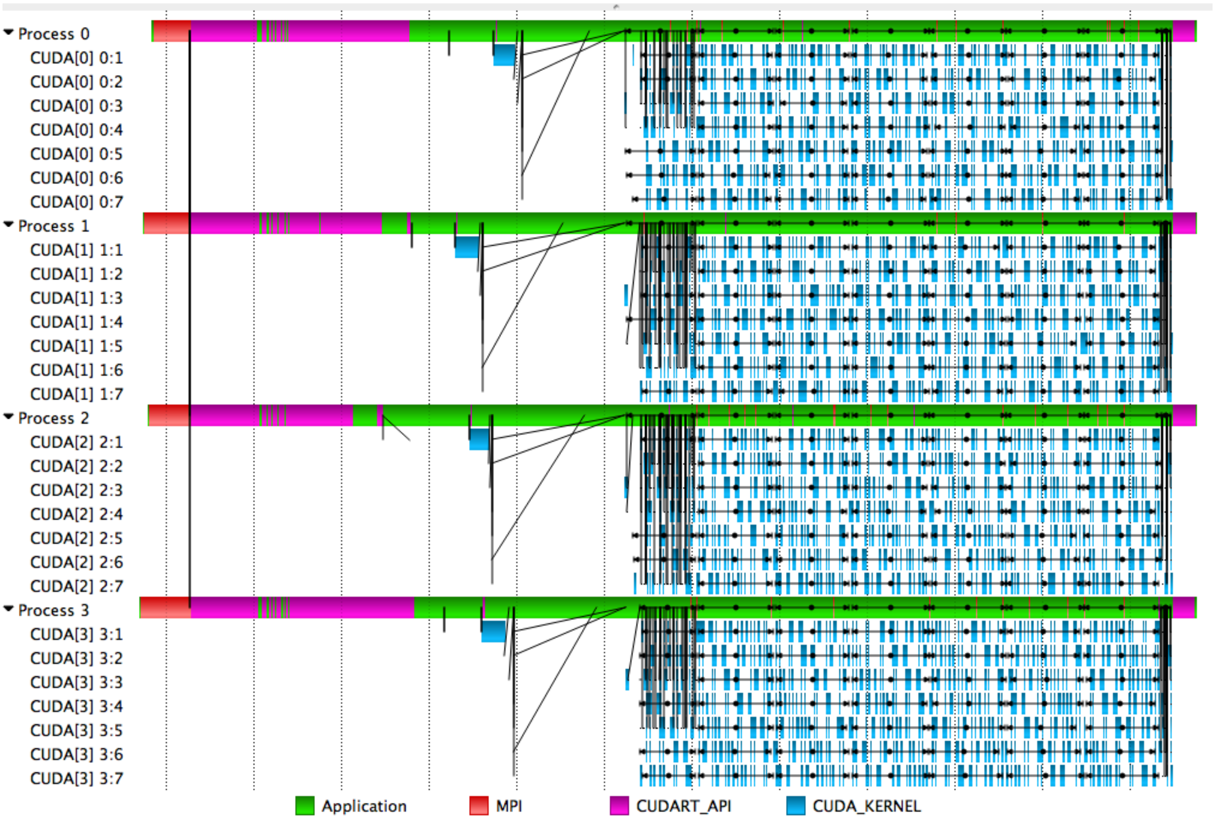
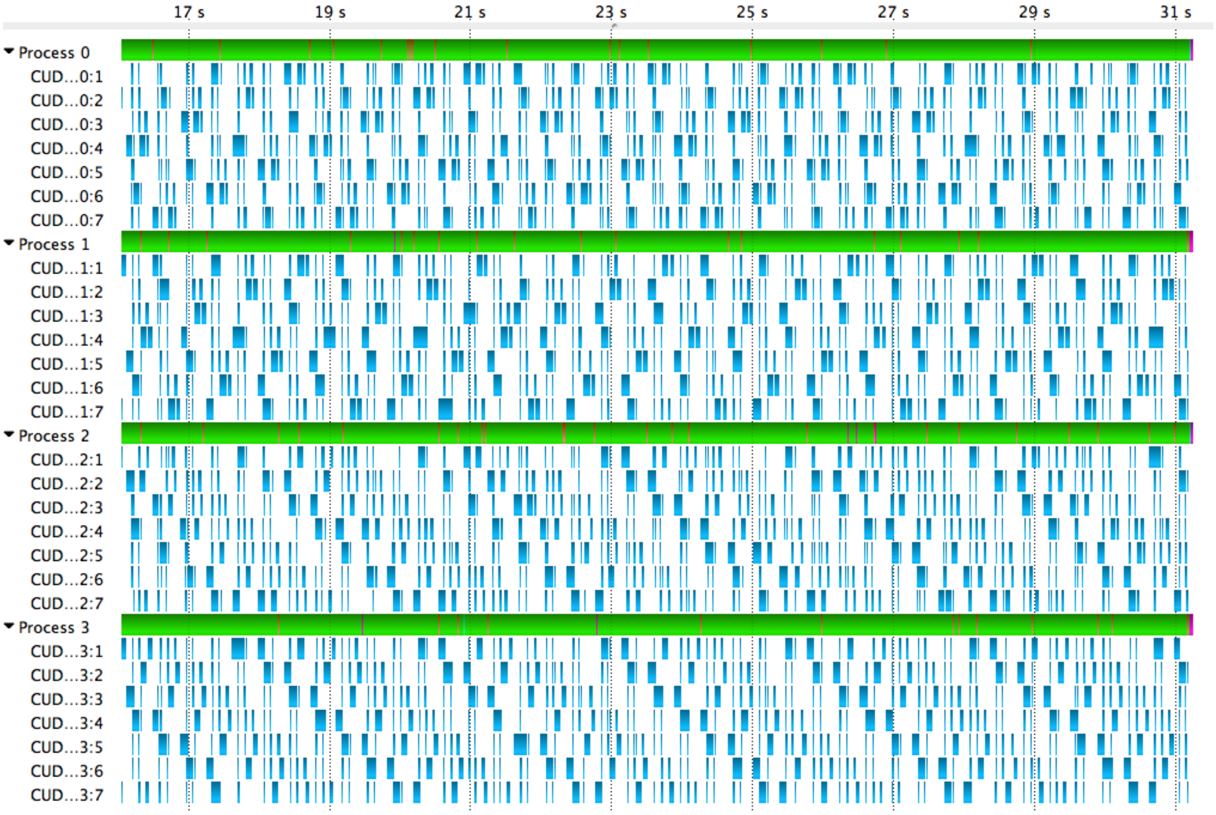


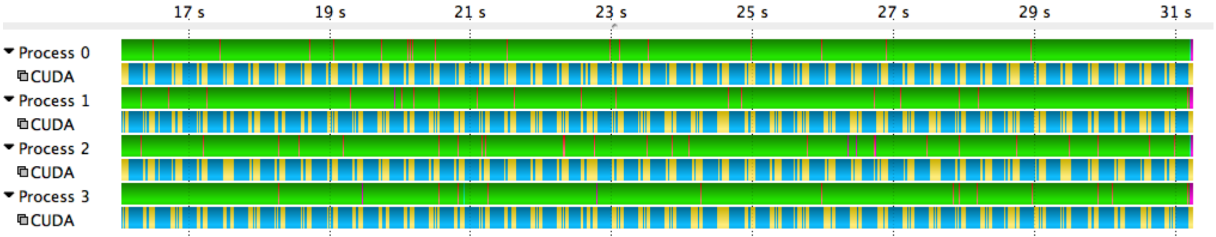
Figure 6.3: Vampir timeline showing a run of the PIConGPU CUDA application.

Using common timeline visualization techniques, as shown in Figure 6.3, the identification of idle times on the GPU, as well as an evaluation of the GPU device utilization, is very challenging. For each CUDA device in Figure 6.3, kernels are scheduled across seven device streams. Only when no kernel is active on all of the seven streams, the respective CUDA device is idle. However, the scheduling of kernels across all available device streams produces visual clutter in the timeline visualization. This phenomenon is visible in the blue areas in Figure 6.3. It significantly increases the required effort to identify inefficiencies and idle times in CUDA application executions. To alleviate the analysis effort the new folding functionality is applied. Using the $select_{idle}$ operation, all streams of a CUDA device are folded into a new timeline. For the corresponding time interval of each visualized pixel of the new timeline, this operation identifies the most prominent activity (highest time share) across all device streams. Idle time (*no-state*) is only selected when no CUDA device stream computes any kernel. That way, the new timeline always shows the most important activities and visualizes idle times only when all device streams are unused. This reduces the required screen space for displaying CUDA activities and inhibits visual clutter. The scattered idle areas (shown in white in Figure 6.3) are reduced, since all concurrent activities on the device are aggregated into one timeline. Thus, idle areas are only visualized when the whole device is unused. Figure 6.4 shows the folded timelines for the CUDA devices of the PIConGPU run. For ease of presentation, the visualization of communication is disabled in the figure. Additionally, in the folded timelines, areas exhibiting GPU idle times are colored in yellow. Figure 6.4(a) and Figure 6.4(b) contrast the plain vs. folded visualization.

While the plain visualization results in visual clutter, the folded visualization clearly reveals the computational structure of the application. Individual iterations become visually detectable. The folding techniques additionally reduce the amount of visualized timelines. Hence, the introduced techniques reduce visual complexity to facilitate easy analysis.

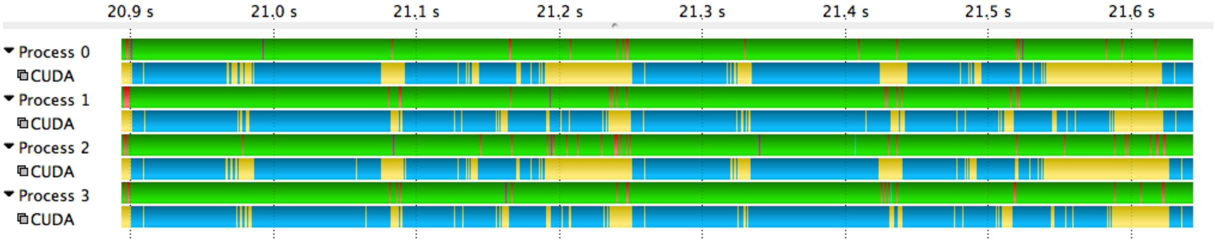


(a) All device streams are visualized independently leading to visual clutter.

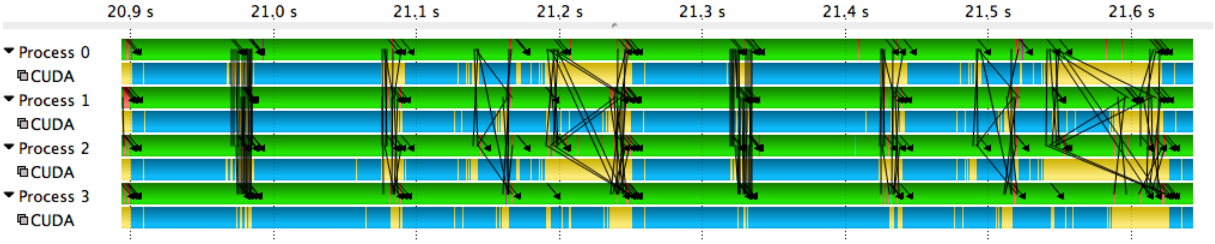


(b) All respective device streams are folded resulting in an easily visible computational structure.

Figure 6.4: Timeline grouping applied to four CUDA devices running PIConGPU.



(a) Zoom to two iterations. Device idle times are clearly visible as yellow areas. The uneven yellow areas at the end of each iteration indicate load imbalances between devices.



(b) The additionally visualized communication (black lines) shows that device idle times correlate with communication phases between host processes and CUDA devices.

Figure 6.5: Visualization of iterations in PIConGPU using folded CUDA streams.

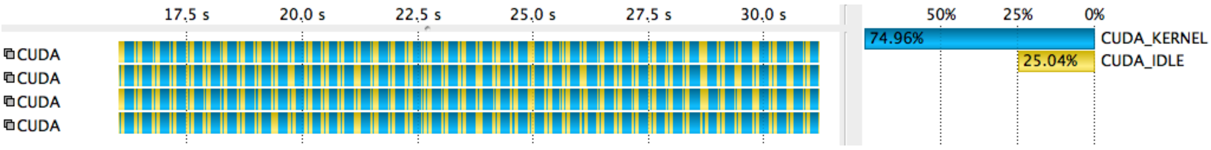


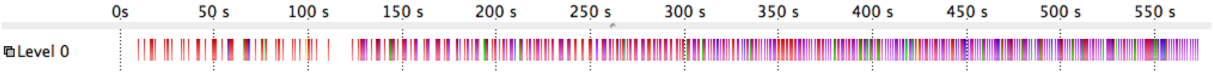
Figure 6.6: Statistic showing the CUDA device utilization.

Besides valuable overviews of an application's behavior, timeline folding techniques enable advanced analysis capabilities. Figure 6.5 shows an in-depth analyses of the computational structure of PIConGPU. Device idle times are easily visible as yellow areas and correlate to communication phases between host processes and their CUDA devices.

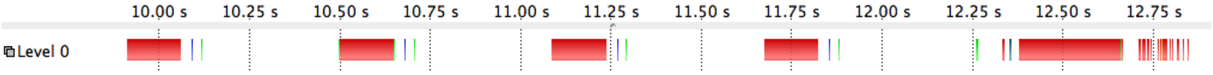
Based on the folded timelines, it is possible to provide additional summary information. Figure 6.6 gives an example. The figure depicts the folded timelines of all four GPU devices on the left. The right side depicts summary information according to the timelines. This summary represents the GPU device utilization. During the compute iterations the GPU device exhibits a utilization of 75%. The introduced folding techniques facilitate the computation of this information. Without folded timelines, complete device idle times cannot be differentiated from individual stream idle times. An evaluation of the GPU device utilization is essential for efficient optimization workflows.

COSMO-SPECS

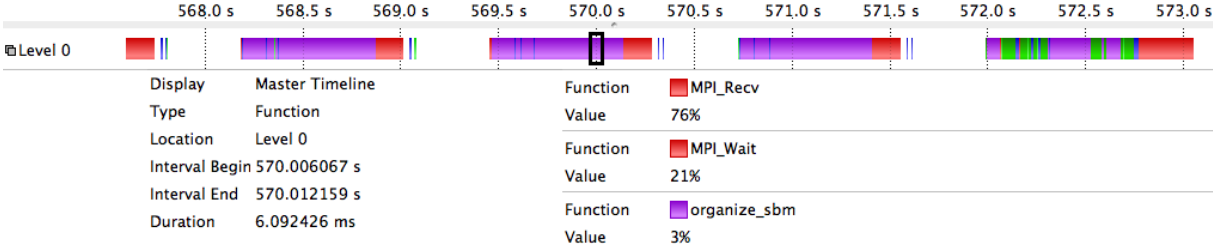
The second application analysis provides an example for the use case *Outlier Detection*. This case study analyzes the execution of a weather forecast code [48]. The code couples two models, COSMO and SPECS, for more accurate simulation of cloud and precipitation processes. COSMO is the regional weather forecast model originally developed at the German Weather Service (DWD). SPECS is a detailed cloud microphysics model developed at the Leibniz Institute for Tropospheric Research (IfT). SPECS computes detailed interactions between aerosols, clouds, and precipitation.



(a) Visualization of the full application execution. The colored portions indicate areas in that processes execute different functions. The frequency of such areas increases over time.



(b) Visualization of the first iteration. Red areas indicate the exchange of simulated values between processes. The workload is still balanced and computations are executed in the white areas.



Display Master Timeline
Type Function
Location Level 0
Interval Begin 570.006067 s
Interval End 570.012159 s
Duration 6.092426 ms

Function 🟥 MPI_Recv
Value 76%

Function 🟥 MPI_Wait
Value 21%

Function 🟪 organize_sbm
Value 3%

(c) Visualization of the last iteration. Large purple areas are visible. In that areas most processes are waiting and only a few processes are computing.

Figure 6.7: Visualization of the folded timelines of a COSMO-SPECS weather forecast code run.

COSMO employs a 2D (horizontal) decomposition into $M \times N$ processor domains and runs without dynamic load balancing. SPECS uses the same data structures and decomposition as the COSMO model and only replaces its cloud microphysics. Compared to COSMO, the SPECS calculations are significantly more compute intensive. During the execution, SPECS introduces large load imbalances, since its computational cost heavily depends on the presence of cloud particles in the grid cell. Thus, the layout of clouds in the application domain determines the local work.

Figure 6.7 demonstrates the $select_{diff}$ folding operation applied to a COSMO-SPECS code run on 100 processes. First, Figure 6.7(a) shows the complete application run. The timeline depicts COSMO activities in green, SPECS activities in purple, and MPI functions in red. The folded timeline visualizes areas that exhibit dissimilar executions across the processes. Such areas are interesting as they indicate potential imbalances. In case of load imbalance, some processes are waiting in barrier calls, while others are still executing computations. White areas in the folded timeline, however, indicate similar work across all processes. As visible in Figure 6.7(a), the load imbalances increase throughout the application execution. The first iteration, shown in Figure 6.7(b), takes less than 3 seconds. The workload is still balanced and no waiting takes place. A different situation is shown in the last iteration, see Figure 6.7(c). This iteration takes more than 5 seconds. Visible in Figure 6.7(c) are large purple areas. In these areas, only the minority of processes execute computations.

Detailed information is visible at the bottom of Figure 6.7(c). There, for a selected portion of the folded timeline (marked by the black rectangle in Figure 6.7(c)), in-depth summary information is given. The summary lists all executed functions in the selected time range, as well as the fraction of time that each function executes within the range. As shown in the bottom of Figure 6.7(c), only 3% of the time is spent in actual calculations. While almost all runtime (97%) is spent for MPI communication or synchronization. These values indicate that almost all processes wait for one or two processes to finish their calculations.

Like shown in this example, the introduced folding techniques allow an easy evaluation of the runtime behavior of an application execution. The techniques intuitively visualize the load imbalance of the application in only one timeline.

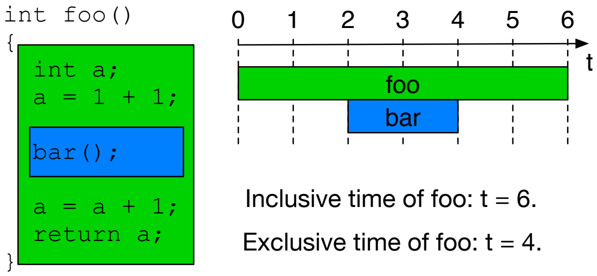


Figure 6.8: Inclusive vs. exclusive time of a function invocation.

6.2 Detection and Visualization of Runtime Imbalances

The identification of performance bottlenecks in parallel applications is a challenging task, as data sets from parallel performance measurements are often large and overwhelming. This section presents an effective and lightweight approach to facilitate visual analysis of performance data. The approach automatically identifies and highlights runtime imbalances in an application run. For parallel applications that must strive to achieve good load balance, this metric efficiently highlights a wide range of load balancing problems.

The approach applies to traces of a parallel application. Profiles do not suffice, since information on runtime variations of application functions is required. Additionally, the analysis visualization targets timeline visualizations that also apply to traces only.

To analyze runtime imbalances the following three steps are performed:

1. Identification of time dominant functions that are used to partition the complete run into small segments[1],
2. Computation of runtime imbalances between these function invocations (segments), and
3. An intuitive visualization to present the overall result.

Three analysis case studies demonstrate the effectiveness of the approach by locating performance bottlenecks in the application runs.

Since the runtime imbalance analysis directly identifies the location of the performance problem, it enables focused subsequent analysis to find the underlying root-cause of the problem. Effectively, the introduced method supports the performance analyst in helping him to focus on performance problems faster.

6.2.1 Identification of Time Dominant Functions

Identifying reoccurring parts of an application is beneficial to detect runtime imbalances during the application run. Reoccurring parts allow to separate an overall execution into multiple segments. Then, the runtimes between segments can be compared to detect imbalances. The first step of the analysis is a selection of suitable segments to highlight runtime imbalances. Since parallel applications usually execute functions repeatedly, as they are being called in loops, such a function is a suitable choice for the segments. To decide which particular function to select as segments, their *inclusive* times are considered.

When measuring a function's invocation, there are two options to report the function's duration: *inclusive* and *exclusive* time. Figure 6.8 depicts the difference between inclusive and exclusive time. Inclusive time represents the complete duration of a function's invocation, from initially entering to finally leaving

---

[1] As invocations of the time dominant function are used as segments, the inclusive time of the dominant function invocation equals the respective segment duration.

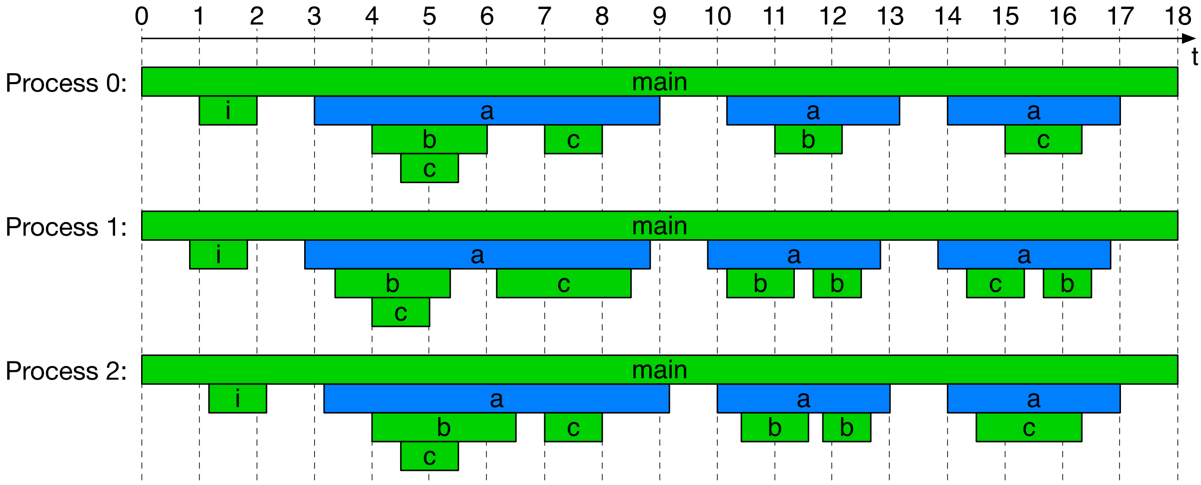


Figure 6.9: For the runtime imbalance analysis one time dominant function is selected. The selected function needs to have a possibly high inclusive time and an invocation count higher than the number of processes. In this example function a fulfills these criteria.

the function call. This time also includes the time spent in sub-functions. The inclusive time for function `foo` in Figure 6.8 starts with entering `foo` ($t = 0$) and stops when `foo` is left ($t = 6$). The inclusive time includes the sub-call to function `bar` and is 6 in the example. The exclusive time, on the contrary, represents only the amount of time spent directly inside the respective function's invocation, excluding sub-functions. The exclusive time of function `foo` in Figure 6.8 starts with entering `foo` at $t = 0$, excludes the sub-call of function `bar` ($t = 2$ to $t = 4$), and ends with leaving of `foo` at $t = 6$, i.e., it is 4 in the example.

Considering the inclusive time to detect dominant functions is reasonable, since it includes the overall performance impact of a function. Figure 6.9 illustrates an example that uses three processes and the functions `main`, `i`, `a`, `b`, as well as `c`. A time dominant function should have a considerable impact on the total application runtime. For that the aggregated inclusive time of each function is considered. Selecting the function with the highest aggregated inclusive time, however, is not a good choice for a dominant function. In the example, this would be the function `main` (54 time steps). Such top call-level functions may be suitable to compare the runtime between processes, but they are not suited to analyze variations over the runtime. Additionally, they provide no segmentation of the overall runtime. Thus, just maximum aggregated inclusive time is not a good selection criterion alone.

As a consequence, the selected time dominant function should exhibit a high aggregated inclusive time, but also feature a higher number of invocations. In the example of Figure 6.9, the function with the highest inclusive time share is `main`. The function `main` is called three times on the three processes in total. Thus, top call-level functions like `main` have exactly as many invocations as there are parallel processing elements. Thus, a time dominant function f is defined as follows:

- For p processing elements, f is invoked at least $2p$ times and there exists no other function that satisfies this condition and has higher aggregated inclusive time.

In the example, the function with the second highest inclusive time share is `a` (36 time steps). Function `a` is called nine times on three processes, i.e., it satisfies the invocation count restriction. Hence, `a` is the time dominant function for the example.

While this selection criterion is heuristic, the subsequent use cases demonstrate that it provides functions that represent iterative application behavior well in practice. Comparing the runtime of such functions, and thus, analyzing the durations of iterations provides a solid foundation to detect a wide range of performance hot spots.

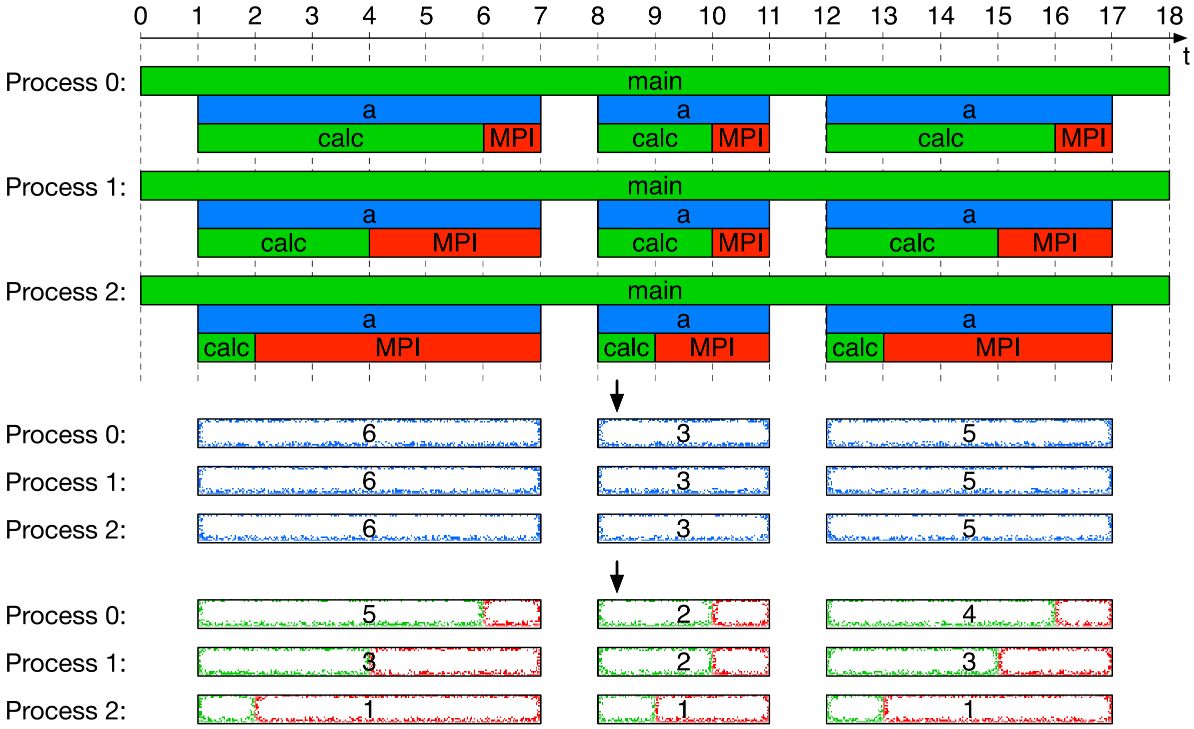


Figure 6.10: Calculation of performance variations. First, the calculation of segment durations (inclusive time of function a in this example) is shown in the middle. Then, the subtraction of synchronization time from the segment durations to compute the synchronization oblivious segment time (SOS-time) is shown at the bottom.

6.2.2 Analysis of Runtime Imbalances

Using the selected time dominant function allows to compare the runtimes of its individual invocations. This comparison highlights shifts in runtime behavior over time and identifies runtime imbalances. For instance, if an application runs gradually slower, the inclusive time of a good dominant function will usually increase as well over time. Also, outlier-iterations with exceptionally long runtime will impact the inclusive time of a dominant function.

The comparison of inclusive times of dominant functions alone has a shortcoming: In many applications, synchronization calls are also included in iterative function behavior. Figure 6.10 illustrates this behavior with the communication function named `MPI`. As previously highlighted, the introduced heuristic selects `a` as the dominant function for this example. The direct comparison of the inclusive times of `a`'s invocations yields the results shown in the middle of Figure 6.10. The iterations in the middle (duration of 3) are twice as fast as the first iteration (duration of 6). This analysis already detects performance variation across iterations. However, if some iterations show a differing behavior from others, there are often only a few processing elements that cause this behavior. With the direct comparison of dominant function durations, it is not possible to identify the processes that cause the differences. The reason is often, that synchronization between processes is included in the iteration code. For instance, considering the example in Figure 6.10. In each iteration all processes first run a calculation (function `calc`) and then call an MPI [103] synchronization operation, e.g., `MPI_Barrier` (indicated as function `MPI`). *Process 2* in Figure 6.10 completes its calculations faster than *Process 0*. As a consequence, the MPI synchronization call in *Process 2* runs longer, as this process is waiting for *Process 0* to finish. The difference of the calculation part between the processes remains hidden, since the synchronization wait time is included in the inclusive time for the dominant function. Therefore, the calculation method requires an adjustment to cover performance variations between segments. Instead of directly using each segment's duration, subtraction of any synchronization time from its inclusive time is necessary. The synchronization time can be easily detected if the application uses common parallelization libraries like MPI [103], OpenMP [111], or similar. In such case, each segment is checked for synchronization operations, e.g., `MPI_Wait`, `MPI_Reduce`, or `omp barrier`, and their runtime is subtracted from the inclusive time of the dominant functions. This adapted segment time is referred to as *synchronization oblivious segment time* (SOS-time) and serves as measure for runtime imbalances. Figure 6.10 (bottom) depicts this process for the example. The SOS-times correctly reflect the performance differences between the processes. For instance, for the first iteration in Figure 6.10 the SOS-time of *Process 2* shows 1 compared to a SOS-time of 5 for *Process 0*, i.e., it highlights the computational load imbalance in the first iteration.

6.2.3 Visualization of Runtime Imbalances

The last step of the introduced approach is the visualization of the SOS-times. Therefore, the analysis methods have been implemented in the Vampir performance analysis framework [18]. To achieve an intuitive visualization, commonly used timeline views of Vampir are overlaid using the SOS-times as values for a new metric counter. For the visualization metric values are encoded with a color-coded scale. Blue—cold—colors indicate short durations, whereas red—warm—colors indicate long durations. Figures 6.11(b), 6.12(b), 6.12(c), and 6.13(b) in the subsequent case studies present examples of this visualization.

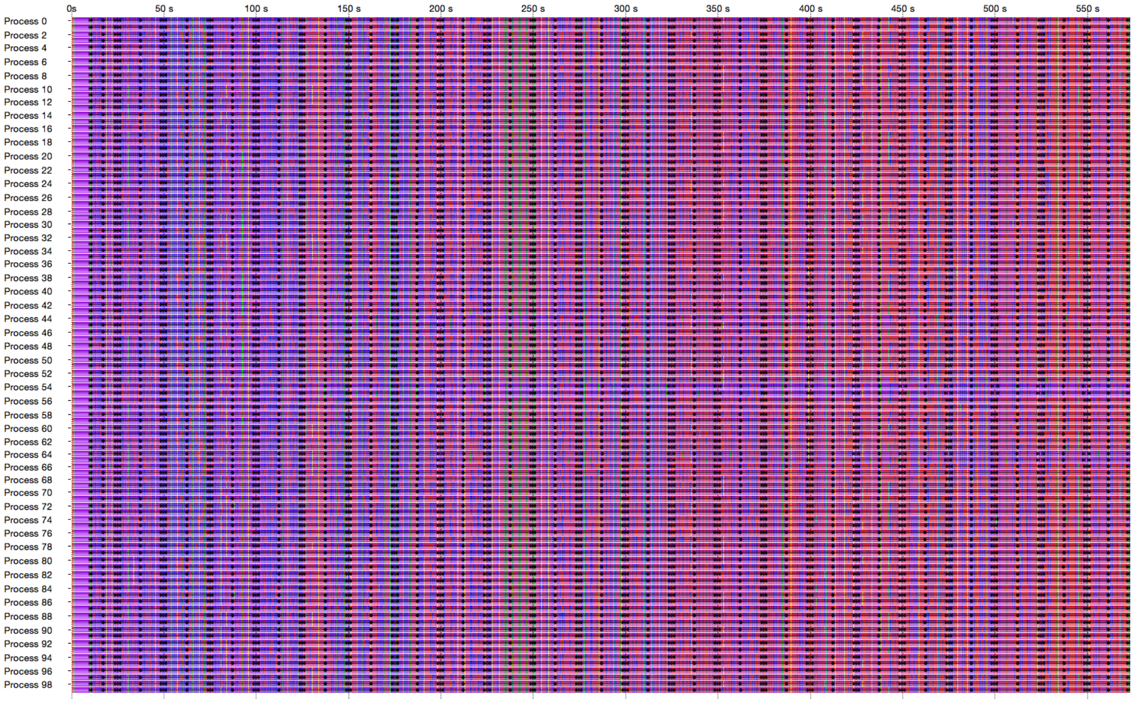
6.2.4 Case Studies

This section demonstrates the applicability of the introduced approach with three use cases. The analysis methods have been implemented as part of the Vampir [18] analysis and visualization toolkit. In the following, trace files with known performance problems have been analyzed to demonstrate the capabilities of the runtime imbalance analysis.

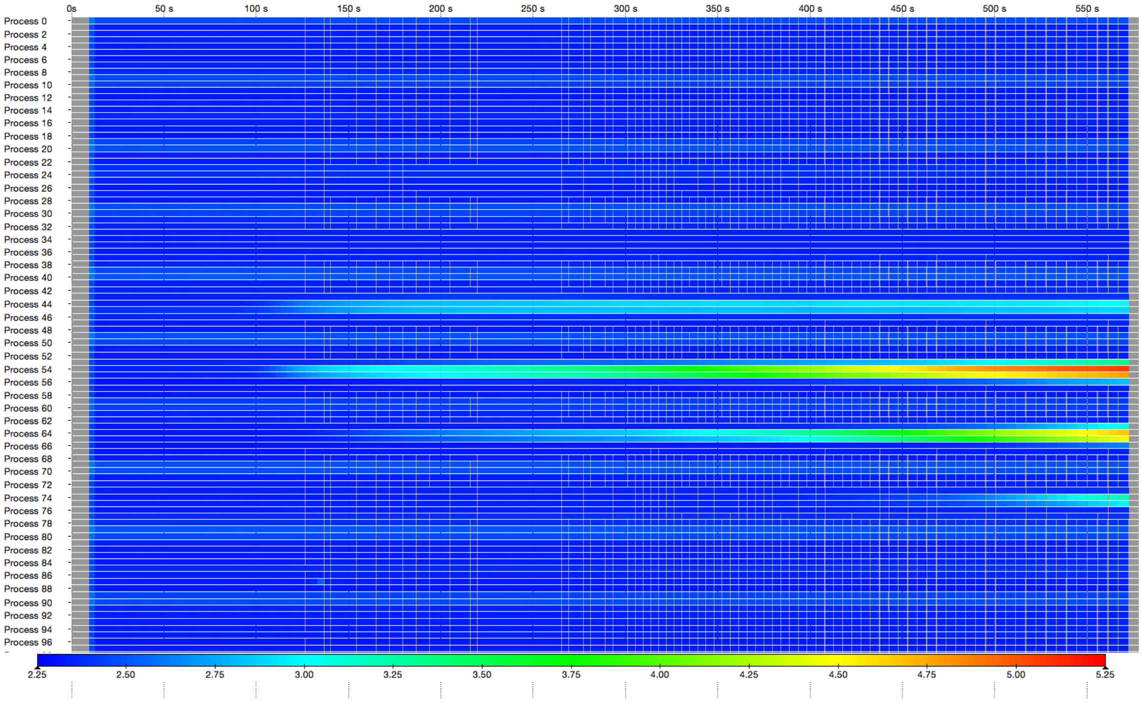
Load-Imbalance - COSMO-SPECS

The first case study is an analysis of the execution of a weather forecast code [48]. The code couples two models, *COSMO* and *SPECS*, for a more accurate simulation of cloud and precipitation processes. COSMO is the regional weather forecast model originally developed at the German Weather Service (DWD). SPECS is a detailed cloud microphysics model developed at the Leibniz Institute for Tropospheric Research (IfT). SPECS computes detailed interactions between aerosols, clouds, and precipitation.

Figure 6.11(a) shows the Vampir timeline visualization of the overall application run. The execution under study uses 100 MPI processes that Vampir individually represents with horizontal bars. The colors then identify the currently active functions across the overall execution time. Red identifies MPI activities, purple SPECS activities, green COSMO activities, and yellow highlights the coupling between the two models. Compared to COSMO, the SPECS calculations are significantly more compute intensive. Therefore, purple areas—SPECS code—dominate the application run. The execution of COSMO code—green areas—is barely visible in Figure 6.11(a). This behavior is caused by the computational demand of the underlying physics. However, Figure 6.11(a) also shows another trend. Throughout the execution, the fraction of MPI—red areas—increases. Up to a point where MPI activities are dominating towards the end of the run. The described heuristic selects a time dominant function whose occurrences represent individual iterations. Comparing the plain inclusive time of this function (segment durations) shows gradually increased durations towards the end of the application run.



(a) Timeline visualization of COSMO-SPECS running on 100 processes, showing increasing MPI durations (red areas) over time.



(b) Runtime variation analysis result. Several processes (middle) exhibit higher runtimes (SOS-time) in their dominant function.

Figure 6.11: Analysis of the COSMO-SPECS weather forecast code. (a) shows the timeline visualization. (b) shows the analysis results.

To find the cause of the degrading performance, the SOS-time is applied. Figure 6.11(b) presents this metric and highlights that only a few processes (*Process 44*, *45*, *54*, *55*, *64*, *65*) exhibit increases in this metric. Particularly *Process 54* needs more time than any other process for its calculations.

The reason for this behavior is a static decomposition of the computational grid. COSMO employs a two dimensional (horizontal) decomposition into $M \times N$ domains and applies no dynamic load balancing. SPECS uses the same data structures and decomposition as the COSMO model, but instead computes cloud microphysics. During the execution, SPECS introduces large load imbalances, since its computational cost heavily depends on the presence and size distribution of various cloud particle types in the grid cell [83]. Thus, the layout of clouds in the application domain determines the local work. In other words, while *Process 54* still performs cloud microphysics calculations, the other processes idle while waiting for it to finish. A solution to this performance problem is to introduce dynamic load balancing for the SPECS model.

The introduced analysis and visualization method correctly represents the performance situation of the application. By following the high—red—values the analyst is pointed directly to the cause of the performance bottleneck.

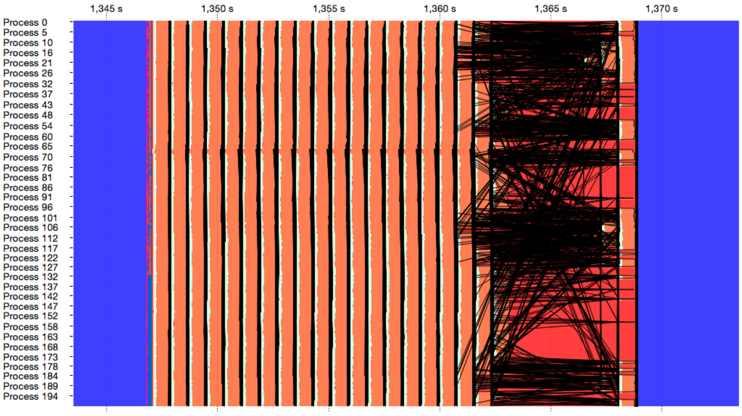
Process Interruption - COSMO-SPECS+FD4

This case study analyzes an extended version of the previous weather forecast code [48]. In this version the developer has added a dynamic load balancing mechanism, called FD4 [83], to the SPECS model. As described in the first case study, the high computational demand of the SPECS code, combined with its high dependence on local workload—presence of cloud particles in the domain—demand a dynamic load balancing for efficient computation.

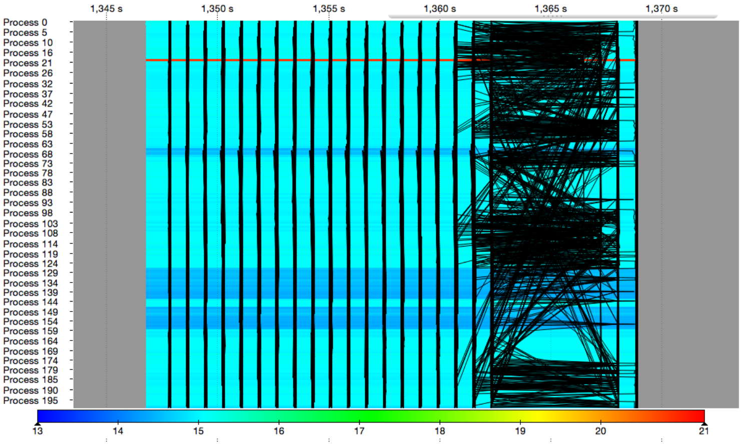
The application run under study uses 200 MPI processes. The initial analysis—not shown—detected that only a few iterations behaved differently and exhibited larger durations than other iterations. The goal of this study is to detect the reason for these slow iterations. Therefore, the analyst used a second measurement run to only record slow iterations. For normal iterations the analyst discarded the tracing data. Figure 6.12(a) shows the timeline visualization of one slow iteration. Again, different colors represent different activity types. Red relates to MPI code, blue indicates areas where performance data was dropped, while orange and white areas relate to SPECS activities. The black lines indicate MPI messages sent from one process to another. The runtimes of COSMO and FD4 are so short compared to SPECS that these areas are not directly visible in Figure 6.12(a). Looking at the behavior in Figure 6.12(a) shows that one SPECS timestep near the end of this iteration takes significantly longer than the others. Especially, increased MPI wait time—more red areas—and higher message transfer times—longer black lines—indicate this behavior. However, the reason causing the slower timestep in this iteration is not immediately visible.

Using the runtime imbalance analysis allows to guide the analyst to the cause of this performance problem. Figure 6.12(b) shows the result of the analysis. The red line in the figure highlights a high SOS-time for *Process 20*. Thus, the performance problem is caused by longer computation time of *Process 20*. To find the exact place of the performance problem, a refinement of the analysis granularity is possible by adapting the dominant function. Choosing a function with a smaller inclusive time results in a more fine-grained segmentation. This option is beneficial to track the origin of a performance problem. Figure 6.12(c) shows the result of the finer segmentation. This figure clearly shows a single function call—red line—that runs significantly longer than all other invocations—blue lines—of this function. A closer inspection of *Process 20* shows that this single function call exhibits a low number of total assigned CPU cycles (measured with the PAPI counter PAPI_TOT_CYC [130]). Therefore, *Process 20* has been interrupted exactly during the execution of this function's invocation. The cause for the interruption is assumed to be an influence from the operating system.

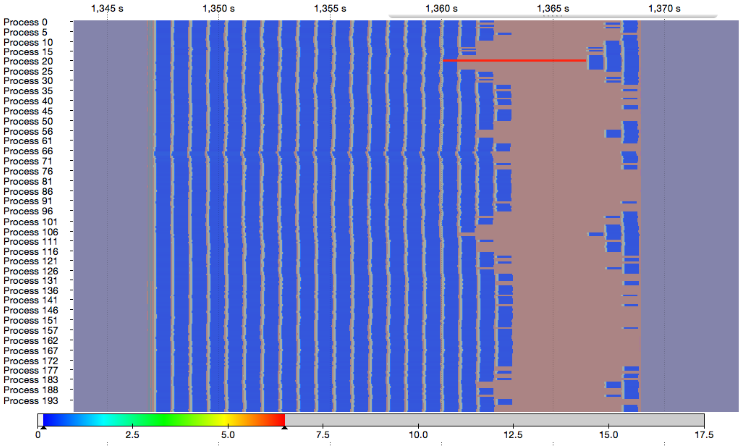
The runtime variation analysis directly points the analyst to the performance bottleneck. Without an extended search, the subsequent analysis can be focused directly on the hot spot and quickly reveal the cause of the performance problem.



(a) Timeline visualization of COSMO-SPECS+FD4 running on 200 processes.



(b) Coarser runtime variation analysis result (SOS-time). Especially *Process 20* exhibits a high duration in its dominant function.



(c) Finer runtime variation analysis result (SOS-time). Using smaller segments sizes allows direct identification of the one function invocation that causes the performance degradation.

Figure 6.12: Analysis of a COSMO-SPECS+FD4 application run. Displayed is just one iteration. (a) shows the timeline visualization. (b) (coarser segments) and (c) (finer segments) show the variation analysis results using SOS-time.

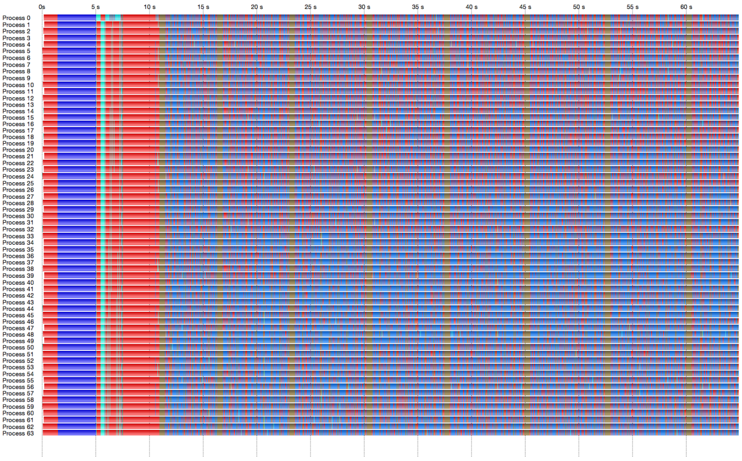
Floating Point Exceptions - WRF

This case study analyzes an application run of the *Weather Research and Forecasting model* (WRF), with a standard benchmark case (12km CONUS) [127]. The application under study uses 64 MPI processes. Figure 6.13(a) presents Vampir's basic timeline visualization. Red areas relate to MPI activities. Blue areas relate to computations of the *dynamical core* of WRF. These parts of the application compute for instance density, temperature, pressure, and winds in the atmosphere. Brown areas relate to the *physical parameterization* calculations of WRF. For instance clouds, rain, and radiation are computed in these parts.

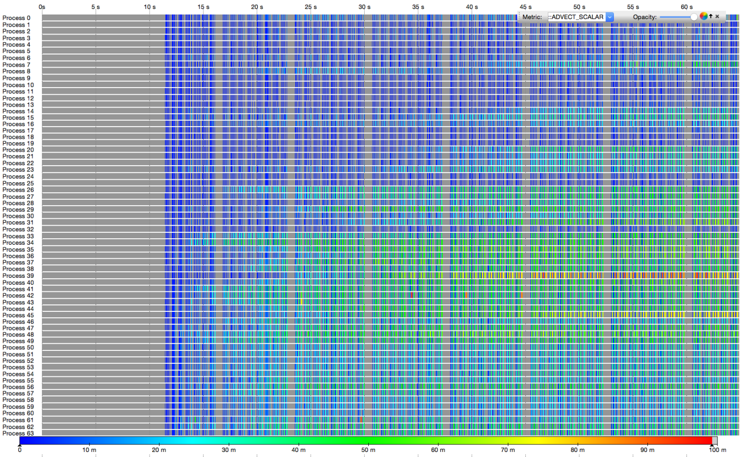
In the early parts of the run (left of Figure 6.13(a)) the application executes model initialization and I/O activities that take about 11 seconds. Afterwards, the actual iterations begin. Basic Vampir statistics for the iterations highlight a 25% fraction of MPI activities, which highlights a noticeable parallelization overhead. The timeline view in Figure 6.13(a) does not present an immediate cause for this overhead.

Figure 6.13(b) visualizes the SOS-time of the dominant function of the application run. The segments located in the lower right part in the figure highlight increased durations. Particularly *Process 39* exhibits higher durations than the other processes. A closer inspection supports this immediate result: *Process 39* computes slower and causes the other processes to wait. Based on hints that floating point intensive functions compute slower, the analyst found that a high number of floating point exceptions slows down *Process 39*. For validation Figure 6.13(c) shows the color-coded values of the counter `FR_FPU_EXCEPTIONS_SSE_MICROTRAPS`. As shown in the figure, *Process 39* exhibits an exceptional high number—red areas—of floating point exceptions. Moreover, comparing Figure 6.13(b) and Figure 6.13(c) shows that the results of the counter `FR_FPU_EXCEPTIONS_SSE_MICROTRAPS` perfectly match the runtime imbalance analysis.

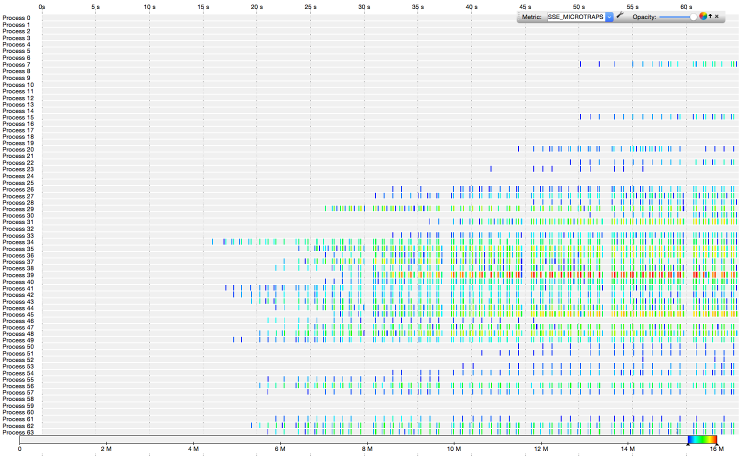
This shows that the introduced approach correctly depicts the application performance behavior. By following the runtime imbalance metric visualization, the analyst is guided closely to the performance issue. If necessary, focused subsequent analyses then reveal the root cause of the performance problem.



(a) Timeline visualization of WRF running on 64 processes.



(b) Runtime variation analysis. Especially *Process 39* exhibits high SOS-times in its segments.



(c) Values of the counter `FR_FPU_EXCEPTIONS_SSE_MICROTRAPS`. *Process 39* shows a high number of floating point exceptions.

Figure 6.13: Analysis of a WRF application run. (a) shows the timeline visualization. (b) shows the variation analysis results and (c) shows values the floating point exceptions counter.

7 Conclusions and Future Work

This dissertation presents novel analysis techniques for structural and temporal comparison of parallel processes. In the context of this work, the term *process* relates to any type of processing element of a parallel application, such as an MPI process, a thread, or a CUDA stream. Measurement systems that record parallel application runs, at very high detail, save their measurement data in the form of application event streams. The detailed event-wise comparison of such streams currently requires manual inspection by the analyst, which is cumbersome and error-prone. This work presents methods that facilitate this comparison and provide powerful and intuitive analysis capabilities for the purposes of parallel performance analysis.

The first contribution of this work consists of methods for the pairwise comparison of processes. The methods are based on sequence alignment algorithms used in bioinformatics and allow structural comparison of event streams on the level of individual events. The key contribution is a hierarchical scheme for sequence alignment algorithms which enables a comparison of complete application event streams. Without the hierarchical scheme, the computational demand of such comparisons easily exceeds the capabilities of available computing hardware.

The second contribution of this work introduces analysis metrics that effectively exploit the advantages of the new structural comparison approach. These alignment-based metrics present intuitive overviews as well as detailed comparisons of performance characteristics between application processes. This type of analysis is suitable for comparing two application runs. Use cases include analyzing the impact of optimizations or studying the performance effects of different hardware. As such comparisons are common tasks when optimizing and analyzing parallel applications performance, both contributions fill a crucial gap in available performance analysis tools by adding efficient comparative analysis capabilities.

The third contribution is an approach for structural comparison of multiple event streams. First, a fast and scalable hash-based pre-clustering step identifies structurally similar processes. The contributed clustering algorithm groups large numbers of processes efficiently into a small number of clusters. Then, multiple sequence alignment methods analyze the event streams in a cluster for detailed structural similarities and differences. This work introduces several adaptions that enable multiple sequence alignment methods for the comparison of large numbers of event streams. Most notably, this work contributes a heuristic scheme accelerating the computation of the required guide-tree and a hierarchical scheme that augments the multiple sequence alignment algorithm. The presented comparison approach computes a so-called merged call tree, that combines the structural information of all compared event streams. The merged call tree provides a compact data structure with rich potential for performance analysis and visualization of parallel applications. Especially the merging of structural similarities and differences into one call tree allows a fast comparison of differences between large numbers of processes. Without the merged call tree, this task had been more challenging as related differences needed to be identified in all processes individually.

The fourth contribution of this work addresses visual scalability of performance data displays. It introduces folding strategies for event timeline visualizations, which add powerful methods for visual aggregation of event streams that help achieving visual scalability and support easy detection of performance issues, such as imbalances or accelerator usage inefficiencies. An additional technique automatically identifies and highlights several types of performance critical sections in an application run. This technique analyzes runtime imbalances throughout the application run and presents the resulting runtime variations in an intuitive visualization that guides the analyst to performance hot spots.

In summary, structural comparison of process event streams is a viable approach for analysis and comparison of parallel applications. The presented contributions add analysis techniques for large-scale application runs and advance scalable event visualization. The techniques have been applied to real-world applications and this work showed how they facilitate the identification of differences between code versions. Novel alignment-based metrics exposed differences that otherwise would have been hard or even impossible to find. Moreover, the developed techniques of this work help achieving scalable and useful trace visualizations for parallel applications. Therefore, this work contributes methods that provide users with a new level of detailed insight into the performance of their codes and will be of substantial help in optimizing them. Prototype implementations demonstrate the applicability of the presented methods. The introduced timeline folding techniques already reached production state and are included in the Vampir performance analysis tool set.

Future work includes refinement and additional enhancements to improve the developed comparison methods. The first step should exploit the potential for parallelization in all introduced alignment algorithms. Pairwise event stream comparisons do not depend on other event streams; thus, comparing runs with large process counts is embarrassingly parallel. Parallelization could considerably accelerate the computation time of the presented comparison approaches and help driving them towards production state. The next step should address the comparison information contained in the merged call tree data structure. It provides potential for the design of automatic analysis methods and use case specific visualizations. The merged call tree can serve as a comparison basis for automatically identifying outlier processes or evaluating the performance impact of detected differences. Finally, using the merged call tree analysis results, specific visualizations that emphasize either common performance behavior or highlight performance critical differences between processes will further improve tool support for performance analysis and comparison of parallel applications.

List of Figures

1.1 History of horizontal resolution and complexity in climate models 2
1.2 IBM BlueGene systems installed at LLNL 3
1.3 Performance optimization process 5
1.4 Manual visual comparison of two processes 6
1.5 Manual visual comparison of multiple processes 7

2.1 Sampling and instrumentation 10
2.2 Profile and trace data representations 11
2.3 Flat gprof profile 12
2.4 Phase-based profile 13
2.5 Jumpshot timeline 14
2.6 Comparison view of Vampir 15
2.7 Structural comparison of two resource hierarchies 16
2.8 Intra-process segment matching scheme 18
2.9 C3G compression scheme 19
2.10 Iteration detection using wavelet and autocorrelation techniques 21
2.11 Detection of the computation structure 22
2.12 Patterns of inefficient behavior 23
2.13 Critical path profile 24

3.1 Timing differences between event streams 29
3.2 Construction of function sequences 30
3.3 Direct comparison of event durations 31
3.4 Types of changes between two function sequences 32
3.5 Dynamic programming matrix calculation step 33
3.6 Dynamic programming matrix 34
3.7 Optimal alignment of two sequences 34
3.8 Hirschberg algorithm scheme 35
3.9 Hirschberg algorithm - horizontal split 35
3.10 Hirschberg algorithm - vertical split 36
3.11 Hirschberg algorithm - new sub-matrices 37
3.12 Dynamic programming matrix using Levenshtein costs 38
3.13 Ukkonen cutoff scheme 38
3.14 The U matrix used by Ukkonen's algorithm 40
3.15 Dynamic programming matrix showing the fields calculated by Ukkonen's algorithm . . 40
3.16 Fast-Alignment-Heuristic 41
3.17 Hierarchical alignment scheme 43
3.18 Hierarchical alignment steps 43
3.19 Analysis sub-sequence length influence on the HA algorithm performance 45
3.20 Non-optimal hierarchical alignment example 1 47
3.21 Non-optimal hierarchical alignment example 2 48
3.22 Difference between direct scores for flat and hierarchical alignments 50
3.23 HA score correction for changes on the top call level 51
3.24 HA score correction for changes on the middle call level 52
3.25 HA score correction for changes on the bottom call level 53

3.26 Alignment times for different input similarities 57
3.27 Alignment times for dissimilar input sequences 57

4.1 Dissimilarity timeline 64
4.2 Artificial timing introduced by the alignment 65
4.3 Runtime skew timeline 66
4.4 Comparison of two BT runs built with different compiler optimization levels 67
4.5 Unaligned rank 0 processes of the default and optimized AMG version 69
4.6 Aligned rank 0 processes of the default and optimized AMG version 69
4.7 Similarity and runtime skew analysis between the default and optimized AMG version . 70
4.8 Comparison of two iterations of ParaDiS versions 2.2.3 and 2.3.5.1 71
4.9 Similarity and runtime skew analysis between ParaDiS versions 2.2.3 and 2.3.5.1 72

5.1 Call trees of two example processes 74
5.2 Call trees of four example processes 78
5.3 Incidence relation table and concept lattice for four example processes 80
5.4 Concept lattice for an execution of the BT kernel 81
5.5 The progressive alignment scheme 82
5.6 Multiple sequence alignment example 84
5.7 MSA result comparison for two different options of adding processes 85
5.8 Input sequences for the guide tree construction example 87
5.9 MSA result achieved by adding sequences using a guide tree approach 87
5.10 MSA result achieved by adding sequences without a guide tree approach 88
5.11 Input processes for the hierarchical MSA scheme example 90
5.12 Merged call tree computed by the hierarchical MSA algorithm 90
5.13 Hierarchical MSA algorithm scheme 91
5.14 Merged call tree for AMG2006 processes 92
5.15 Merged call tree for ParaDiS processes 93
5.16 Scalability analysis of the hash-based clustering algorithm 96
5.17 Scalability analysis of the K-Medoids algorithm 96
5.18 Scalability analysis of the heuristic guide tree construction approach 98
5.19 Scalability analysis of the similarity matrix based guide tree construction approach . . . 99

6.1 Illustration of the state selection for pixels in a timeline chart 102
6.2 Illustration of the folding operation $select_{max}$ for three event streams 103
6.3 Vampir timeline showing a run of the PIConGPU CUDA application 105
6.4 Timeline grouping applied to four CUDA devices running PIConGPU 106
6.5 Visualization of iterations in PIConGPU using folded CUDA streams 107
6.6 Statistic showing the CUDA device utilization 107
6.7 Visualization of the folded timelines of a COSMO-SPECS weather forecast code run . . 108
6.8 Inclusive vs. exclusive time of a function invocation 109
6.9 Selection scheme for time dominant functions 110
6.10 Calculation of performance variations 111
6.11 Analysis of a COSMO-SPECS application run 113
6.12 Analysis of a COSMO-SPECS+FD4 application run 115
6.13 Analysis of a WRF application run 117

List of Tables

2.1 Limitations of existing techniques - Visual analysis 26
2.2 Limitations of existing techniques - Structural analysis 27
2.3 Limitations of existing techniques - Temporal analysis 28

3.1 Memory requirement for the Needleman–Wunsch algorithm 53
3.2 Alignment times for equal sequences 55
3.3 Alignment times for half-equal sequences 55
3.4 Alignment times for small equal blocks sequences 55
3.5 Alignment times for large equal blocks sequences 55
3.6 Alignment times for completely dissimilar sequences 55
3.7 Number of computed dynamic programming matrix cells for an example 56
3.8 Alignment times for AMG2006 processes 58
3.9 Number of sub-alignments during the hierarchical alignment of AMG2006 58
3.10 Alignment times for ParaDiS processes 59
3.11 Number of sub-alignments during the hierarchical alignment of ParaDiS 59
3.12 Alignment times for ParaDiS processes (increased detail version) 59
3.13 Number of sub-alignments during the hierarchical alignment of ParaDiS (increased detail) 60
3.14 Alignment times for the comparison of ParaDiS with AMG2006 60
3.15 Number of sub-alignments during the hierarchical alignment of ParaDiS with AMG2006 60

4.1 Similarity between ParaDiS processes 72
4.2 Function time difference table for the rank0/rank0 comparison of ParaDiS 72

5.1 Hash map with the clustering result for an example 80
5.2 Guide tree table with ordered results 87
5.3 Number of occurrences of characters in the sequences given in Figure 5.8 88
5.4 Sequences of Figure 5.8 ranked according to the number of their top characters 88
5.5 Memory requirement for the similarity matrix computed by K-Medoids and DBSCAN . 95
5.6 Clustering times for ParaDiS 95
5.7 Clustering time study with three real-world applications 97
5.8 Comparison of the generation times for the two guide tree construction approaches . . . 97
5.9 Multiple alignment time study with three real-world applications 99

Bibliography

[1] L. Adhianto, S. Banerjee, M. Fagan, M. Krentel, G. Marin, J. Mellor-Crummey, and N. R. Tallent. HPCToolkit: Tools for Performance Analysis of Optimized Parallel Programs. *Concurrency and Computation: Practice and Experience*, 22(6):685–701, 2010.

[2] Allinea MAP 6.0. `http://www.allinea.com/products/map/`, 2016.

[3] Allinea DDT and MAP User Guide — Viewing Stacks in Parallel. `https://www.allinea.com/user-guide/forge/userguide.html`, October 2015.

[4] AMD CodeXL Tool Suite 1.2. `http://developer.amd.com/tools-and-sdks/heterogeneous-computing/codexl/`, 2013.

[5] Gene M. Amdahl. Validity of the Single Processor Approach to Achieving Large-Scale Computing Capabilities. In *Proceedings of American Federation of Information Processing Societies (AFIPS), Spring Joint Computer Conference*, AFIPS '67, pages 483–485, 1967.

[6] Dorian C. Arnold, Dong H. Ahn, Bronis R. de Supinski, Gregory L. Lee, Barton P. Miller, and Martin Schulz. Stack Trace Analysis for Large Scale Debugging. In *Proceedings of the 2010 IEEE 21th International Parallel and Distributed Processing Symposium*, IPDPS '07, Los Alamitos, CA, USA, 2007. IEEE Computer Society.

[7] Athanasios Arsenlis, Wei Cai, Meijie Tang, Moono Rhee, Tomas Oppelstrup, Gregg Hommes, Tom G. Pierce, and Vasily V. Bulatov. Enabling Strain Hardening Simulations with Dislocation Dynamics. *Modelling and Simulation in Materials Science and Engineering*, 15(6):553, 2007.

[8] D. H. Bailey, E. Barszcz, J. T. Barton, D. S. Browning, R. L. Carter, L. Dagum, R. A. Fatoohi, P. O. Frederickson, T. A. Lasinski, R. S. Schreiber, H. D. Simon, V. Venkatakrishnan, and S. K. Weeratunga. The NAS Parallel Benchmarks - Summary and Preliminary Results. In *Proceedings of the 1991 ACM/IEEE Conference on Supercomputing*, Supercomputing '91, pages 158–165, New York, NY, USA, 1991. ACM.

[9] Robert Bell, AllenD. Malony, and Sameer Shende. ParaProf: A Portable, Extensible, and Scalable Tool for Parallel Performance Profile Analysis. In *Euro-Par 2003 Parallel Processing*, volume 2790 of *Lecture Notes in Computer Science*, pages 17–26. Springer Berlin Heidelberg, 2003.

[10] Richard Bellman. *Dynamic Programming*. Princeton University Press, Princeton, NJ, USA, 1957.

[11] Shajulin Benedict, Ventsislav Petkov, and Michael Gerndt. PERISCOPE: An Online-Based Distributed Performance Analysis Tool. In *Tools for High Performance Computing 2009*, pages 1–16. Springer Berlin Heidelberg, 2010.

[12] G. Bertoni, J. Daemen, M. Peeters, and G. Van Assche. The KECCAK reference. `http://keccak.noekeon.org/Keccak-reference-3.0.pdf`, January 2011.

[13] David Böhme, Bronis R. de Supinski, Markus Geimer, Martin Schulz, and Felix Wolf. Scalable Critical-Path Based Performance Analysis. In *Proceedings of the 26th IEEE International Parallel & Distributed Processing Symposium (IPDPS), Shanghai, China*, pages 1330–1340. IEEE Computer Society, May 2012.

[14] David Böhme, Markus Geimer, and Felix Wolf. Characterizing Load and Communication Imbalance in Large-Scale Parallel Applications. In *Proceedings of the 26th IEEE International Parallel & Distributed Processing Symposium Workshops & PhD Forum (IPDPSW), Shanghai, China*, pages 2538–2541. IEEE Computer Society, May 2012.

[15] David Böhme, Markus Geimer, Felix Wolf, and Lukas Arnold. Identifying the Root Causes of Wait States in Large-Scale Parallel Applications. In *Proceedings of the 2010 39th International Conference on Parallel Processing*, ICPP '10, pages 90–100, Washington, DC, USA, 2010. IEEE Computer Society.

[16] Ronny Brendel, Michael Heyde, Holger Brunst, Tobias Hilbrich, and Matthias Weber. Edge Bundling for Visualizing Communication Behavior. SUBMITTED TO: *3nd Workshop on Visual Performance Analysis (VPA), held in conjunction with SC16*, 2016.

[17] Holger Brunst. *Integrative Concepts for Scalable Distributed Performance Analysis and Visualization of Parallel Programs*. Ph.D. Dissertation, Technische Universität Dresden, 2008.

[18] Holger Brunst and Matthias Weber. Custom Hot Spot Analysis of HPC Software with the Vampir Performance Tool Suite. In *Proceedings of the 6th International Parallel Tools Workshop*, pages 95–114. Springer Berlin Heidelberg, September 2012.

[19] Holger Brunst, Manuela Winkler, Wolfgang E. Nagel, and Hans-Christian Hoppe. Performance Optimization for Large Scale Computing: The Scalable VAMPIR Approach. In *Computational Science - ICCS 2001*, volume 2074 of *Lecture Notes in Computer Science*, pages 751–760. Springer Berlin Heidelberg, 2001.

[20] H. Burau, R. Widera, W. Hönig, G. Juckeland, A. Debus, T. Kluge, U. Schramm, T.E. Cowan, R. Sauerbrey, and M. Bussmann. PIConGPU: A Fully Relativistic Particle-in-Cell Code for a GPU Cluster. *Plasma Science, IEEE Transactions on*, 38(10):2831–2839, October 2010.

[21] M. Bussmann, H. Burau, T. E. Cowan, A. Debus, A. Huebl, G. Juckeland, T. Kluge, W. E. Nagel, R. Pausch, F. Schmitt, U. Schramm, J. Schuchart, and R. Widera. Radiative Signatures of the Relativistic Kelvin-Helmholtz Instability. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '13, pages 5:1–5:12, New York, NY, USA, 2013. ACM.

[22] Marc Casas, Rosa M. Badia, and Jesús Labarta. Automatic Phase Detection of MPI Applications. In *Proceedings of the 14th Conference on Parallel Computing (ParCo 2007)*, September 2007.

[23] Marc Casas, Rosa M. Badia, and Jesús Labarta. Automatic Structure Extraction from MPI Applications Tracefiles. In *Proceedings of the 13th international Euro-Par conference on Parallel Processing*, Euro-Par'07, pages 3–12, 2007.

[24] Marc Casas, Rosa M. Badia, and Jesús Labarta. Automatic Analysis of Speedup of MPI Applications. In *Proceedings of the 22nd annual international Conference on Supercomputing*, ICS '08, pages 349–358, 2008.

[25] Marc Casas, Rosa M. Badia, and Jesús Labarta. Automatic Phase Detection and Structure Extraction of MPI Applications. *International Journal of High Performance Computing Applications*, 24(3):335–360, August 2010.

[26] Marc Casas, Harald Servat, Rosa M. Badia, and Jesús Labarta. Extracting the Optimal Sampling Frequency of Applications Using Spectral Analysis. *Concurrency and Computation: Practice and Experience*, 24(3):237–259, March 2011.

[27] Marc Casas Guix. *Spectral Analysis of Executions of Computer Programs and its Applications on Performance Analysis*. Ph.D. Dissertation, Universitat Politècnica de Catalunya, 2010.

[28] Anthony Chan, William Gropp, and Ewing Lusk. An Efficient Format for Nearly Constant-Time Access to Arbitrary Time Intervals in Large Trace Files. *Scientific Programming*, 16(2-3):155–165, 2008.

[29] DARPA. ExaScale Computing Study: Technology Challenges in Achieving Exascale Systems (TR-2008-13). `http://www.cse.nd.edu/Reports/2008/TR-2008-13.pdf`, 2008.

[30] Jack J. Dongarra, Piotr Luszczek, and Antoine Petitet. The LINPACK Benchmark: Past, Present, and Future. *Concurrency and Computation: Practice and Experience*, 15(9):803–820, 2003.

[31] Sean R Eddy. What is dynamic programming? *Nature Biotechnology*, 22(7):909–910, July 2004.

[32] Martin Ester, Hans-Peter Kriegel, Jörg Sander, and Xiaowei Xu. A Density-Based Algorithm for Discovering Clusters in Large Spatial Databases with Noise. In *KDD'96*, pages 226–231, 1996.

[33] J. Fenlason and R. Stallman. *GNU gprof: The GNU Profiler*. Free Software Foundation, 1988. `http://sourceware.org/binutils/docs/gprof/`.

[34] Hormozd Gahvari, Allison H. Baker, Martin Schulz, Ulrike Meier Yang, Kirk E. Jordan, and William Gropp. Modeling the Performance of an Algebraic Multigrid Cycle on HPC Platforms. In *Proceedings of the International Conference on Supercomputing*, ICS '11, pages 172–181, 2011.

[35] Todd Gamblin. *Scalable Performance Measurement and Analysis*. Ph.D. Dissertation, University of North Carolina at Chapel Hill, 2009.

[36] Todd Gamblin, Bronis R. de Supinski, Martin Schulz, Rob Fowler, and Daniel A. Reed. Clustering Performance Data Efficiently at Massive Scales. In *Proceedings of the 24th ACM International Conference on Supercomputing*, ICS '10, pages 243–252, New York, NY, USA, 2010. ACM.

[37] Fabian Gasper, Klaus Görgen, Prabhakar Shrestha, Mauro Sulis, Jehan Rihani, Markus Geimer, and Stefan Kollet. Implementation and Scaling of the Fully Coupled Terrestrial Systems Modeling Platform (TerrSysMP v1.0) in a Massively Parallel Supercomputing Environment – A Case Study on JUQUEEN (IBM Blue Gene/Q). *Geoscientific Model Development*, 7(5):2531–2543, October 2014.

[38] Markus Geimer, Felix Wolf, Brian J. N. Wylie, and Bernd Mohr. Scalable Parallel Trace-based Performance Analysis. In *Proceedings of the 13th European PVM/MPI User's Group Conference on Recent Advances in Parallel Virtual Machine and Message Passing Interface*, EuroPVM/MPI'06, pages 303–312, Berlin, Heidelberg, 2006. Springer-Verlag.

[39] Markus Geimer, Felix Wolf, Brian J. N. Wylie, and Bernd Mohr. A Scalable Tool Architecture for Diagnosing Wait States in Massively Parallel Applications. *Parallel Computing*, 35(7):375–388, July 2009.

[40] M. Gerndt and M. Ott. Automatic Performance Analysis with Periscope. *Concurrency and Computation: Practice and Experience*, 22(6):736–748, April 2010.

[41] P. Gibbon, F. N. Beg, E. L. Clark, R. G. Evans, and M. Zepf. Tree-code simulations of proton acceleration from laser-irradiated wire targets. *Physics of Plasmas*, 11(8):4032–4040, 2004.

[42] H. H. Goldstine and A. Goldstine. The Electronic Numerical Integrator and Computer (ENIAC). *Mathematical Tables and Other Aids to Computation*, 2(15), July 1946.

[43] J. González, J. Giménez, and J. Labarta. Automatic Detection of Parallel Applications Computation Phases. In *Parallel and Distributed Processing. 23rd IEEE International Symposium on*, IPDPS '09, pages 1–11, May 2009.

[44] J. González, J. Giménez, and J. Labarta. Automatic Evaluation of the Computation Structure of Parallel Applications. In *Parallel and Distributed Computing, Applications and Technologies. Proceedings of the 10th International Conference on*, PDCAT '09, pages 138–145, December 2009.

[45] Juan González, Judit Giménez, and Jesús Labarta. Performance Analytics: Understanding Parallel Applications Using Cluster and Sequence Analysis. In *Proceedings of the 7th International Parallel Tools Workshop*. Springer Berlin Heidelberg, September 2013.

[46] Juan González, Kevin Huck, Judit Giménez, and Jesús Labarta. Automatic Refinement of Parallel Applications Structure Detection. In *LSPP '12: Proceedings of the 2012 Workshop on Large-Scale Parallel Processing*, May 2012.

[47] Susan L. Graham, Peter B. Kessler, and Marshall K. Mckusick. gprof: A Call Graph Execution Profiler. *SIGPLAN Not.*, 17(6):120–126, June 1982.

[48] V. Grützun, O. Knoth, and M. Simmel. Simulation of the influence of aerosol particle characteristics on clouds and precipitation with LM-SPECS: Model description and first results. *Atmospheric Research*, 90(2–4):233–242, 2008.

[49] Dan Gusfield. *Algorithms on Strings, Trees, and Sequences: Computer Science and Computational Biology*. Cambridge University Press, 1997.

[50] John L. Gustafson. Reevaluating Amdahl's Law. *Commun. ACM*, 31(5):532–533, May 1988.

[51] J. A. Hartigan and M. A. Wong. Algorithm AS 136: A K-Means Clustering Algorithm. *Journal of the Royal Statistical Society. Series C (Applied Statistics)*, 28(1):100–108, 1979.

[52] Matthias Hauswirth. *Understanding Program Performance Using Temporal Vertical Profiles*. Ph.D. Dissertation, University of Colorado at Boulder, Boulder, CO, USA, 2005.

[53] Matthias Hauswirth, Amer Diwan, Peter F. Sweeney, and Michael C. Mozer. Automating Vertical Profiling. *SIGPLAN Not.*, 40(10):281–296, October 2005.

[54] S. Henikoff and J. G. Henikoff. Amino Acid Substitution Matrices from Protein Blocks. *Proceedings of the National Academy of Sciences of the United States of America (PNAS)*, 89(22):10915–10919, November 1992.

[55] Robert Henschel, Matthias Lieber, Le-Shin Wu, Phillip M. Nista, Brian J. Haas, and Richard D. LeDuc. Trinity RNA-Seq Assembler Performance Optimization. In *Proceedings of the 1st Conference of the Extreme Science and Engineering Discovery Environment: Bridging from the eXtreme to the Campus and Beyond*, XSEDE '12, pages 45:1–45:8, New York, NY, USA, 2012. ACM.

[56] Marc-André Hermanns, Manfred Miklosch, David Böhme, and Felix Wolf. Understanding the Formation of Wait States in Applications with One-sided Communication. In *Proceedings of the 20th European MPI Users' Group Meeting*, EuroMPI '13, pages 73–78, New York, NY, USA, 2013. ACM.

[57] Desmond G. Higgins and Paul M. Sharp. CLUSTAL: a package for performing multiple sequence alignment on a microcomputer. *Gene*, 73:237–244, 1988.

[58] Marcus Hilbrich, Matthias Weber, and Ronny Tschüter. Automatic Analysis of Large Data Sets: A Walk-Through on Methods from Different Perspectives. In *Cloud Computing and Big Data (CloudCom-Asia), 2013 International Conference on*, pages 373–380, December 2013.

[59] D. S. Hirschberg. A Linear Space Algorithm for Computing Maximal Common Subsequences. *Communications of the ACM*, 18(6):341–343, June 1975.

[60] Jeffrey K. Hollingsworth and Barton P. Miller. Dynamic Control of Performance Monitoring on Large Scale Parallel Systems. In *Proceedings of the 7th International Conference on Supercomputing*, ICS '93, pages 185–194, New York, NY, USA, 1993. ACM.

[61] K.A. Huck and A.D. Malony. PerfExplorer: A Performance Data Mining Framework For Large-Scale Parallel Computing. In *Supercomputing, 2005. Proceedings of the ACM/IEEE SC 2005 Conference*, pages 41–41, 2005.

[62] Kevin A. Huck, Allen D. Malony, Sameer Shende, and Alan Morris. Scalable, Automated Performance Analysis with TAU and PerfExplorer. In *Proceedings of the 14th Conference on Parallel Computing (ParCo 2007)*, pages 629–636, 2007.

[63] Intel Trace Analyzer and Collector. `http://software.intel.com/en-us/articles/intel-trace-analyzer/`, 2013.

[64] Intel VTune Amplifier 2013. `http://software.intel.com/en-us/vtuneampxe_2013_ug_lin`, 2013. Document number: 326734-011.

[65] Katherine E. Isaacs, Alfredo Giménez, Ilir Jusufi, Todd Gamblin, Abhinav Bhatele, Martin Schulz, Bernd Hamann, and Peer-Timo Bremer. State of the Art of Performance Visualization. In *EuroVis - STARs*. The Eurographics Association, 2014.

[66] Paul Jaccard. Étude comparative de la distribution florale dans une portion des Alpes et du Jura. *Bulletin de la Société Vaudoise des Sciences Naturelles*, 37:547–579, 1901.

[67] Karen L. Karavanic. *Experiment Management Support for Parallel Performance Tuning*. Ph.D. Dissertation, University of Wisconsin – Madison, 1999.

[68] Karen L. Karavanic, John May, Kathryn Mohror, Brian Miller, Kevin Huck, Rashawn Knapp, and Brian Pugh. Integrating Database Technology with Comparison-based Parallel Performance Diagnosis: The PerfTrack Performance Experiment Management Tool. In *Proceedings of the 2005 ACM/IEEE Conference on Supercomputing*, SC '05, Washington, DC, USA, 2005. IEEE Computer Society.

[69] Karen L. Karavanic and Barton P. Miller. Experiment Management Support for Performance Tuning. In *Proceedings of the 1997 ACM/IEEE Conference on Supercomputing (CDROM)*, Supercomputing '97, pages 1–10, New York, NY, USA, 1997. ACM.

[70] Karen L. Karavanic and Barton P. Miller. A Framework for Multi-Execution Performance Tuning. In *On-line monitoring systems and computer tool interoperability*, pages 61–89. Nova Science Publishers, Inc., Commack, NY, USA, 2003.

[71] L. Kaufman and P. Rousseeuw. *Clustering by Means of Medoids*. Reports of the Faculty of Mathematics and Informatics. Delft University of Technology. Fac., Univ., 1987.

[72] L. Kaufman and P. J. Rousseeuw. *Finding Groups in Data: An Introduction to Cluster Analysis*. Wiley Series in Probability and Statistics. Wiley, 2005.

[73] Andreas Knüpfer. Construction and Compression of Complete Call Graphs for Post-Mortem Program Trace Analysis. In *Proceedings of the 2005 International Conference on Parallel Processing*, ICPP '05, pages 165–172, Washington, DC, USA, 2005. IEEE Computer Society.

[74] Andreas Knüpfer. *Advanced Memory Data Structures for Scalable Event Trace Analysis*. Ph.D. Dissertation, Technische Universität Dresden, 2009.

[75] Andreas Knüpfer, Ronny Brendel, Holger Brunst, Hartmut Mix, and Wolfgang E. Nagel. Introducing the Open Trace Format (OTF). In *Proceedings of the 6th international conference on Computational Science - Volume Part II*, ICCS'06, pages 526–533, Berlin, Heidelberg, 2006. Springer-Verlag.

[76] Andreas Knüpfer and Wolfgang E. Nagel. Compressible Memory Data Structures for Event-based Trace Analysis. *Future Generation Computer Systems*, 22(3):359–368, February 2006.

[77] Andreas Knüpfer, Bernhard Voigt, Wolfgang E. Nagel, and Hartmut 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*, PARA'06, pages 430–439, Berlin, Heidelberg, 2007. Springer-Verlag.

[78] J. Labarta, J. Giménez, E. Martínez, P. Gonzáles, H. Servat, G. Llort, and X. Aguilar. Scalability of Visualization and Tracing Tools. In *Parallel Computing: Current & Future Issues of High-End Computing, Proceedings of the International Conference ParCo 2005*, volume 33 of *NIC series*, pages 869–876, Jülich, 2006. John von Neumann Institute for Computing.

[79] Jesús Labarta. New Analysis Techniques in the CEPBA-Tools Environment. In *Tools for High Performance Computing 2009*, pages 125–143. Springer Berlin Heidelberg, 2010.

[80] Peter H. Lauritzen. Supercomputing and Climate Modeling. Summer School: An Introduction to Climate Modeling (May 28), University of Stockholm, Sweden, 2012. `http://www.cgd.ucar.edu/cms/pel/publications.html`.

[81] Lawrence Livermore National Laboratory (LLNL). `https://www.llnl.gov`, 2013.

[82] Vladimir I. Levenshtein. Binary Codes Capable of Correcting Deletions, Insertions, and Reversals. *Soviet Physics Doklady*, 10(8):707–710, 1966.

[83] Matthias Lieber, Verena Grützun, Ralf Wolke, Matthias S. Müller, and Wolfgang E. Nagel. Highly Scalable Dynamic Load Balancing in the Atmospheric Modeling System COSMO-SPECS+FD4. In *Applied Parallel and Scientific Computing - 10th International Conference, PARA 2010*, volume 7133 of *LNCS*, pages 131–141, 2012.

[84] Germán Llort, Marc Casas, Harald Servat, Kevin Huck, Judit Giménez, and Jesús Labarta. Trace Spectral Analysis toward Dynamic Levels of Detail. In *Proceedings of the 2011 IEEE 17th International Conference on Parallel and Distributed Systems*, ICPADS '11, pages 332–339, 2011.

[85] Germán Llort, Harald Servat, Juan González, Judit Giménez, and Jesús Labarta. On the Usefulness of Object Tracking Techniques in Performance Analysis. In *Proceedings of SC13: International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '13, pages 29:1–29:11, New York, NY, USA, 2013. ACM.

[86] Stuart P. Lloyd. Least Squares Quantization in PCM. *IEEE Transactions on Information Theory*, 28(2):129–137, 1982.

[87] Daniel Lorenz, David Böhme, Bernd Mohr, Alexandre Strube, and Zoltán Szebenyi. Extending Scalasca's Analysis Features. In *Tools for High Performance Computing 2012*, pages 115–126. Springer Berlin Heidelberg, 2013.

[88] J. B. MacQueen. Some Methods for Classification and Analysis of MultiVariate Observations. In *Proceedings of the Fifth Berkeley Symposium on Mathematical Statistics and Probability*, volume 1, pages 281–297. University of California Press, 1967.

[89] Allen D. Malony, Sameer Shende, and Alan Morris. Phase-Based Parallel Performance Profiling. In *Parallel Computing: Current & Future Issues of High-End Computing, Proceedings of the International Conference ParCo 2005*, volume 33, pages 203–210. Central Institute for Applied Mathematics, Jülich, Germany, 2005.

[90] John Mellor-Crummey, Robert J. Fowler, Gabriel Marin, and Nathan Tallent. HPCView: A Tool for Top-down Analysis of Node Performance. *The Journal of Supercomputing*, 23(1):81–104, August 2002.

[91] Barton P. Miller. What to Draw? When to Draw?: An Essay on Parallel Program Visualization. *J. Parallel Distrib. Comput.*, 18(2):265–269, June 1993.

[92] Barton P. Miller, Mark D. Callaghan, Jonathan M. Cargille, Jeffrey K. Hollingsworth, R. Bruce Irvin, Karen L. Karavanic, Krishna Kunchithapadam, and Tia Newhall. The Paradyn Parallel Performance Measurement Tool. *IEEE Computer*, 28(11):37–46, November 1995.

[93] Webb Miller and Eugene W. Myers. A File Comparison Program. *Software: Practice and Experience*, 15(11):1025–1040, 1985.

[94] Kathryn Mohror and Karen L. Karavanic. An Investigation of Tracing Overheads on High End Systems. Technical Report TR-06-06, Portland State University, Department of Computer Science, December 2006.

[95] Kathryn Mohror and Karen L. Karavanic. A Study of Tracing Overhead on a High-Performance Linux Cluster. In *Proceedings of the 12th ACM SIGPLAN symposium on Principles and practice of parallel programming*, PPoPP '07, pages 158–159, 2007.

[96] Kathryn Mohror and Karen L. Karavanic. Scalable Event-based Performance Measurement in High-End Environments. *SIGMETRICS Perform. Eval. Rev.*, 35(3):64–65, December 2007.

[97] Kathryn Mohror and Karen L. Karavanic. Towards Scalable Event Tracing for High End Systems. In *Proceedings of the Third international conference on High Performance Computing and Communications*, HPCC'07, pages 695–706, Berlin, Heidelberg, 2007. Springer-Verlag.

[98] Kathryn Mohror and Karen 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*, SC '09, pages 55:1–55:12, 2009.

[99] Kathryn Mohror and Karen L. Karavanic. Trace Profiling: Scalable Event Tracing on High-End Parallel Systems. *Parallel Computing*, 38(4–5):194–225, 2012.

[100] Kathryn Mohror, Karen L. Karavanic, and Allan Snavely. Scalable Event Trace Visualization. In *Proceedings of the 2009 international conference on Parallel processing*, Euro-Par'09, pages 228–237, Berlin, Heidelberg, 2010. Springer-Verlag.

[101] Kathryn M. Mohror. *Scalable Event Tracing on High-End Parallel Systems*. Ph.D. Dissertation, Portland State University, Department of Computer Science, 2009.

[102] Gordon E. Moore. Cramming More Components onto Integrated Circuits. *Electronics Magazine*, 38(8):114–117, April 1965.

[103] MPI: A Message-Passing Interface Standard, Version 3.0. http://www.mpi-forum.org/docs/mpi-3.0/mpi30-report.pdf, 2012.

[104] Frank Mueller, Xing Wu, Martin Schulz, Bronis R. de Supinski, and Todd Gamblin. ScalaTrace: Tracing, Analysis and Modeling of HPC Codes at Scale. In *Proceedings of the 10th international conference on Applied Parallel and Scientific Computing - Volume 2*, PARA'10, pages 410–418, Berlin, Heidelberg, 2012. Springer-Verlag.

[105] Eugene W. Myers. An O(ND) Difference Algorithm and Its Variations. *Algorithmica*, 1(2):251–266, 1986.

[106] W. E. Nagel, A. Arnold, M. Weber, H.-Ch. Hoppe, and K. Solchenbach. VAMPIR: Visualization and Analysis of MPI Resources. *Supercomputer*, 12:69–80, 1996.

[107] Saul B. Needleman and Christian D. Wunsch. A General Method Applicable to the Search for Similarities in the Amino Acid Sequence of Two Proteins. *Journal of Molecular Biology*, 48(3):443–453, 1970.

[108] Oleg Y. Nickolayev, Philip C. Roth, Daniel, and Daniel A. Reed. Real-Time Statistical Clustering For Event Trace Reduction. In *International Journal of Supercomputer Applications and High Performance Computing*, pages 144–159, 1997.

[109] Michael Noeth, Frank Mueller, Martin Schulz, and Bronis R. de Supinski. Scalable Compression and Replay of Communication Traces in Massively Parallel Environments. In *Parallel and Distributed Processing. IEEE International Symposium on*, IPDPS '07, pages 1–11, March 2007.

[110] Michael Noeth, Prasun Ratn, Frank Mueller, Martin Schulz, and Bronis R. de Supinski. ScalaTrace: Scalable Compression and Replay of Communication Traces for High-performance Computing. *Journal of Parallel and Distributed Computing*, 69(8):696–710, August 2009.

[111] OpenMP. `http://openmp.org/wp/openmp-specifications`, November 2015.

[112] Vincent Pillet, Jesús Labarta, Toni Cortes, and Sergi Girona. PARAVER: A Tool to Visualize and Analyze Parallel Code. In *Proceedings of WoTUG-18: Transputer and occam Developments*, pages 17–31, March 1995.

[113] Robert Preissl. *Exploitation of Dynamic Communication Patterns through Static Analysis*. Ph.D. Dissertation, Johannes Kepler University of Linz, Austria, Institute of Graphics and Parallel Processing (GUP), 2009.

[114] Robert Preissl, Bronis R. de Supinski, Martin Schulz, Daniel J. Quinlan, Dieter Kranzlmüller, and Thomas Panas. Exploitation of Dynamic Communication Patterns through Static Analysis. In *Proceedings of the 39th International Conference on Parallel Processing*, ICPP '10, pages 51–60. IEEE Computer Society, September 2010.

[115] Robert Preissl, Thomas Köckerbauer, Martin Schulz, Dieter Kranzlmüller, Bronis R. de Supinski, and Daniel J. Quinlan. Detecting Patterns in MPI Communication Traces. In *Proceedings of the 37th International Conference on Parallel Processing*, ICPP '08, pages 230–237. IEEE Computer Society, 2008.

[116] Robert Preissl, Martin Schulz, Dieter Kranzlmüller, Bronis R. de Supinski, and Daniel J. Quinlan. Transforming MPI Source Code based on Communication Patterns. *Future Generation Computer Systems*, 26(1):147–154, January 2010.

[117] Robert Preissl, Martin Schulz, Dieter Kranzlmüller, Bronis R. Supinski, and Daniel J. Quinlan. Using MPI Communication Patterns to Guide Source Code Transformations. In *Proceedings of the 8th international Conference on Computational Science, Part III*, ICCS '08, pages 253–260, Berlin, Heidelberg, 2008. Springer-Verlag.

[118] Prasun Ratn, Frank Mueller, Bronis R. de Supinski, and Martin Schulz. Preserving time in large-scale communication traces. In *Proceedings of the 22nd Annual International Conference on Supercomputing*, ICS '08, pages 46–55, New York, NY, USA, 2008. ACM.

[119] Raúl Rojas. Konrad Zuse's Legacy: The Architecture of the Z1 and Z3. *Annals of the History of Computing, IEEE*, 19(2):5–16, April-June 1997.

[120] Diego Rossinelli, Babak Hejazialhosseini, Panagiotis Hadjidoukas, Costas Bekas, Alessandro Curioni, Adam Bertsch, Scott Futral, Steffen J. Schmidt, Nikolaus A. Adams, and Petros Koumoutsakos. 11 PFLOP/s Simulations of Cloud Cavitation Collapse. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '13, pages 3:1–3:13, New York, NY, USA, 2013. ACM.

[121] Philip C. Roth. ETRUSCA: Event Trace Reduction Using Statistical Data Clustering Analysis. Master's thesis, Department of Computer Science, University of Illinois at Urbana-Champaign, December 1995.

[122] Philip C. Roth, Dorian C. Arnold, and Barton P. Miller. MRNet: A Software-Based Multicast/Reduction Network for Scalable Tools. In *Proceedings of the 2003 ACM/IEEE Conference on Supercomputing*, SC '03, New York, NY, USA, 2003. ACM.

[123] Peter J. Rousseeuw. Silhouettes: A Graphical Aid to the Interpretation and Validation of Cluster Analysis. *Journal of Computational and Applied Mathematics*, 20(0):53–65, 1987.

[124] Martin Schulz and Bronis R. de Supinski. Practical Differential Profiling. In *Proceedings of the 13th international Euro-Par conference on Parallel Processing*, Euro-Par'07, pages 97–106, Berlin, Heidelberg, 2007. Springer-Verlag.

[125] Martin Schulz, Jim Galarowicz, Don Maghrak, William Hachfeld, David Montoya, and Scott Cranford. Open|SpeedShop: An Open Source Infrastructure for Parallel Performance Analysis. *Scientific Programming*, 16(2-3):105–121, April 2008.

[126] Sadi Evren Seker, Oguz Altun, Ugur Ayan, and Cihan Mert. A Novel String Distance Function Based on Most Frequent K Characters. *International Journal of Machine Learning and Computing (IJMLC)*, 4:177–183, 2014.

[127] Gilad Shainer, Tong Liu, John Michalakes, Jacob Liberman, Jeff Layton, Onur Celebioglu, Scot A Schultz, Joshua Mora, and David Cownie. Weather Research and Forecast (WRF) Model Performance and Profiling Analysis on Advanced Multi-core HPC Clusters. In *10th LCI International Conference on High-Performance Clustered Computing*, 2009.

[128] T. F. Smith and M. S. Waterman. Identification of Common Molecular Subsequences. *Journal of Molecular Biology*, 147(1):195–197, March 1981.

[129] Fengguang Song, Felix Wolf, Nikhil Bhatia, Jack Dongarra, and Shirley Moore. An Algebra for Cross-Experiment Performance Analysis. In *Proceedings of the 2004 International Conference on Parallel Processing*, ICPP '04, pages 63–72, Washington, DC, USA, 2004. IEEE Computer Society.

[130] Dan Terpstra, Heike Jagode, Haihang You, and Jack Dongarra. Collecting Performance Data with PAPI-C. In *Tools for High Performance Computing 2009*, pages 157–173. Springer Berlin Heidelberg, 2010.

[131] The TOP500 List of the World's Fastest Supercomputers. `http://www.top500.org`, 2016.

[132] Julie D. Thompson, Desmond G. Higgins, and Toby J. Gibson. CLUSTAL W: improving the sensitivity of progressive multiple sequence alignment through sequence weighting, position-specific gap penalties and weight matrix choice. *Nucleic Acids Research*, 22(22):4673–4680, 1994.

[133] Esko Ukkonen. Algorithms for approximate string matching. *Information Control*, 64(1-3):100–118, March 1985.

[134] Vampir 8 Manual. `http://www.vampir.eu/tutorial/manual`, 2016.

[135] Jeffrey Vetter and Chris Chambreau. mpiP: Lightweight, Scalable MPI Profiling. `http://mpip.sourceforge.net`, 2011.

[136] Michael Wagner, Ben Fulton, and Robert Henschel. Performance Optimization for the Trinity RNA-Seq Assembler. In *Proceedings of the 9th International Parallel Tools Workshop*. Springer Berlin Heidelberg, 2015.

[137] Michael Wagner, Andreas Knüpfer, and Wolfgang E. Nagel. Hierarchical Memory Buffering Techniques for an In-Memory Event Tracing Extension to the Open Trace Format 2. In *Parallel Processing (ICPP), 2013 42nd International Conference on*, pages 970–976, 2013.

[138] Lusheng Wang and Tao Jiang. On the Complexity of Multiple Sequence Alignment. *Journal of Computational Biology*, 1(4):337–348, 1994.

[139] Warren M. Washington, Lawrence Buja, and Anthony Craig. The Computational Future for Climate and Earth System Models: On the Path to Petaflop and Beyond. *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences*, 367(1890):833–846, March 2009.

[140] Matthias Weber, Ronny Brendel, and Holger Brunst. 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*, ISPA '12, pages 247–254, Washington, DC, USA, July 2012. IEEE Computer Society.

[141] Matthias Weber, Ronny Brendel, Tobias Hilbrich, Kathryn Mohror, Martin Schulz, and Holger Brunst. Structural Clustering: A New Approach to Support Performance Analysis at Scale. In *Proceedings of the 30th IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 484–493. IEEE Computer Society, May 2016.

[142] Matthias Weber, Ronald Geisler, Holger Brunst, and Wolfgang E. Nagel. Folding Methods for Event Timelines in Performance Analysis. In *Proceedings of the 29th IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 205–214. IEEE Computer Society, May 2015.

[143] Matthias Weber, Ronald Geisler, Tobias Hilbrich, Matthias Lieber, Ronny Brendel, Ronny Tschüter, Holger Brunst, and Wolfgang E. Nagel. Detection and Visualization of Performance Variations to Guide Identification of Application Bottlenecks. In *Proceedings of the 45th International Conference on Parallel Processing Workshops (ICPPW)*. IEEE Computer Society, 2016.

[144] Matthias Weber, Kathryn Mohror, Martin Schulz, Bronis R. de Supinski, Holger Brunst, and Wolfgang E. Nagel. Alignment-Based Metrics for Trace Comparison. In *Proceedings of the 19th International Conference on Parallel Processing*, Euro-Par'13, pages 29–40. Springer-Verlag, Berlin, Heidelberg, 2013.

[145] Matthias Weber, Kathryn Mohror, Martin Schulz, Bronis R. de Supinski, Holger Brunst, and Wolfgang E. Nagel. Structural Comparison of Parallel Applications. Research poster, Supercomputing 2013, Denver, CO, November 2013.

[146] F. Wolf and B. Mohr. Automatic Performance Analysis of Hybrid MPI/OpenMP Applications. In *Parallel, Distributed and Network-Based Processing, 2003. Proceedings. Eleventh Euromicro Conference on*, pages 13–22, 2003.

[147] Felix Wolf. *Automatic Performance Analysis on Parallel Computers with SMP Nodes*. Ph.D. Dissertation, RWTH Aachen, Forschungszentrum Jülich, 2003.

[148] Felix Wolf, Brian J. N. Wylie, Erika Ábrahám, Daniel Becker, Wolfgang Frings, Karl Fürlinger, Markus Geimer, Marc-André Hermanns, Bernd Mohr, Shirley Moore, Matthias Pfeifer, and Zoltán Szebenyi. Usage of the SCALASCA Toolset for Scalable Performance Analysis of Large-Scale Parallel Applications. In *Proceedings of the 2nd Parallel Tools Workshop, Stuttgart, Germany*, pages 157–167. Springer, July 2008.

[149] C. Eric Wu, Anthony Bolmarcich, Marc Snir, David Wootton, Farid Parpia, Anthony Chan, Ewing Lusk, and William Gropp. From Trace Generation to Visualization: A Performance Framework for Distributed Parallel Systems. In *Proceedings of the 2000 ACM/IEEE Conference on Supercomputing (CDROM)*, Supercomputing '00. IEEE Computer Society, 2000.

[150] Brian J. N. Wylie and Markus Geimer. Large-scale Performance Analysis of PFLOTRAN with Scalasca. In *Proceedings of the 53rd Cray User Group meeting, Fairbanks, AK, USA*. Cray User Group Inc., May 2011.

[151] Brian J. N. Wylie, Markus Geimer, and Felix Wolf. Performance Measurement and Analysis of Large-scale Parallel Applications on Leadership Computing Systems. *Scientific Programming*, 16(2-3):167–181, April 2008.

[152] Omer Zaki, Ewing Lusk, William Gropp, and Deborah Swider. Toward Scalable Performance Visualization with Jumpshot. *International Journal of High Performance Computing Applications*, 13(3):277–288, August 1999.