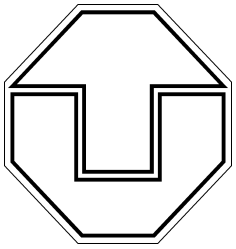

Technische Universität Dresden

Department of Computer Science

Institute of Computer Engineering

Chair of Computer Architecture

Diploma Thesis

submitted in partial fulfilment of the requirements
for the degree of Diplom-Informatiker

Automatic Profile-Based Characterisation of Performance Traces for Highly Parallel Applications

Author: Ronny Brendel (Born on the 11th October, 1985 in Meissen)
Professor: Prof. Dr. Wolfgang E. Nagel
Tutors: Matthias Weber, Dr. Holger Brunst

Dresden, 28th January, 2015

Task

The goal of this thesis is to develop methods that automatically compare and categorise performance measurements of highly parallel applications. Performance data within a single program run as well as between multiple runs is to be compared. The analysis should be based on profile data and invocation relationships between functions. The work will focus on the following key tasks:

- Develop a similarity metric based on profile data.
- Develop a method to efficiently group traces using the developed similarity metric and further properties.
- Rank structural and temporal differences between traces according to their significance.
- Determine significant anchor-points to assist further trace comparison, based on the analysis methods described above.
- Design a novel, informative, and scalable visualisation of profile data, based on the analysis methods described above.

Statement of Academic Integrity

I hereby declare that I prepared this thesis independently and without use of tools other than specified. Foreign thoughts, taken literally or in spirit, are marked as such. I also declare that I have not filed the present work at any other location or will submit it.

Ronny Brendel

Abstract

To exploit today's computers, developers need to design for concurrency. At the same time, synchronisation of parallel computation can be costly and cause performance loss. Therefore, software performance analysis and visualisation is more important than ever. Common analysis workflows compare multiple performance measurements to quantify the impact of optimisation, compare different hardware platforms, or search for load imbalances.

This thesis introduces novel comparative techniques to advance performance analysis and visualisation. It devises measures for detecting structural similarity between processes. Along with it, an efficient scheme to store and retrieve the required similarity data is introduced. The effectiveness and efficiency of the techniques are demonstrated using 15 different applications with up to 65,000 processes. Due to their interesting characteristics, two of these applications are investigated in greater detail. The approach is then used to improve a current visualisation. Furthermore, a promising, novel profile viewer is proposed, which among other innovations leverages the introduced techniques.

Acknowledgments

First of all, I want to thank Prof. Wolfgang Nagel for his support and encouragement during the last ten years. The Center for Information Services and High Performance Computing is a great place to work and learn at. Every employee and student takes part in making it the pleasent work environment it is, and for this I am indebted to them.

I want to express my gratitude to Holger Brunst, Matthias Weber and Tobias Hilbrich for proofreading my thesis and for giving valuable insight into technical and linguistic aspects of it. Claudia Schmidt, Ralph Müller-Pfefferkorn, Andreas Knüpfer, Hartmut Mix and Holger Brunst guided and supported me throughout my formal training, for which I am sincerely grateful.

I deeply appreciate the efforts of my parents Constanze and Maik Brendel to raise me well and support me in every facet of life. Of my friends, I especially want to thank Emil, Rebecca, Hermann and Frank for being awesome.

Contents

1 Introduction

Computers pervade everyday life. They are part of telephones, watches, TVs, cars and even washing machines. Every computer program needs to satisfy performance goals. The reasons are diverse. Examples include delivering a smooth user experience for the newest mobile phone, real-time constraints in brake systems and timely generation of the weather forecast for tomorrow. To achieve these goals, performance optimisation has become an integral part of the software development process.

Scientific computing relies on parallelism to leverage the power of large computer systems. Multiple threads of execution need to synchronise in order to work together effectively. Synchronisation can be costly and cause performance loss. Performance analysis and optimisation is, therefore, essential to get the most out of these expensive machines.

Over the last ten years, parallelisation went mainstream. Even a mobile phone's central processing unit contains up to eight cores, today. Programmers need to design for concurrency. Thus, optimisation to leverage parallelism effectively is more important than ever.

Common performance analysis and tuning workflows first analyse a program to find out why it does not meet its performance goal. After finding a performance problem and implementing an optimisation, one measures its consequences. These two performance measurements are then compared to quantify the impact of the chosen optimisation. One can also compare the performance of an application on different hardware platforms in order to decide which one to buy. Another application of *comparative analysis* is to search for load imbalance problems by examining two processes' timings in contrast.

Large-scale computers have hundreds of thousands of computing units. Manually comparing performance data of the resulting quantity is impractical. The need for automatic comparison arises. One straightforward instance of automatic comparison is the detection of similarly behaving processes to observe only one representative in detail. This approach also reduces the time and storage overhead of the analysis. Current software performance analysis tools often store and present raw data and rarely employ automatic comparison techniques.

This thesis introduces a structural similarity measure along with an efficient way to store and retrieve the information necessary for computing this measure. A modified version of these techniques is then used to discern a special case of dissimilarity. To demonstrate the effectiveness of the introduced techniques, first the modified variant is employed to verify varying levels of analysis granularity and to discern spawned threads from other differences. Second, the techniques are applied to improve an existing performance visualisation, as well as to develop a novel visualisation for highly parallel applications.

The structure of this thesis is as follows: Chapter 2 gives an introduction to software performance analysis and comparative techniques in particular, and highlights current developments in the field. Chapter 3 introduces two measures and proposes an efficient storage mechanism for the required data. Having presented the core techniques, their usefulness and performance is demonstrated in Chapter 4. To illustrate how the approach can be incorporated into performance visualisation software, two examples are presented. First, an existing visualisation is improved. Second, a novel profile viewer, leveraging the developed comparison techniques, is proposed.

2 Background & Related Work

This chapter gives an introduction to performance analysis and iterates current developments in comparative techniques.

2.1 Performance Analysis

A wealth of different information can be recorded during a program's runtime. First, function calls and their timing are of interest. There exist two different kinds of timing – *exclusive* and *inclusive*. The former excludes time spent in subroutines, whereas the latter includes it (Figure 2.1). Furthermore, recording the volume of calls and the relation between function invocations is important. In addition to this central information, other metrics like memory usage, hardware performance counters, temperature, network- and disk bandwidth can, for example, be recorded.

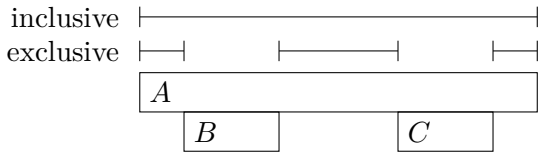

Figure 2.1: Exclusive and inclusive time of function A. It has two subroutines: B and C.

Software performance analysis can roughly be put into two categories—*profiling* and *tracing*. In tracing, information is exhaustively gathered, whereas profiling measures in intervals.

Profilers present a condensed view of a program's execution. They accumulate values on a per-function- or per-line-of-source-code-basis. The latter technique is called call path profiling [32]. Profiles are typically presented as tables (e.g. Table 2.1) or bar charts.

To obtain the required information for generating a profile, the underlying framework uses sampling. *Sampling* probes the running program in intervals and stores information like the active function call stack, the currently executed CPU[1] instruction and auxiliary metrics. The intervals are usually one or ten milliseconds long. Due to this, one cannot generally guarantee correctness of derived statistics. An error estimate as a function of event rate and sampling rate can be given. Profiles are typically small in file size and the generated runtime overhead is low. By modifying the sampling rate one can adjust the incurred overhead.

[1] short for central processing unit

Table 2.1: Output of a profiler for an example C++ program run

Time (%)	Seconds	Function name
5.44	1.21	`QListData::isEmpty`
2.96	0.66	`QHash::findNode`
2.67	0.60	`QList::last`
2.52	0.56	`QList::isEmpty`
2.04	0.46	`QList::end`
1.71	0.38	`handleEnter`
1.44	0.32	`FunctionCall::FunctionCall`
0.58	0.13	`QHash::find`
0.56	0.13	`QHashNode::same_key`
0.34	0.08	`std::forward`

There exists a multitude of profilers differing in focus and functionality. Most prominent is GNU gprof [29], which supports profiling single-threaded applications. The Google Performance Tools [27] include a profiler that is similar to gprof, but supports multi-threading. Another similar tool is Callgrind [11] along with the visualisation program KCachegrind [84]. These three profilers have in common that they do not directly support multi-process programming. An application is assumed to consist of exactly one process.

Much of scientific computing takes places on clusters of computers. Leveraging their potential requires multi-process programs, which in turn need performance tuning. Three tools that support this kind of application are Allinea MAP [3], HPCToolkit [1] and Open|Speedshop [70].

A more detailed way of measuring software performance is tracing. Unlike profiling, every function call is recorded and, depending on the actual implementation, for each of them all measurements are taken. As a result of the potentially large amount of recorded information, trace visualisers typically employ interactive presentation techniques. Figure 2.2 shows one such visualisation, in which users can zoom in and out, choose different views and filter events to investigate their application's runtime behaviour. To be able to record every entry into and exit from a function, tracing environments *instrument* an application. The added monitoring code writes the measurement results along with a timestamp into a log file. This file is called the *trace file*. The amount of application-unrelated work incurred by tracing depends mainly on the rate of recorded events. Function enter and leave events occur at a rapid pace. Therefore, tracing overhead is typically very large compared to profiling. There exist various ways to reduce this overhead including tracing only a subset of functions e.g. only certain API[2] calls, recording only a limited number of calls to a function, and tracing only during an interval instead of the whole program's runtime.

As a representative of tracing software, the Vampir Tool-Set [47], developed at the Technische Universität Dresden, is widely deployed on high performance computers. It consists

[2]short for application programming interface

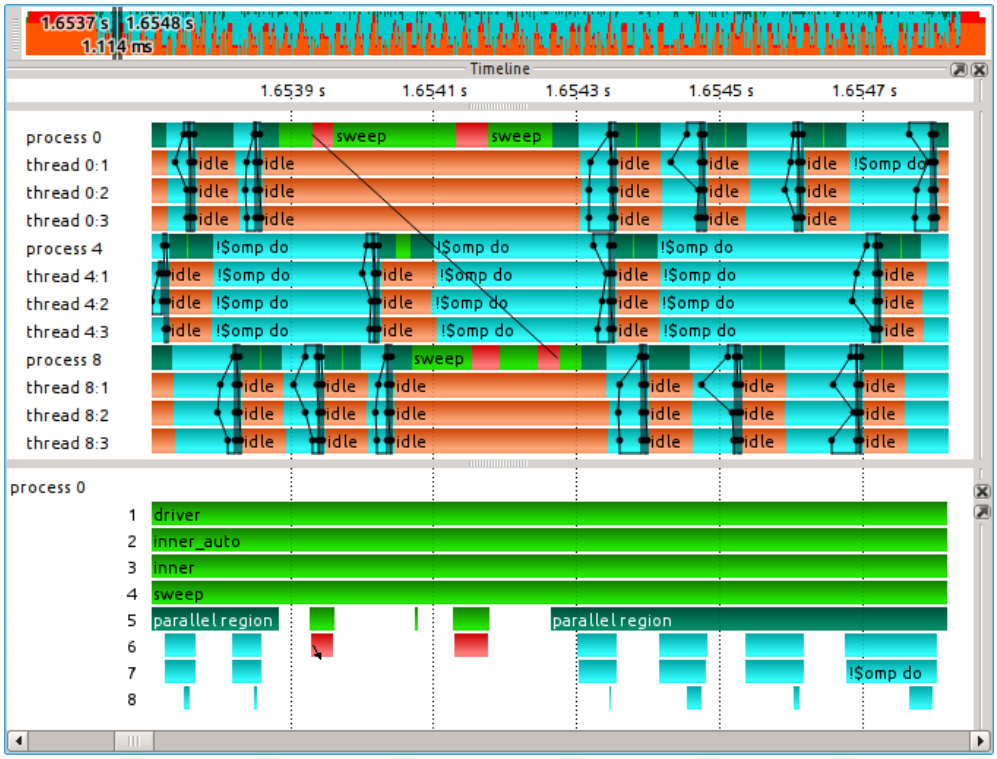

Figure 2.2: Example trace visualisation showing (from top to bottom) a summary view of active functions over the course of the program run, a timeline showing which function is active on different processes/threads, and the call stack over time of process 0. Black lines between processes indicate communication.

of the open-source trace recording infrastructure VampirTrace [79], the open-source trace file format OTF [46] and the commercially available visualiser Vampir. Figure 2.3 shows Vampir's tiled user interface. Each so-called *display* presents a different view on the recorded information. The topmost display is the *Zoom Toolbar*, which provides simple, intuitive navigation. It presents a summarised view of the program activity over time. Colours encode different functions or groups of functions. In this example, inter-process communication functions are red and OpenMP [61] (multithreading-related) functions are light blue. The *Master Timeline* is located on the left. Processes, threads and graphics accelerator streams are arranged vertically. For each of these and each point in time, the currently active function is coded in colour. Arrows and black lines depict inter-process communication. The user can zoom in horizontally and vertically to restrict the shown time interval and processes. At the top-right, a flat profile is depicted as a bar chart. In this figure, it shows function groups instead of functions. The *Communication Matrix View* can be found at the bottom-right. It visualises different metrics related to inter-process communication for each pair of partners. Here, the number of messages, coded in colour, between processes is depicted.

Another tracing tool chain is Scalasca [22], which is developed at the Forschungszentrum Jülich. Aside from recording traces, it has built-in wait state detection and comes with the

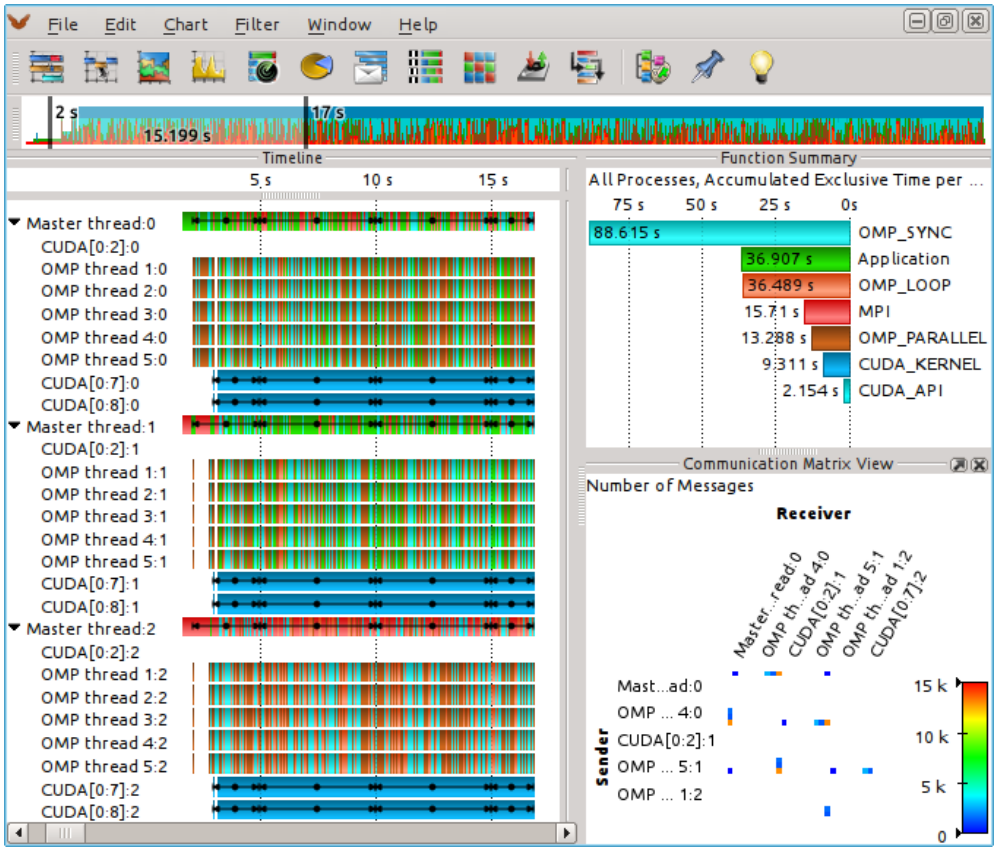

Figure 2.3: Example screenshot of Vampir visualising a trace. The left display shows which functions are active on different processes and threads during program execution. The top-right display depicts a profile. The bottom-right indicates how many messages a process exchanges with which other processes.

visualisation program Cube [72]. The jointly developed successor of, both, VampirTrace and Scalasca is the performance measurement infrastructure Score-P [5]. Along with it comes the new file format OTF 2.

Similarly to Vampir, the Intel Trace Analyzer [38] provides trace visualisation, but with a more limited set of features. Another trace viewer is Paraver [63]. It is more customisable than others and comes with a flexible file format that has no pre-defined semantics. Everything recorded can be combined and visualised in various ways.

Tau [71] is yet another tracing tool, and offers the full bandwidth of recording, analysis and visualisation.

Most of the mentioned tools use a hybrid approach between profiling and tracing. API calls of special interest are traced, whereas the rest of the program is recorded via sampling. Some tracing environments also support sampling. Most tracing tools can derive a profile from a trace file. Because of this, a clear distinction cannot be drawn.

The ultimate goal of performance analysis is to help the application developer find performance problems in his code. Programs are becoming increasingly parallel because

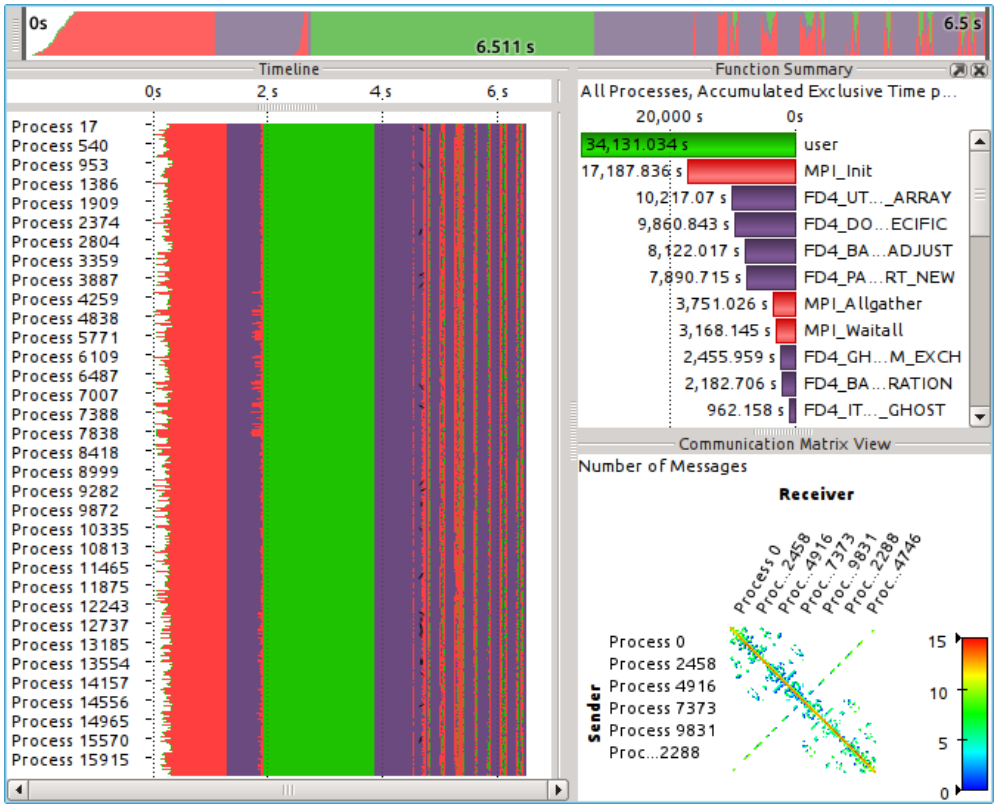

Figure 2.4: Visualisation where the number of processes exceeds the amount of horizontal and vertical pixels.

of rising core counts and the use of specialised accelerators like GPGPUs[3], FPGAs[4] and the Intel XEON Phi Coprocessor [87]. The resulting challenges for performance analysis tools can be divided into two categories—data structures and algorithms on the one hand, and visualisation on the other. Tools developers need to design for limited main memory size and need to achieve scalability with respect to the degree of parallelism used in the to-be-analysed application, the degree of parallelism used for analysis and the size of the trace file. Visualisation tools need to cope with a limited number of pixels on the screen and achieve scalability with respect to the degree of parallelism employed in applications. Cases where plain display of performance data is problematic already exist. Figure 2.4 is a screenshot of a visualisation for 16,384 processes. Screens do not have as many pixels horizontally nor vertically. In the near future, there will exist applications with millions of concurrent threads of execution.

Performance analysis tools mostly present raw data. Profiles accumulate over all threads and each function call. Trace visualisers present timelines, profiles and communication matrices. Most tools lack the ability to compare measurements between threads or different function calls. Surveying the use of comparative techniques in performance analysis will be the topic of the next section.

[3]General-purpose computing graphics processing units
[4]Field-programmable gate array

2.2 Comparative Performance Analysis

Being able to compare performance measurements helps in a number of ways. Developers can improve the visual scalability of existing tools by, for example, collapsing similar processes and just displaying one representative. This saves precious screen space. On the other hand new visualisations, which display comparison information, can be implemented. Showing timing differences between processes helps exposing load imbalance problems in parallel applications. In-depth comparison of timing before and after an optimisation pass aids developers understanding their programs. To analyse hardware differences, fine-grained comparison of program runs between platforms or perhaps different compilers seems worthwhile.

Open|SpeedShop [70] calculates differences between profiles. Similarly, eGprof [69] compares two GNU gprof profiles, and can also show differences between call trees. A *call tree* is a tree representing all encountered function call stack configurations. Aside from analysing differences between profiles, Cube can also combine performance information of multiple program runs into new theoretical ones that use untried program configurations. The goal here is to reduce the number of test runs necessary to analyse a large number of configurations of a program.

Vampir can display an arbitrary number of traces side-by-side to simplify visual inspection of differences. Intel Trace Analyzer is able to do the same, but additionally computes differences and speedup based on profiles.

By evaluating differences between specific performance metrics over time, Mohror and Karavanic [58] are able to detect segments of a program run, like loops and I/O phases. This information is then used to only store one representative segment and thus reduce trace file size and analysis overhead. Casas et al. [12] suggest to construct discrete signals from sampled metrics, e.g. the signal of active number of compute processes over time. Wavelet and correlation analysis, known from the field of signal processing, is then performed on these signals to differentiate phases of a program run and speed up analysis by focussing on representative regions.

Knüpfer [48, 45] proposes the *Compressed Complete Call Graph*. This in-memory data structure behaves like a trace, but internally uses a lossy compression scheme that relies on similarities in function call structure and relative timing. The technique is also used to highlight repeating function call patterns in a prototype visualisation. Ratn et al. [66] employ comparison techniques to achieve lossless compression of communication traces.

Weber et al. [82, 83] apply sequence alignment algorithms to the function call stack over time to detect differences between processes. Based on this, similarity metrics and a prototype visualisation are devised. Juckeland and Reiniger [42, 67] introduced the so-called *Parallel Program Flow Graph*, which unifies the sequence of active functions over time of multiple processes. Based on this, they propose a visualisation scheme that saves screen space compared to current timeline views. To compare the function call sequences Knüpfer's and Weber's methods are used.

This section suggests that differential profile analysis is well researched, but implementations exist almost exclusively in academia. Differential trace analysis and visualisation is young and only prototype implementations exist.

As a special type of comparative technique, the next section gives an overview about the use of clustering in performance analysis and motivates the development of specialised, more scalable clustering schemes.

2.3 Clustering

One goal of comparative analysis is to group similar processes into clusters. Jain [40] gives an excellent overview about currently employed techniques. Common clustering algorithms, for example k-means [17, 52, 55, 74] and DBSCAN [15], take as input n points in a d-dimensional space or a matrix containing the similarity of each pair of points plus the goal number of clusters k. The output is a set of disjoint sets, called clusters, each of which contains points considered similar. Here, most of the time points are processes and the dimensions are performance metrics.

Juckeland and Reiniger's method, mentioned in the previous section, provides a clustering in the sense that equal functions are merged and processes are, thus, clustered.

Gonzales et al. [25] describe a tool that characterises computation phases, the time between inter-process communication, using two metrics—instructions per second and completed instructions. It then clusters these periods, using the two metrics as dimensions, applying the DBSCAN algorithm. The result is a classification of phases, that differ in instructions per second executed in a certain amount of time. These periods are coloured differently to improve visualisation. In a follow-up paper [26] the authors employ alignment techniques to evaluate the quality of their structure detection approach for multiple cases.

To reduce the file size of so-called *effort traces*, Gamblin et al. [18] introduced a parallel version of the k-medoids [44] clustering scheme. The method is used to detect similar processes of which only one representative is stored. It is possible to reconstruct the complete effort trace afterwards.

Vampir's *Process Summary* display, proposed by Brunst [8], uses clustering to show a condensed view of the profiles of all individual processes. Figure 2.5 shows this display, along with the traditional, accumulated profile for context. To determine the clusters, k-means is used. The dimensions are the ten functions with the largest accumulated exclusive execution time. Each point is a process with its accumulated exclusive time for each of these functions as values. Users can choose the goal number of clusters, which defaults to the number of bars that fit into the available drawing height. The result is a clustering where processes sharing similar profiles are grouped together. Each cluster is represented as a stacked bar chart. On the left, next to the bars, the number of processes in the cluster and the positions of them among all processes is depicted. In each bar there is a staircase representation marking minimum, average and maximum of the accumulated exclusive execution time of this function among the processes in the cluster.

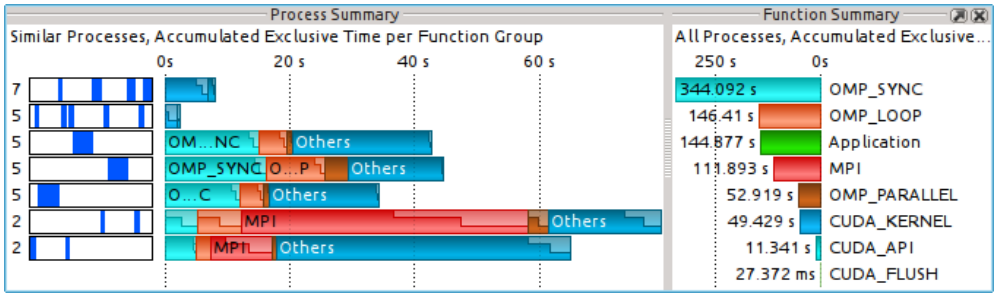

Figure 2.5: Vampir's Process Summary uses clustering to indicate similarities and differences between profiles of all individual processes. The right shows the traditional profile, which accumulates values for all processes.

Common clustering algorithms, including k-means and DBSCAN, take at least $\mathcal{O}(n^2)$ steps to cluster n processes. Calculating the similarity matrix, in preparation for clustering, requires an all-to-all comparison and, therefore, takes at least $\mathcal{O}(n^2)$ steps, too. For 50,000 processes, regardless of dimensionality and similarity measure, the number of steps is already very large. Common clustering algorithms, also, make assumptions about the data. For example, k-means requires clusters to be shaped roughly spherically. Using DBSCAN is problematic when different clusters differ in density. Another downside is that most algorithms need to be provided with a goal number of clusters, which leads to unnatural clusters. One suggested remedy is to apply clustering repeatedly and evaluate and compare the quality of the results. To obtain a natural clustering of processes in less than $\mathcal{O}(n^2)$ steps requires a different approach.

The following chapter introduces a set of techniques to efficiently cluster processes. The clustering, which depends only on profile information, is used to characterise traces of individual process. This characterisation can subsequently be used in a variety of ways including choosing representative processes and choosing similar processes for in-depth performance comparison.

3 New Methods for Comparative Performance Analysis

After providing some notation and general definitions, a novel structural similarity measure is proposed. The next section introduces an efficient mechanism to store the measure for all processes. Completing the picture, computational complexity properties of the introduced techniques are analysed. In the last section, a variation of the approach is used to detect a special case of dissimilarity.

3.1 Preliminaries

This section enumerates conventions that are used throughout the thesis.

- Numerous terms exist to describe what is essentially a thread of execution. Individual terms vary in technical details and in which context they are used. Examples include process, thread and CUDA stream [14]. To avoid the lengthy *thread of execution* and the ambiguous *thread*, throughout this thesis *process* is used to describe all of them. Usually, the difference between individual terms does not matter, if so, it is highlighted.
- In general, everything in particular sets, relations, domains and co-domains of functions are assumed to be finite. This does not restrict the applicability of what follows. It merely removes some of the intricacies of subsequent definitions and assertions.
- $\mathcal{P}$ denotes the set of all processes. Individual processes are labelled $P, P_1, P_2, P_3, \ldots$ or have actual names like `process 1` and `thread 3`. Similarly, $\mathcal{F}$ denotes the set of all program functions. Individual functions are labelled $F, F_1, F_2, F_3, \ldots$ or $A, B, C, D, \ldots$ or have actual names like `main` and `fopen`.
- Let S be a set. 2^S denotes the set of all subsets of S, also called the power set of S.

3.2 A Structural Similarity Measure

A *similarity measure* can be defined as a function $s : \mathcal{P} \times \mathcal{P} \to [0, 1]$ which satisfies the following conditions:

$$\begin{aligned}
&\text{for all } P_1, P_2 \in \mathcal{P} : 0 \leq s(P_1, P_2) \leq 1 \\
&\text{for all } P_1, P_2 \in \mathcal{P} : s(P_1, P_2) = s(P_2, P_1) \\
&\text{for all } P \in \mathcal{P} : s(P, P) = 1
\end{aligned}$$

That means, it maps two processes to a real number between 0 and 1, is symmetric and yields 1 if both arguments coincide. The lower the number, the less similar two processes are with regard to the chosen measure.

One approach to capture a process's structure is to record the functions it invokes. The mapping *functions* assigns each process the set of functions it calls during a program run. Formally:

$$\text{functions} : \mathcal{P} \to 2^{\mathcal{F}}$$
$$\text{functions}(P) := \{F \in \mathcal{F} \mid F \text{ is called at least once on } P\}$$

Processes that have many functions in common are likely similar. Therefore, using the Jaccard index [39], structural similarity of two processes can now be expressed as the number of functions they have in common divided by the number of functions either process calls. More formally:

$$\text{functionsimilarity}(P_1, P_2) := \frac{|\ \text{functions}(P_1) \cap \text{functions}(P_2)\ |}{|\ \text{functions}(P_1) \cup \text{functions}(P_2)\ |}$$

If both sets are empty, the result is defined to be 1. As required for being a similarity measure, the result is always between 0 and 1. The measure is also symmetric, since union and intersection of sets is commutative. Similarly, comparing a process to itself yields 1, because union and intersection of sets are idempotent.

Figure 3.1 depicts two example call trees that serve as an illustration for the proposed measure. Call trees represent all encountered function call stack configurations of a program run. Here, the function sets are identical, since both processes call the same functions, i.e.:

$$\begin{aligned}\text{functions}(\texttt{process 1}) &= \text{functions}(\texttt{process 2}) \\ &= \{\texttt{main}, \texttt{MPI\_Init}, \texttt{fopen}, \texttt{fclose}\}\end{aligned}$$

Therefore, similarity according to function similarity is 1.

$$\text{functionsimilarity}(\texttt{process 1}, \texttt{process 2}) = 1$$

A more complex example will be provided below. Before that, a second structural similarity measure is introduced.

Considering only called functions is limited in expressiveness. What a program function computes also depends on supplied arguments, global state and external influences. Incorporating these factors is hard due to the sheer size of argument domains, number of different states and limited predictability. Therefore, a simpler variant providing some context to function invocations is proposed. It uses function pairs instead of single functions. Such a pair consist of a function which calls a function. Formally:

$$\text{functionpairs} : \mathcal{P} \to 2^{\mathcal{F}\times\mathcal{F}}$$
$$\text{functionpairs}(P) := \{(F_1, F_2) \in \mathcal{F} \times \mathcal{F} \mid F_1 \text{ calls } F_2 \text{ at least once on } P\}$$

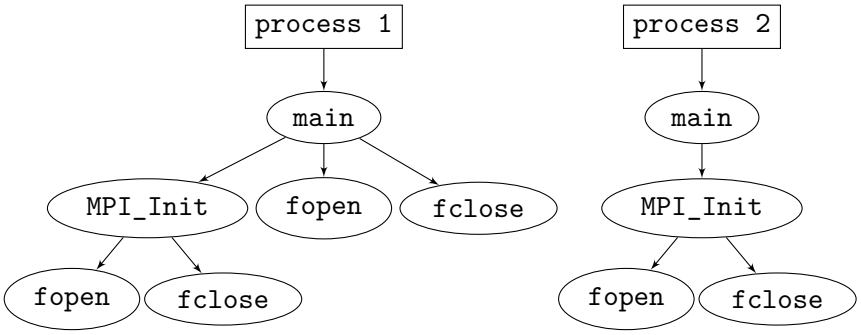

Figure 3.1: Call trees for two processes of an example program run

The previously introduced principles still hold for pairs, thus *function pair similarity* is defined as follows:

$$\text{functionpairsimilarity}(P_1, P_2) := \frac{|\ \text{functionpairs}(P_1) \cap \text{functionpairs}(P_2)\ |}{|\ \text{functionpairs}(P_1) \cup \text{functionpairs}(P_2)\ |}$$

To improve readability, $F_1 \to F_2$ denotes the pair (F_1, F_2).

An example, using the above call trees (Figure 3.1) shall illustrate the measure. The function pair set of each process contains exactly the directed edges of the corresponding call tree, i.e.:

$$\begin{aligned}\text{functionpairs}(\texttt{process 1}) = \{&0 \to \texttt{main},\ \texttt{main} \to \texttt{MPI\_Init},\\ &\texttt{main} \to \texttt{fopen},\ \texttt{main} \to \texttt{fclose},\\ &\texttt{MPI\_Init} \to \texttt{fopen},\ \texttt{MPI\_Init} \to \texttt{fclose}\}\\ \text{functionpairs}(\texttt{process 2}) = \{&0 \to \texttt{main},\ \texttt{main} \to \texttt{MPI\_Init},\\ &\texttt{MPI\_Init} \to \texttt{fopen},\ \texttt{MPI\_Init} \to \texttt{fclose}\}\end{aligned}$$

Set semantics removes duplicate elements. Incidentally, there are no duplicate pairs in this example. Because, in principle, processes can have multiple functions on the base level, not just `main`, a virtual root function called 0 is used. Another problem without this function, would be if the call stack had depth 1. No pairs would then be recorded. To obtain the function pair similarity, first the intersection and union of both function pair sets has to be computed.

$$\begin{aligned}&\text{functionpairs}(\texttt{process 1}) \cap \text{functionpairs}(\texttt{process 2}) = \text{functionpairs}(\texttt{process 2})\\ &\text{functionpairs}(\texttt{process 1}) \cup \text{functionpairs}(\texttt{process 2}) = \text{functionpairs}(\texttt{process 1})\end{aligned}$$

Next, one can take the size of the sets

$$\begin{aligned}|\ \text{functionpairs}(\texttt{process 1}) \cap \text{functionpairs}(\texttt{process 2})\ | &= 4\\ |\ \text{functionpairs}(\texttt{process 1}) \cup \text{functionpairs}(\texttt{process 2})\ | &= 6\end{aligned}$$

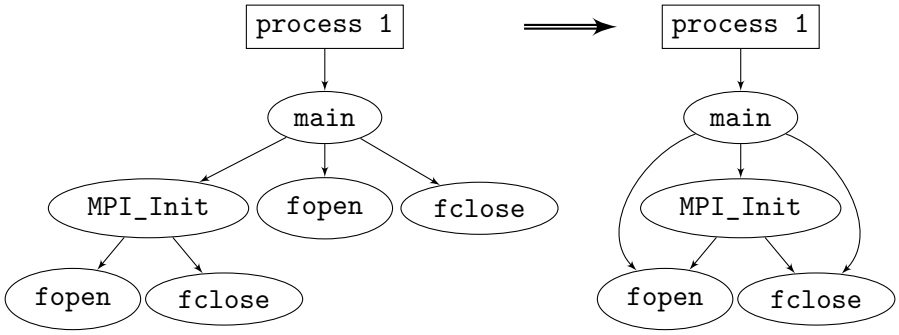

Figure 3.2: Example compressed call tree for one process

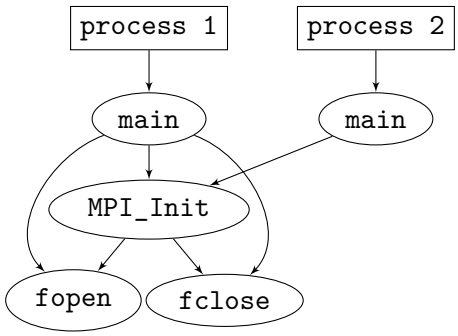

Figure 3.3: Example compressed call tree for two processes

and, finally, obtain the desired quotient.

$$\begin{aligned} &\text{functionpairsimilarity}(\texttt{process 1}, \texttt{process 2}) \\ &\quad = \frac{|\ \text{functionpairs}(\texttt{process 1}) \cap \text{functionpairs}(\texttt{process 2})\ |}{|\ \text{functionpairs}(\texttt{process 1}) \cup \text{functionpairs}(\texttt{process 2})\ |} \\ &\quad = \frac{2}{3} \approx 66.6\% \end{aligned}$$

The processes have four function pairs of a total of six in common. Therefore, the similarity according to this measure is approximately 66.6%. In this work, the focus lies on function pair similarity. For brevity and ease of presentation, the measure based on single functions is sometimes used.

One alternative to the above techniques is to compare call trees. Andreas Knüpfer [48], introduced in Section 2.2, uses tree comparison to compress trace files. Adapting the approach for call trees, the idea is as follows. Take the call tree and unite all nodes, bottom-up, that have identical sub-trees. Using the example from above (Figure 3.1), compressing process 1's call tree yields the one depicted in Figure 3.2. Next, process 2 can be added and then compression can be applied again. The result is presented in Figure 3.3. One straightforward way to measure similarity of compressed call trees is to divide the number of shared nodes by the number of all nodes.

Merging only absolutely equal sub-trees proved to be subpar for quantifying similarity. Oftentimes, there are small differences in sub-trees which results in nodes not being

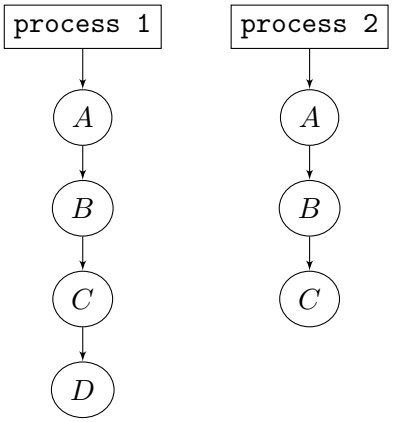

Figure 3.4: Example Call tree that would not be merged using sub-tree matching

merged. An example, using the call trees in Figure 3.4, illustrates this. Processes 1 and 2 are equal to the point where function C calls additionally D on process 1. No nodes would be merged, since the trees do not contain any equal sub-tree. The resulting similarity is 0. Using function pair similarity, instead, yields a more reasonable $\frac{3}{4}$. Any similarity measure for compressed call trees based on the number of shared nodes has this problem.

If call trees are equal, function pair similarity as well yields 1, since having equal sub-trees implies identical function pair sets. Depending on how recursion is dealt with, function pair similarity of two processes is almost always higher than call tree similarity (number of shared nodes divided by number of all nodes).

Function pair similarity is influenced by the functions a processes calls and where they are called from. Inlining functions, for example, changes the similarity value. The number of calls and their timing does not influence the outcome of comparing two processes. It, therefore, also ignores how many iterations of a loop are done. If calls are reordered inside a function, similarity does not change. Recursion is problematic for some analysis tools and workflows, since the resulting call trees and stacks can get very deep and, compared to the structure of the source code, falsely represent program structure. Function pair similarity handles this gracefully, because if F is recursive, only the pair $F \rightarrow F$ is added. Recursion depth is ignored.

Concluding the previous observations, function pair similarity evaluates structure, regardless of timing, iteration counts and recursion depth. Processes with 100% similarity form groups. The number of such groups is supposed to be stable with increasing process counts, and thus should provide a scalable grouping. Statically compiled source code is expected to not call different functions when problem size or process numbers vary [2, 6, 41]. Chapter 4 analyses whether this expectation holds in practice.

Mind that applying the measure is not restricted to processes of one program run. It can also be employed to compare multiple runs and even different programs. Similar processes can, therefore, be identified across runs in preparation for in-depth comparative performance analysis.

To be useful for large numbers of processes, calculating the measure for all pairs of processes in sub-quadratic time is key. In order to achieve this, per-process information, required to compute the measure, needs to be stored in a manner that reduces the data so that subsequent querying for similarity values is quick. Alternatively, one can store everything in a straightforward manner and devise a sub-quadratic querying algorithm. The next section elaborates how efficient similarity storage and computation is achieved.

3.3 Efficient Storage and Calculation of Set-Based Metrics

To calculate similarity, the set of function pairs for each process is needed. Imagine straightforwardly storing all processes and set elements in a two-dimensional matrix. The matrix elements are crosses signalling whether or not an element belongs to the process. Further, suppose there are 1,000,000 processes with up to 1,000 set elements, and elements are identified using a 32-bit value. Storing the whole table would then take roughly four gibibytes[1] of storage space. Using bit vectors, one could lower the storage need. But this increases handling complexity and does not improve scalability. Even requiring 100 mebibytes is still large considering the technique is used to just classify processes in preparation for deeper analysis. Thus, the primary goal was to find a way to store all set elements and processes with few or no duplicates.

The *concept lattice*, known from the field of *formal concept analysis* [20] emerged as a suitable data structure. A concept lattice is a type of *order*. A *total order* $R \subseteq X \times X$ on a set X, for example, is a binary relation that is transitive, antisymmetric and total. Formally:

$$\begin{aligned}
&\text{for all } a,b,c \in X &&: \text{ if } (a,b) \in R \text{ and } (b,c) \in R \text{ then } (a,c) \in R && \text{(transitivity)}\\
&\text{for all } a,b \in X &&: \text{ if } (a,b) \in R \text{ and } (b,a) \in R \text{ then } a = b && \text{(antisymmetry)}\\
&\text{for all } a,b \in X &&: (a,b) \in R \text{ or } (b,a) \in R && \text{(totality)}
\end{aligned}$$

An example is the natural numbers ordered by $\leq$ (Figure 3.5). Removing the need for totality yields the partial order. A *partial order* $R \subseteq X \times X$ on a set X is a binary relation that is transitive, antisymmetric and reflexive. Reflexivity means each element in X is related to itself. Formally:

$$\text{for all } a \in X : \ (a,a) \in R$$

An example of such an order is the natural numbers ordered by divisibility (Figure 3.6).

Another kind of order is the bounded lattice. A *bounded lattice* R on a set X is a finite partial order where for each pair of elements in X there exists a unique least upper bound $\vee$ (also called supremum or join) and a unique greatest lower bound $\wedge$ (also called

[1] $1,000,000 \times 1,000 \times 4$ bytes $= 4,000,000$ bytes

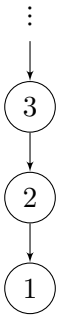

Figure 3.5: Example totally ordered set: $(\mathbb{N}, \leq)$

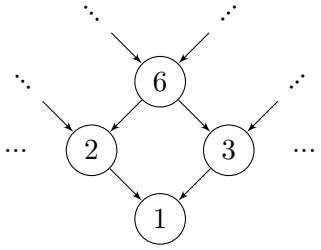

Figure 3.6: Example partially ordered set: $(\mathbb{N}, |)$

infimum or meet). A supremum s of two elements $a, b \in X$ is an element which satisfies the following:

$$(a, s) \in R \text{ and } (b, s) \in R$$
$$\text{for all } x \in X: \text{ if } (a, x) \in R \text{ and } (b, x) \in R \text{ then } (s, x) \in R$$

It is the smallest element that is larger than both a and b according to R. Similarly, an infimum i of two elements $a, b \in X$ satisfies:

$$(i, a) \in R \text{ and } (i, b) \in R$$
$$\text{for all } x \in X: \text{ if } (x, a) \in R \text{ and } (x, b) \in R \text{ then } (x, i) \in R$$

These properties imply that there is an absolute largest element and an absolute smallest one. An example is the set $\{A, B, C\}$ ordered by $\subseteq$ with the join operation being the set union and the meet being set intersection (Figure 3.7).

Extending this, a concept lattice consists of two interlocked bounded lattices. One is the complement of the other. Formally, a *formal context* [20] is a triple (O, A, I) where O is the set of objects, A the set of attributes and $I \subseteq O \times A$ the incidence relation. Each object has a set of attributes. For the use case of comparing processes, the set of objects is the set of processes, and attributes are function pairs or functions. A formal context defines a *concept lattice* by specifying so-called *concepts*, and a partial order on them. A concept lattice can be represented as a graph where concepts are nodes and the order determines the edges. In the context of this thesis, it is not important to distinguish

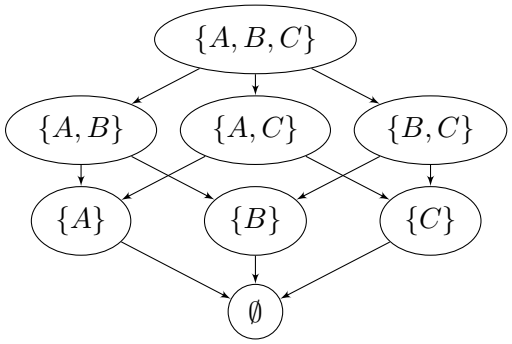

Figure 3.7: Example bounded lattice: $(\{A, B, C\}, \cup, \cap, \emptyset, \{A, B, C\})$

between formal context and concept lattice. They represent the exact same information.

An example shall illustrate this formalism. Let $(\mathcal{P}, \mathcal{F}, \mathcal{I})$ be a formal context where

$$\mathcal{P} = \{P_1, P_2, P_3, P_4\}$$
$$\mathcal{F} = \{A, B, C\}$$
$$\mathcal{I} = \{(P_1, A),\ (P_2, B),\ (P_2, C),\ (P_3, A),\ (P_3, B),\ (P_3, C),\ (P_4, B),\ (P_4, C)\}$$

holds. The incidence relation $\mathcal{I}$ can also be represented as a table (Table 3.1). P_1 calls just A. P_2 and P_4 both call B and C. P_3 calls all three functions. An algorithm then extracts the concepts and the partial order on them from the formal context. The resulting concept lattice is depicted in Figure 3.8. It can be read as follows:

- 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 call the empty set of functions.
- The bottom node signifies that only P_3 calls all functions, and in particular that it calls every function which P_1, P_2 and P_4 call.
- P_1 is different from P_2 and P_4.
- P_2 and P_4 call the same functions.

This graph contains redundant information. Removing it, without changing its meaning, yields the more readable graph in Figure 3.9. It can be interpreted as follows:

- No process calls a common subset of functions.
- P_1 calls A.
- P_2 and P_4 call the same functions, namely B and C.
- P_3 subsumes all other processes and, thus, calls A, B and C.

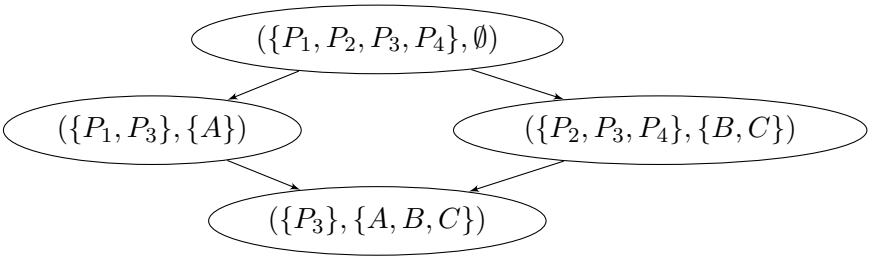

Figure 3.8: Example concept lattice including redundant information

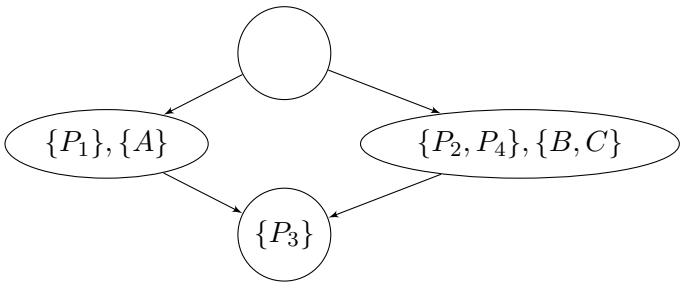

Figure 3.9: Example concept lattice excluding redundant information

A concept lattice is a directed acyclic graph of function inheritance. Nodes inherit attributes from others that are reachable along upward-edges. Each object and attribute is contained in the concept lattice exactly once.

Two examples shall give an intuition on concept lattices for real-world application runs. The first one, depicted in Figure 3.10, is the result of a WRF [57, 85] weather simulation run using 64 processes. Each node contains a set of processes and function pairs. The size of each set is given in parenthesis, followed by a few example elements. Because internally, functions are represented by integer identifiers and to save space, numbers are used to denote functions. In this example, every process except the first has the same function pairs. Process 1 has 111 additional pairs. The second example (Figure 3.11) shows the concept lattice of function pairs for an AMG2006 [4] run, a differential equation solver, using 512 processes. This lattice has 14 nodes and 21 edges, and is therefore larger and more complex than the previous lattice.

Algorithms to create concept lattices are involved and unintuitive. Because of this and since the techniques are used without modification, elaboration on how they operate is foregone. The interested reader is pointed towards the websites of Uta Priss [64] and

Table 3.1: Example incidence relation: $\mathcal{I}$

	A	B	C
P_1	×		
P_2		×	×
P_3	×	×	×
P_4		×	×

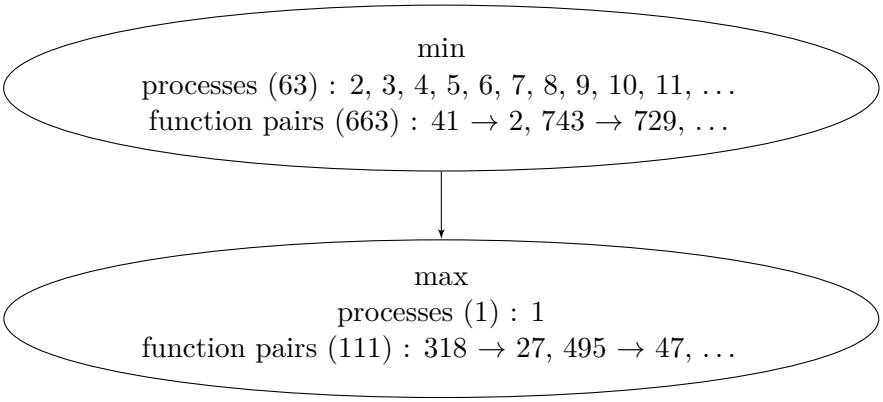

Figure 3.10: Concept lattice of function pairs for a WRF weather simulation run utilising 64 processes

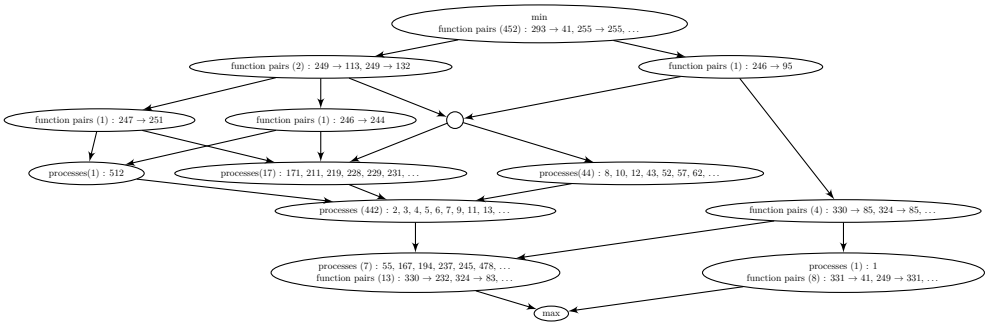

Figure 3.11: Concept lattice of function pairs for an AMG2006 differential equation solver run utilising 512 processes

Bernhard Ganter [21], and the textbook [20]. One algorithm to create a concept lattice is, for example, *Ganter's Next Closure* algorithm [19, 20]. It requires the complete incidence relation to be present in main memory. As previously stated, these tables can get very large—larger than available main memory. To avoid this, an iterative algorithm by Dean van der Merwe, Sergei Obiedkov, and Derrick Kourie [80]—from here on called *van der Merwe's algorithm*—is used.

After creating the concept lattice, the next step is to extract similarity values for each pair of processes from it. The result is a so-called *similarity matrix* which is a matrix of equivalence classes, since processes on the same lattice node yield the same values when compared to other processes. Function similarity, which is defined as

$$\text{functionsimilarity}(P_1, P_2) := \frac{|\ \text{functions}(P_1) \cap \text{functions}(P_2)\ |}{|\ \text{functions}(P_1) \cup \text{functions}(P_2)\ |}$$

and analogously function pair similarity, can be obtained as follows. $\text{functions}(P)$ is the union of function sets of nodes reachable along upward-edges starting at P's node. $\text{functions}(P_1) \cap \text{functions}(P_2)$ is the union of function sets of nodes reachable from both P_1's and P_2's node along upward-edges. $\text{functions}(P_1) \cup \text{functions}(P_2)$ is the union of

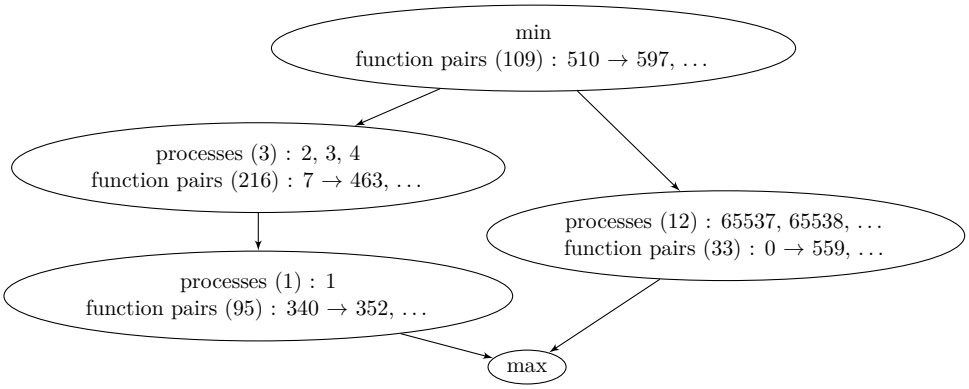

Figure 3.12: Concept lattice of function pairs for a WRF weather simulation run utilising four processes and twelve threads

function sets of nodes reachable from either P_1's or P_2's node along upward-edges. Since each function is contained in the chosen lattice representation exactly once, instead of calculating the union of sets, it suffices to take the sum of set sizes. This yields the desired cardinality. This approach works for functions and function pairs alike. In fact, many more set-based measures can be handled this way.

An example illustrates the procedure of obtaining the similarity matrix from a lattice. Figure 3.12 shows the concept lattice of a WRF run using four processes and twelve threads. Before extracting similarities, the following explains how to read the shown lattice:

- Processes have identifiers 1–4, threads have 65,537 and above.
- There are three groups of processes/threads: process 1, processes 2–4 and a group containing all threads.
- All processes and threads have 109 function pairs in common.
- Process 1 subsumes processes 2, 3 and 4.
- The twelve threads are in a group of their own and differ from the processes substantially.

To obtain the function pair similarity of processes 1 and 2, the following value needs to be calculated:

$$\frac{|\,\text{functionpairs}(\texttt{process 1}) \cap \text{functionpairs}(\texttt{process 2})\,|}{|\,\text{functionpairs}(\texttt{process 1}) \cup \text{functionpairs}(\texttt{process 2})\,|}$$

functionpairs(`process 1`) are the function pairs contained in the three nodes on the left reachable from process 1's node along upward-edges. functionpairs(`process 2`) are the two nodes reachable from process 2's node along upward-edges. The intersection of both is, therefore, the upper two nodes, whereas the union also includes the function pairs from

Table 3.2: Similarity matrix of the WRF example run

	Process 1	Process 2, 3, 4	Threads
Process 1	1	0.77	0.24
Process 2, 3, 4	0.77	1	0.30
Threads	0.24	0.30	1

process 1's node. Putting it all together, function pair similarity is about 77%. More formally:

$$\text{functionpairsimilarity}(\texttt{process 1}, \texttt{process 2}) = \frac{109 + 216}{109 + 216 + 95} \approx 77\%$$

Function pair similarity of processes 1 and 3, as well as 1 and 4 are the same as for processes 1 and 2.

As for comparing processes 2–4 to the threads, the intersection in the similarity formula consists of the function pairs from the uppermost node. The union contains the ones from the topmost three nodes. Function pair similarity is therefore:

$$\text{functionpairsimilarity}(\texttt{process 2}, \texttt{thread 1}) = \frac{109}{109 + 216 + 33} \approx 30\%$$

The resulting similarity matrix is depicted in Table 3.2. Since it is symmetric and every diagonal entry is 1, it would suffice to display the three bottom-left entries.

The next section discusses complexity theoretic properties and possible parallelisation of the introduced approach.

3.4 Scalability Considerations

Obtaining the similarity of each pair of processes requires multiple steps. First, the lattice is constructed, from which in a second step similarities are extracted. To construct the concept lattice, for each process the set of function pairs has to be inserted using van der Merwe's algorithm. Next, for each node in the concept lattice that has at least one process, and each other node in the concept lattice that has at least one process the graph is traversed upwards. During this traversal, the numerator and denominator from the similarity formula for this pair of process groups are obtained. After finishing the traversal, the fraction is calculated.

The worst-case space complexity of storing a concept lattice, as shown by Godin, Missaoui, and April [24], is $\mathcal{O}(2^a p)$ where a is the total number of attributes and p the process count. Therefore and according to van der Merwe et al. [80], the worst-case time complexity of generating the lattice and computing the similarity matrix is $\mathcal{O}(2^a p^3 a)$. The latter bound is not tight.

In practice, as mentioned in Section 3.2, processes are expected to fall into few, i.e. a constant number of, groups. The attribute count is constant as well, since the number of

functions in a program does not increase with rising process counts or increasing problem size. The space complexity of storing the lattice is, therefore, expected to be independent of the process and attribute count, i.e. $\mathcal{O}(1)$. Because similarity matrix computation only depends on the number of nodes in the lattice, the most expensive part of the procedure is inserting each process's function pairs into the lattice. Therefore, the expected time complexity of lattice creation and similarity extraction is $\mathcal{O}(p)$ with p being the process count.

Being faster than linear in the number of processes is not possible when executing the steps serially, since some work has to be done for each process. One can argue that linear time is not good enough for large process numbers. Which is why parallelisation of the approach should be considered. Performance analysis tools for high performance computing, especially tracing tools, heavily depend on parallelism to cope with the large amount of data and associated processing. Algorithms to generate concept lattices in parallel already exist [35, 51, 60]. They suggest that an algorithm creating the concept lattice in expected $\mathcal{O}(\log p)$ steps can be devised. One straightforward solution would be to first create multiple lattices in parallel. These intermediate lattices can then be merged into a final lattice in a tree-esque or all-to-one fashion. Merging in this case means inserting each node from the previous lattice into the new one. Each process in a node is known to be equal according to the similarity measure and, therefore, multiple processes can be inserted in one step. Speeding up the computation of the similarity matrix is straightforward, since each cell can independently be calculated.

The next section uses the introduced techniques with a twist to detect a special class of differences between processes.

3.5 The Subsumption Measure

Slightly modifying the shown techniques makes it possible to discern whether processes are dissimilar because one does additional work, i.e. performs extra function calls. More specifically, the following cases can be distinguished from other differences:

- Function calls are replaced by their implementation (*inlining*).
- One process calls additional functions, for example when printing debug output or writing to disk.
- Processes have different levels of tracing detail.
- Threads are spawned from a process using the fork-join model.

This distinction is useful for further differential analysis, and generally automatic comparison of processes inside one and between multiple program runs.

An example (Figure 3.13) illustrates the idea. Process 1 calls A, which then calls B. For process 2, A has been inlined and B is called directly. The resulting concept lattice

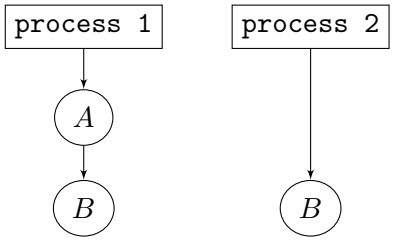

Figure 3.13: Example call trees, where function A has been inlined in process 2

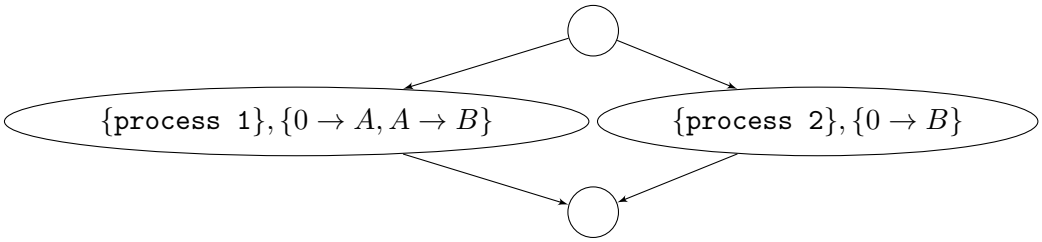

Figure 3.14: Concept lattice of function pairs for the example call trees

is depicted in Figure 3.14. Function pair similarity yields 0%, because processes 1 and 2 have no pairs in common, although there is a certain similarity.

The proposed *subsumption measure* modifies the previous approach as follows. The first step is to take the transitive closure $(\ )^+$ of the function pair set for each process. *Transitive closure* of a relation $R \subseteq X \times X$ is commonly defined as:

$$R^+ := \bigcup_{i \geq 1} R^i$$
$$R^1 := R$$
$$R^{j+1} := R \circ R^j$$

where $j \geq 1$ and $\circ$ denotes the composition of relations, i.e.:

$$S \circ T := \{(a, c) \in X \times Z \mid \text{ there exists } b \in Y : (a, b) \in T \text{ and } (b, c) \in S\}$$

with $S \subseteq Y \times Z$ and $T \subseteq X \times Y$. This means, function calls with intermediate steps are added to each process's function pair set. In the example, $0 \to B$ is added to process 1's set. Thus, the transitively closed function pair sets for processes 1 and 2 are:

$$\text{functionpairs}(\texttt{process 1})^+ = \text{functionpairs}(\texttt{process 1}) \cup \{0 \to B\}$$
$$\text{functionpairs}(\texttt{process 2})^+ = \text{functionpairs}(\texttt{process 2})$$

Scalability does not suffer from this step, since the number of function pairs is constant, the transitive closure can be computed for all processes independently, and the size of the resulting lattice does not increase. Informally, differences between function pair sets cause lattice nodes to be split. Taking the transitive closure of two function pair sets potentially

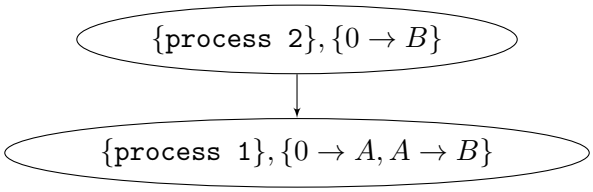

Figure 3.15: Concept lattice of the transitively closed function pair sets for the example call trees

lessens differences between them, but never creates new differences. The second step is to construct the lattice the same way as before. For the example, the resulting lattice is depicted in Figure 3.15. The third and last step is to calculate the measure for each pair of process groups. This approach uses a different measure, a specialisation of the Tversky index [78]:

$$\text{functionpairsubsumption}(P_1, P_2) := \frac{|\ \text{functionpairs}(P_1)^+ \cap \text{functionpairs}(P_2)^+\ |}{|\ \text{functionpairs}(P_2)^+\ |}$$

If P_2's function pair set is empty, the result is defined to be 1. For the example, the function pair subsumption measure is 1. This means that, indeed, process 1 does at least as much work as process 2. Asking the reverse yields $\frac{1}{3}$, because process 2 calls fewer functions than process 1. In particular, this means that the measure is not symmetric and, thus, no similarity measure according to the definition. The modified formula is motivated by the following thoughts:

- Imagine scoring subsumption and exclusion element-wise. Subsumption is counted using positive points and exclusion using negative points. The measure can, then, be expressed as a fraction of positive points over both positive and negative points.
- If P_2 carries out work that P_1 does not, then this must decrease the measure. It, therefore, counts negatively.
- If P_1 performs work that P_2 does not, that is to be ignored. Counting them as positives would cause the measure to be arbitrarily close to 1 when P_1's function pair set is much larger than P_2's, despite for example them being disjoint.
- Only common function pairs between both sets count positively. Therefore, the numerator of the formula is the intersection of both sets.
- The denominator is the numerator plus the number of function pairs that P_2 alone has. In sum, that is the size of P_2's set, possibly overlapping with P_1's.[2]
- Thus, if P_1's set includes all of P_2's function pairs, the intersection is the same as P_2's set and the resulting fraction is 1.

[2] $(A \cap B) \cup (B \setminus A) = B$ where A and B are sets

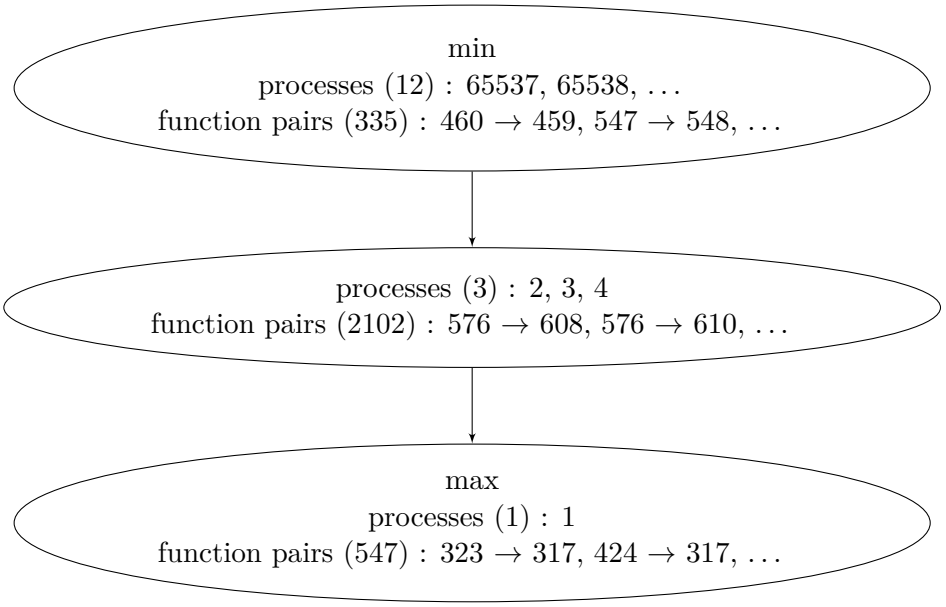

Figure 3.16: Concept lattice of the transitively closed function pair sets for a WRF weather simulation run utilising four processes and twelve threads

As a real-world example, take the previous WRF weather simulation (Figure 3.12 on page 21). Applying the described techniques, yields the concept lattice depicted in Figure 3.16. Querying it for function pair subsumption, yields that processes 1–4 subsume the threads' function pairs. One can, therefore, safely assume that these twelve threads perform a subset of the work the processes do. They are not as different as the 24% and 30% similarity in Table 3.2 on page 22 suggests. Mind that this conclusion cannot be drawn without transitively closing the function pair sets. The relation between process 1 and 2–4 does not change, because even without the modification, process 1 subsumes processes 2–4.

The next chapter demonstrates the applicability and scalability of the introduced techniques. Furthermore, in addition to improving an existing visualisation, a new one leveraging function similarity values is presented.

4 Evaluation

This chapter assesses the ability of the introduced techniques to aid software performance analysis and visualisation. The first two sections evaluate their effectiveness and efficiency. The third one demonstrates the suitability of the subsumption measure to identify related processes despite apparent differences. The last part shows how similarity can be used to improve visualisation.

Although this chapter is centred around analysing traces, it is not mandatory to use tracing. Function or function pair sets can be obtained in many ways including sampling program runs or extracting them from given call trees. Care has been taken to exclude the effects of tracing in the following evaluation, so that observations, results and conclusions apply regardless of how function pairs are collected.

4.1 Applicability

Before evaluating scalability in greater detail, an overview of how the techniques perform on a variety of programs is given. Table 4.1 summarises the results of this broad assessment, which is subsequently discussed. For each application the concept lattice of function pairs is constructed and then the similarity between all pairs of processes is computed.

Table 4.1: Measured timing and lattice size for selected application traces.

	Trace			**Result**		
Application	Proc.	∼Size (MiB)	Function pairs	t_{eval} (ms)	Nodes	Groups
HPL	2,360	200	8	< 10	2	2
GROMACS	36	3,700	1,381	< 10	24	11
CCLM	180	650	180	< 10	6	3
COSMO-SPECS	100	250	50	< 10	1	1
WRF	64	300	774	< 10	2	2
FD4	65,536	1,000	55	55	22	14
HOMME	1,024	200	179	< 10	3	3
AMG2006	1,024	3,700	440	66	11	7
IRS	64	4,250	989	34	18	7
LULESH	432	350	406	49	182	35
ParaDiS	128	600	649	3,486	6,367	74
PIConGPU	39	300	474	11	60	17
BT-MZ	16	150	126	< 10	5	3
HPCC MPI-FFTE	128	1,200	109	70	7	4
PEPC	16,384	1,200	113	15	2	2

All measurements were done on a notebook featuring an Intel Core i5-2520M processor and dual channel DDR3 PC3-10600 RAM. For each program run the name of the application and three important parameters of the resulting trace are given. These three factors namely process count, trace size and total function pair count influence the timing and outcome of applying the techniques. The number of function pairs lies between the number of functions and its square. As before, processes include threads and CUDA [14] streams.

For each application the results are depicted on the right side of the table. The evaluation time t_{eval}, the number of nodes in the concept lattice and the number of process groups are of interest. The evaluation time consists of three steps. First, add each process's function pairs to the lattice. After this, finalise the lattice and compute the similarity matrix from it. It does not include obtaining the function pairs. Predicting timing is difficult, since it is determined by many factors namely the number of processes to be added, the size and structure of the lattice before each insertion, the number of function pairs to be added in each step and the final lattice's size and structure. The number of process groups equals the number of nodes with a non-empty process set, and is therefore equal to or smaller than the node count.

The test applications are a selection of benchmarks, pseudo-applications and real-world applications:

- HPL [34], which stands for "High-Performance Linpack", is a benchmark that solves systems of linear equations. It is used to determine the 500 most powerful, known computers on earth—the TOP500 list [76].
- GROMACS [30, 81], which stands for "Groningen Machine for Chemical Simulations", is a molecular dynamics program specialised in simulating proteins, lipids and nucleic acids. It can also be used for non-biological systems, for example polymers. GROMACS leverages parallelism using MPI [59], OpenMP [61] and CUDA.
- CCLM [68], which stands for "COSMO Climate Limited-area Modelling", predicts climate using the COSMO model [13]. COSMO is used, for example, by the Deutscher Wetterdienst to generate weather forecasts.
- COSMO-SPECS [31], improves the accuracy of COSMO's predictions by additionally simulating cloud microphysics.
- WRF [57, 85], which stands for "The Weather Research & Forecast Model" simulates weather, similar to COSMO.
- FD4 [49, 50] stands for "Four-Dimensional Distributed Dynamic Data structures" and adds dynamic load balancing to COSMO-SPECS and WRF. This particular trace is of a benchmark run that evaluates FD4's scalability.
- HOMME [33], which stands for "High-Order Methods Modeling Environment", is another weather model.

- AMG [4], which stands for "Algebraic Multigrid Solver", is a benchmark that solves systems of differential equations.
- IRS [37], which stands for "Implicit Radiation Solver" benchmarks how suited a computer is to solve systems of diffusion equations. AMG2006 and IRS are both part of a benchmark suite compiled at the Lawrence Livermore National Laboratory.
- LULESH [43, 53], which stands for "Livermore Unstructured Lagrangian Explicit Shock Hydrodynamics", is a pseudo-application that retains the numerical algorithms, data motion and programming style of a typical hydrodynamics simulation. It is used to evaluate the capability of a high performance computer to run such simulations.
- ParaDiS [9, 62], which stands for "Parallel Dislocation Simulator", simulates the behaviour of solids under external stress. This way one can, for example, predict where cracks appear and how they expand, move and join.
- PIConGPU [10], which stands for "Particle-In-Cell on Graphics Processing Units", is an implementation of the particle-in-cell method for clusters using NVIDIA graphics cards. It simulates the behaviour of charged particles accelerated by a laser.
- BT-MZ stands for "block tri-diagonal multi-zone" and is a pseudo-application emulating the workload of a typical fluid dynamics simulation. The term *multiple zones* means that it uses multiple processes and multithreading for parallelisation. BT-MZ is part of the NAS Parallel Benchmarks [7].
- HPCC MPI-FFTE is a benchmark that performs complex, one-dimensional, discrete Fourier transforms (DFT) using the FFTE package [16]. DFT is widely used in digital signal processing, in image processing and to solve partial differential equations. MPI-FFTE is included in the HPCC benchmark suite [54].
- PEPC [23], which stands for "Pretty Efficient Parallel Coulomb Solver", is an N-body simulation using an octree as internal representation. It simulates the interaction between a laser and charged particles.

For each trace, except one, the evaluation time is less than 0.1 second. Most of the time it is far smaller. This underlines the applicability of the introduced techniques. Constructing the lattice for the ParaDiS trace takes a very long 3.226 seconds and calculating the similarity matrix takes the remaining 0.26 seconds. Most of the time, the number of process groups is below ten. As expected, group counts vary seemingly independent of the number of processes. Thus, most applications exhibit fairly regular function call behaviour. Please note that groups contain processes equal according to the similarity measure. As indicated in Section 3.4, the number of steps required to calculate the similarity matrix from a given concept lattice is in $\mathcal{O}(g^2 n)$ where g is the number of groups and n the number of nodes in the lattice. Because of this, the size of the lattice for the ParaDiS run is of concern.

To investigate the findings and concerns more deeply, the next section provides further measurements for two applications, AMG2006 and ParaDiS.

4.2 Scalability

This section evaluates the introduced methods in greater detail using two representative applications from the previous section. The first one is AMG2006, which exhibits a semi-regular function call pattern as a result of employing iterative refinement techniques. The second one is ParaDiS, where the methods seem to struggle.

To recap, AMG2006 is a benchmark solving systems of differential equations. ParaDiS simulates solids under external stress.

4.2.1 Measurement Setup

Every performance measurement was conducted on the computer system Taurus (Figure 4.1) at the Technische Universität Dresden. The partition *sandy* was chosen for all tests. It features two Intel Xeon E5-2690 processors [86] (Sandy Bridge microarchitecture) and 32–128 gibibytes of RAM per motherboard. Hyper-Threading [36] is disabled system-wide. Measurements were done using one processor core at a fixed clock rate of 2.9 GHz, without Turbo Boost [77]. Even though only one core was used per measurement, the motherboard has been allocated for exclusive access to avoid the influence of other programs running on the same node. Every measurement was repeated five times and the median of them has been used. Minimum, average and maximum values were monitored for abnormalities.

Figure 4.1: Photo of the high performance computer Taurus, on which scalability testing of the introduced techniques was done

4.2.2 AMG2006

The traces generated for the measurements have up to 4,096 processes, are up to 60 gibibytes large, call between 207 and 217 distinct functions and have 436–483 different function pairs. The discrepancy of function numbers stems from different code paths being taken for different process counts.

Figure 4.2 shows the lattice size for increasing process numbers. One example for such a concept lattice is Figure 3.11 on page 20. The time it takes to construct the lattice, finalise it and extract the similarity for each pair of processes is depicted in Figure 4.3. Finalising the lattice, i.e. removing redundant function pairs and processes from it, takes almost all of this time. The rest, i.e. adding each process's function pairs and calculating the similarity matrix at the end, took under four milliseconds for each run. While iteratively constructing the lattice, van der Merwe's algorithm requires all duplicate function pairs

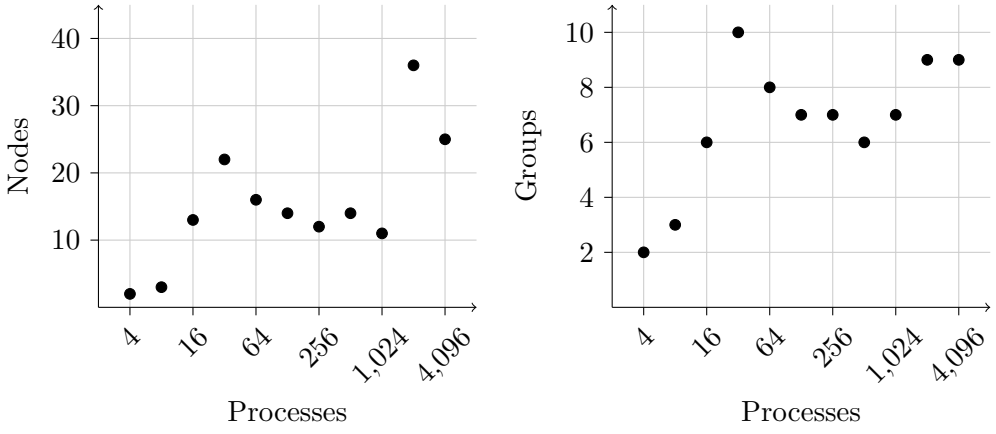

Figure 4.2: Number of nodes and groups in the concept lattices for AMG2006 runs

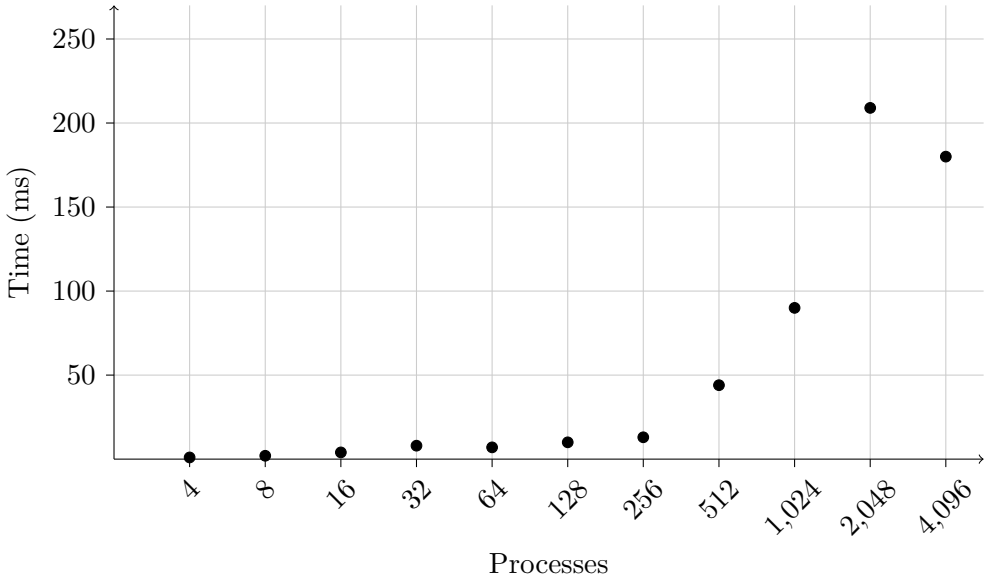

Figure 4.3: Timing of the concept lattice construction and evaluation for AMG2006 runs

(as shown in Figure 3.8 on page 19) to be present. Only after constructing the final lattice it is allowed to remove them, so that (as for example in Figure 3.9 on page 19) each process and function pair is contained exactly once.

The largest number of nodes observed is 36, whereas group counts are at most ten. There is no definite upward or downward trend. Numbers vary in an expected fashion. Different process counts can cause different domain decompositions and therefore different iterative refinement behaviour as well. It is therefore not surprising that the lattice size is unsteady. Generally speaking, the techniques are expected to show this kind of differences when changing the number of processes or the problem size. Observing an upward trend would be concerning.

The time it takes to compute the similarity matrix is expectedly low—at most 209 milliseconds. As predicted, it seems to depend linearly on the number of processes and is additionally influenced by the size of the lattice. Particularly, comparing the measurements between 1,024 and 4,096 processes hints at this. The timing can likely be improved by devising a variant of van der Merwe's algorithm that does not require redundant attributes to be present, through parallelisation, or by fine-tuning the implementation e.g. by reconsidering the choice of containers for internal data storage.

4.2.3 ParaDiS

The trace files used contain data of up to 4,096 processes, are up to 13 gibibytes large, call between 318 and 325 distinct functions, and have 625–655 different function pairs. Dynamic load balancing has been disabled in ParaDiS.

Figures 4.4 and 4.5 show the size of the resulting concept lattice for each run. There are up to 57,814 nodes and 216 groups in a lattice. The group count is tolerable, whereas the number of nodes is not. In fact, it is so large that it takes hours to compute the similarity matrix. Almost all of these nodes neither contain processes nor function pairs. They are empty. Nodes like this are necessary for a concept lattice to have for each pair of nodes a unique lowest upper bound and a unique greatest lower bound (see definitions in Section 3.3). For extracting similarities this property is not strictly necessary. The described approach traverses the nodes of a concept lattice without using suprema and infima. Because of this, a new step is added to the techniques. For each empty node it removes this node and forwards each incoming edge onto all outgoing edges. The step is invoked after removing redundant information and before computing the similarity matrix. Implementing this optimisation lowers the node count and, therefore, enables faster similarity matrix computation. Figure 4.6 depicts the number of non-empty nodes for each run. It looks very similar to that of the group counts.

As previously stated, van der Merwe's algorithm requires the concept lattice to retain duplicate attributes. This means that during construction there are no empty nodes which could be removed. And even if, the algorithm needs an intact concept lattice to work. Therefore, empty nodes cannot be removed before finalising the lattice. This in turn means

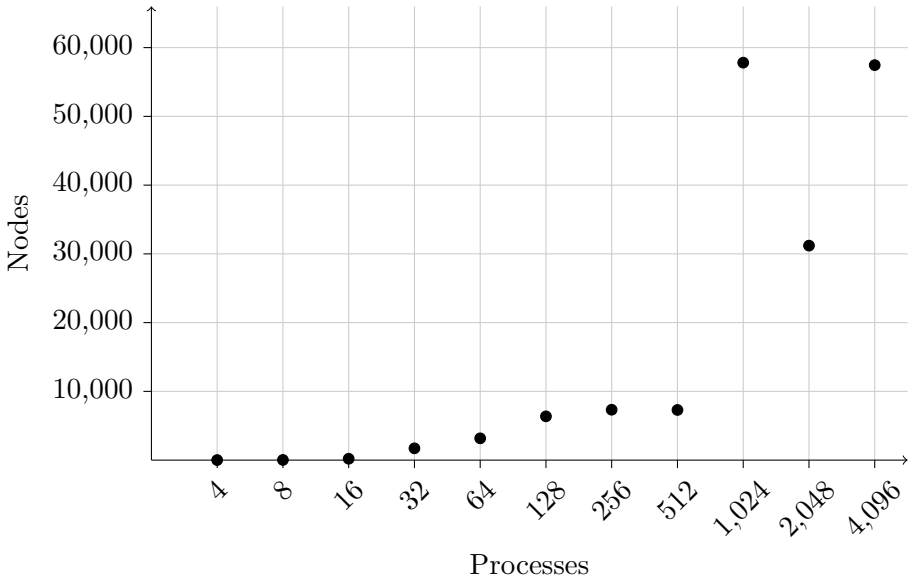

Figure 4.4: Number of nodes in the concept lattices for ParaDiS runs

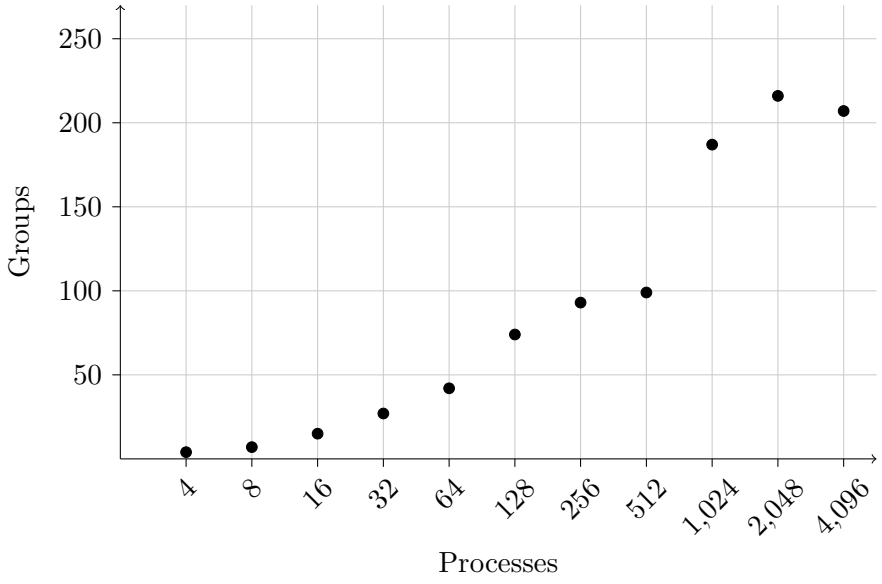

Figure 4.5: Number of groups for ParaDiS runs

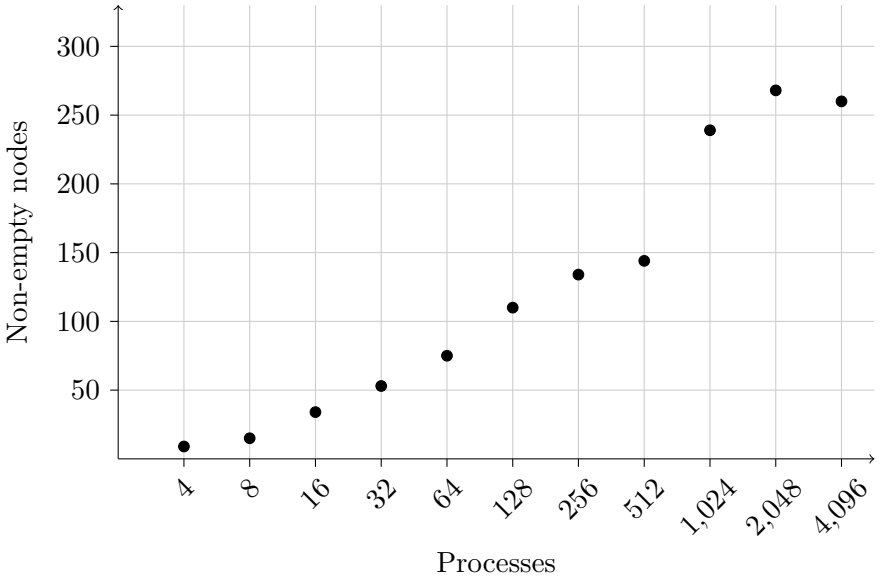

Figure 4.6: Number of non-empty nodes in the concept lattices for ParaDiS runs

that, unlike for similarity matrix computation, the number of empty nodes still influences the timing of lattice construction.

Figure 4.7 depicts the timing of calculating the similarity matrix. It uses two different time scales to improve readability of the results below 1,024 processes. Notice that the timing has been split into three phases. The first one iteratively constructs the lattice. The second one removes redundant information from it and deletes empty nodes. The last one computes the similarity matrix from the final lattice. Mind that the timing of subsequent phases is stacked in the chart.

The time it takes to insert all processes into the lattice mainly depends on the number of processes and the number of nodes in the lattice. It takes at most 30 seconds for 1,024 processes. Finalising the lattice, which takes a maximum of 20.5 seconds for 1,024 processes, is influenced by the number of nodes. Therefore, the timing of these two phases strongly resembles the node count chart (Figure 4.4). Computing the similarity matrix, which takes up to 9.9 seconds, depends on the number of non-empty nodes and the group count. Therefore, calculating the similarity matrix does not take less time for 2,048 processes than for 1,024 and 4,096 as is the case in the previous two phases.

Overall, the number of nodes and groups is much larger than hoped for. From these measurements, one cannot clearly determine a trend. It is possible, that the number of nodes and groups increase with rising process counts. Assuming the techniques are used for grouping similar processes in preparation for further performance analysis, the timing is unacceptable.

Detecting whether the lattice grows too large is simple. An implementation can set an upper limit for the number of nodes and monitor adherence to it. The more interesting question is how to alleviate the problem. One straightforward way is to use functions instead of function pairs for the lattice. The rest of the techniques is applied without

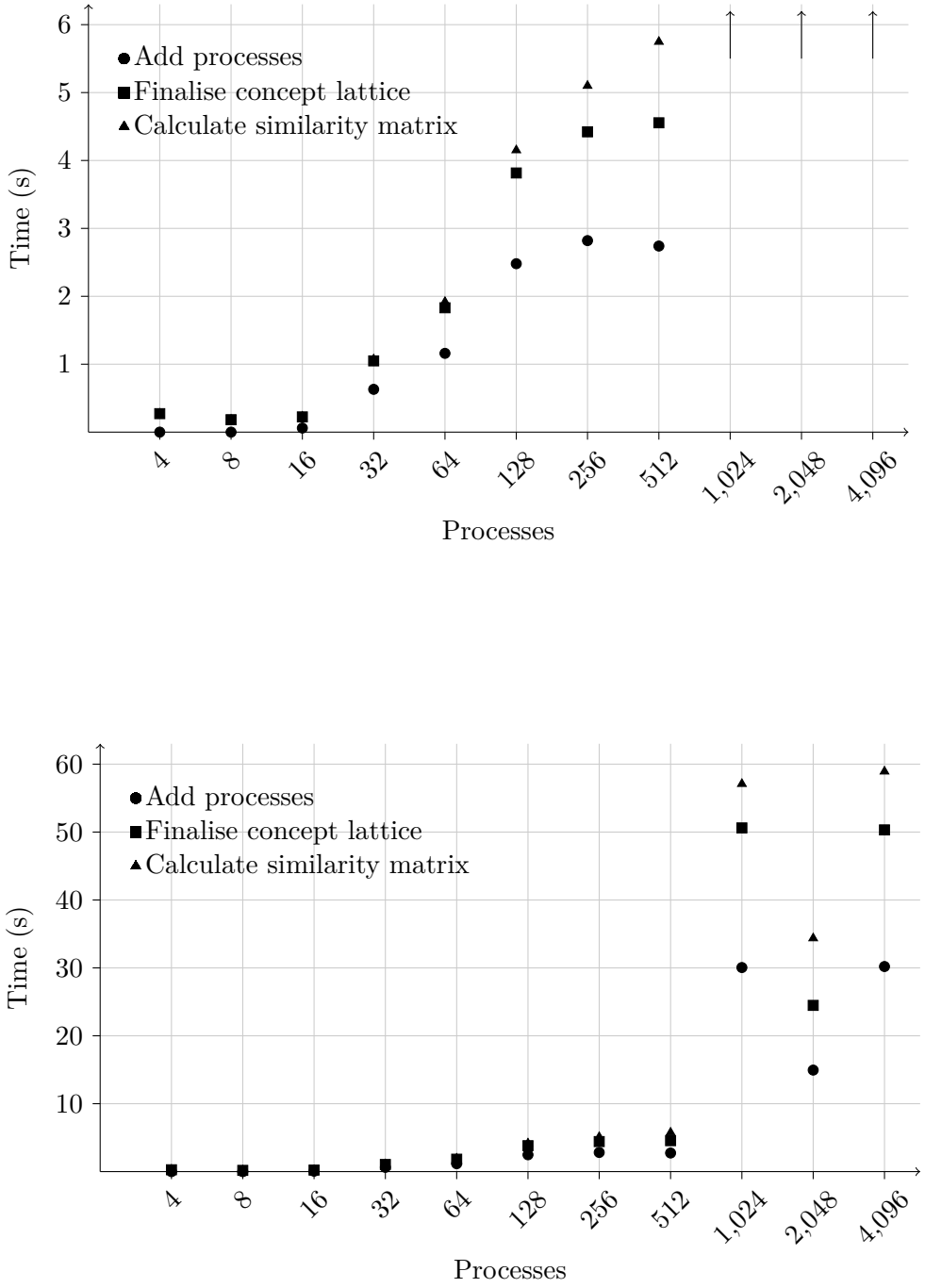

Figure 4.7: Stacked plot of the different concept lattice construction and evaluation phases for ParaDiS runs

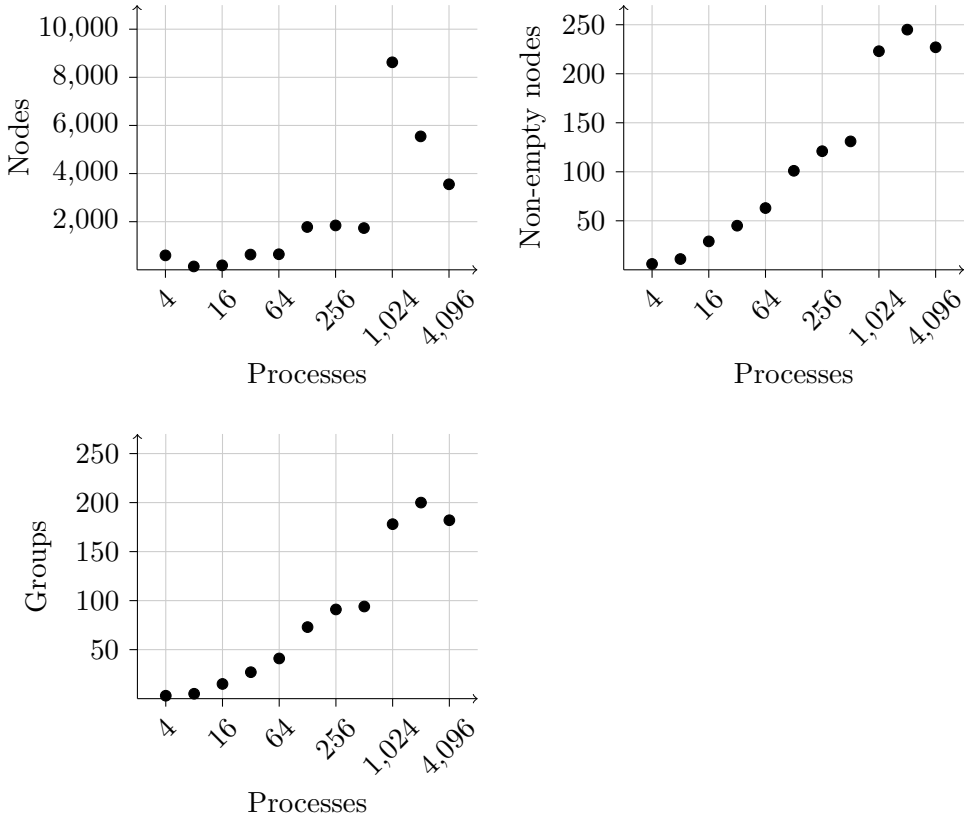

Figure 4.8: Number of nodes, non-empty nodes and groups in the concept lattices for ParaDiS runs using functions instead of function pairs

modification. This yields a coarsening of the grouping. This means, processes that are equal according to the function pair similarity measure are equal under function similarity measure, as well. Although being unlikely, the similarity of two unequal processes can decrease when switching to functions.

Figures 4.8 and 4.9 depict the lattice size and timing when using functions instead of function pairs. The trends are, as expected, similar to the function pair variant. While the number of nodes decreases to roughly 15% of the function pair lattice, the number of non-empty nodes and groups is only slightly less. The maximum number of nodes is 8,624, the largest number of non-empty nodes is 245 for 2,048 processes and the maximum group count is 200 for 2,048 processes. As expected from these numbers, adding processes and finalising the concept lattice is significantly faster. Adding processes takes up to 2.1 seconds. Finalisation takes at most 1.7 seconds. Since the number of non-empty nodes and groups is only a little less, computing the similarity matrix improves just slightly. A maximum of 7.9 seconds is measured.

While the node count is much lower than in the function pair variant, the number of non-empty nodes and groups improved only slightly. The upside is that the resulting groups are almost the same as in the function pair case. In an implementation, it is advisable to construct both, the function and function pair lattice, at the same time and pick one (or neither) upon hitting a set limit of nodes or groups. Using this variant, the

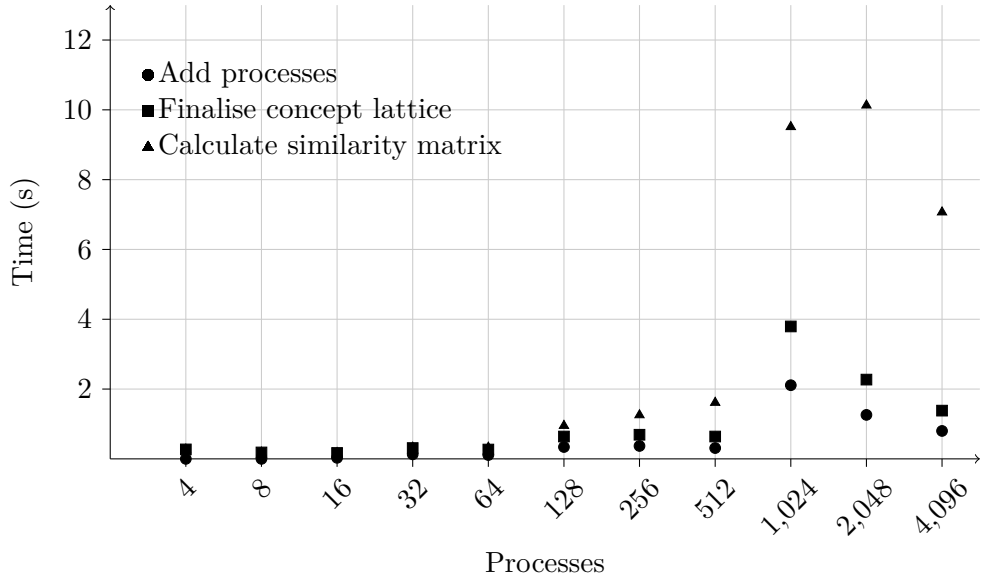

Figure 4.9: Stacked plot of the different concept lattice construction and evaluation phases for ParaDiS runs using functions instead of function pairs

time required to compute the similarity matrix is lowered from 58.9 to 10.1 seconds for the worst-case. If a performance analysis tool is only interested in the grouping and not the specific similarity values, the needed time is lowered from 50.6 down to 3.8 seconds.

There are two other ways to improve scalability. The first one is to filter functions on demand. While iteratively constructing the concept lattice, the implementation can monitor the number of nodes. When the insertion of a particular process causes the node count to increase steeply, the implementation can repeat the insertion excluding attributes causing the blow-up. Alternatively, it redoes the whole construction omitting them completely. The difference between the two is that the former one is faster but some attributes are not added to all processes that should have them, which might be undesirable. By removing functions or function pairs, information about program structure is lost. Therefore, one has to plan for this loss of accuracy and/or retain important information.

The second approach is to use hashing. It is viable only if the process grouping and no specific similarity values are needed. For each process the canonical form of the function pair set is hashed and inserted into a hash table. The set itself can be discarded. This yields the same equalities as before. Hash collisions introduce false positives. The likelihood of negative effect can be decreased by using a strong hash function. Compared to the techniques presented in this thesis the hashing approach can only detect equality but is extremely fast, requires less memory and can be parallelised easily.

The next section explains why lattice-based similarity fails for ParaDiS. It generalises the findings and discusses implications.

4.2.4 Irregular Function Call Patterns

Programs exhibit regular function call patterns [2, 6, 41]. This is especially true for the dominant style of parallel programming *single program multiple data* [73]. Which functions are called on which process and in which order is largely independent of the process count and problem size. However, this is not entirely true. Two classes of applications have been identified that have substantial differences in the code paths taken between different processes. ParaDiS fits into one of them.

A *scheduler* is a mechanism distributing tasks to processes, e.g. to optimise resource usage. Different functions encapsulate different tasks. The process on which a function then runs is practically random from the perspective of the performance analysis tool. This is, for example, the case for CUDA streams. A graphics card is assigned one or more streams. A scheduler dynamically assigns work packages to streams at runtime. Generally, two runs of the same program do not produce the same function or function pair sets for the streams. For the same reason, comparing two streams inside one run yields random differences. Consequently, the results of the techniques are unpredictable, if the analysed application implements scheduling.

Imagine an application using domain decomposition, where every process simulates one or more cells of a three-dimensional grid. On this grid a number of objects, for examples planets, stars, lasers, spaceships and explosions are placed. Simulating a time step for an object invokes object-specific functions. Objects interact with each other and travel between cells as the simulation progresses. The functions a process calls, then, depend on the spatial position of each object. The described scenario can also use dynamic load balancing and, therefore, reassign objects between processes at runtime. Comparing two runs or two processes inside one run can yield differences that the techniques of this thesis cannot process effectively. ParaDiS is such an application. The execution of code paths on a process depends on the domain decomposition and the location and state of each object.

In both cases, the effects also depend on the total execution time of the program. The longer an application runs, e.g. the more time steps it simulates, the more similar the function and function pair sets can get. The opposite is less likely but possible.

To conclude this section, it can be stated that the introduced similarity methods are unsuited for the described application types. Fortunately, these programs represent a minority and there exists a number of remedies. For example, one can filter functions, or use hashing as described at the end of the previous section. Another possibility is to combine irregular processes into one and apply the techniques to the combined process and the regularly behaving ones. Depending on the situation this approach might be acceptable.

The next section evaluates the suitability of the subsumption measure to detect a special class of differences using two applications.

4.3 Subsumption Measure

Section 3.5 introduced a variant of the similarity measure that is robust to fluctuations in function call patterns due to the omission of calls. If two processes are dissimilar according to function pair similarity, it might still be the case that they are similar, but some functions are missing or some invocations have been inlined. First, this section demonstrates the effectiveness of the subsumption measure for detecting threads. Second, the measure is used to identify granularity differences between traces of different processes.

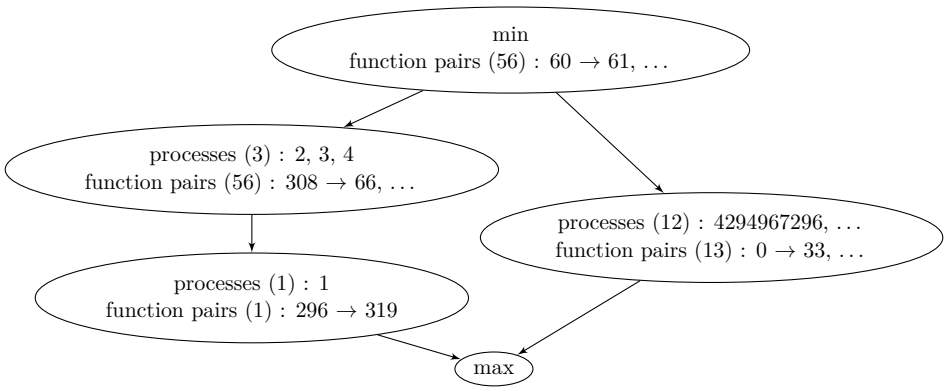

Figure 4.10: Concept lattice of function pairs for a BT fluid dynamics simulation run utilising four processes and twelve threads

In Section 3.5 the capability of the approach for detecting spawned threads has already been demonstrated for the WRF weather simulation. A second instance, using the pseudo-application BT, is presented here. As mentioned in Section 4.1, BT imitates the computation of a typical fluid dynamics simulation and solves systems of partial differential equations for this purpose. It uses MPI and OpenMP for parallelisation. Figure 4.10 shows the regular concept lattice for the BT run, which executes on four processes and twelve threads. To recap, each node contains a set of processes and function pairs. The size of the set is in parenthesis and a few example entries follow. Each node inherits the function pair sets from nodes reachable along upward-edges. In this run, processes have the identifiers 1 to 4, whereas threads are 4,294,967,296 ($= 2^{32}$) and above. All processes and threads share 56 function pairs. Process 1 has one additional pair compared to processes 2–4 and, thus, subsumes them even before taking the transitive closure of their function sets. Threads differ from processes. Table 4.2 depicts the resulting similarity

Table 4.2: Similarity matrix for a BT fluid dynamics simulation run utilising four processes and twelve threads

	Process 1	Process 2, 3, 4	Threads
Process 1	1	0.99	0.44
Process 2, 3, 4	0.99	1	0.45
Threads	0.44	0.45	1

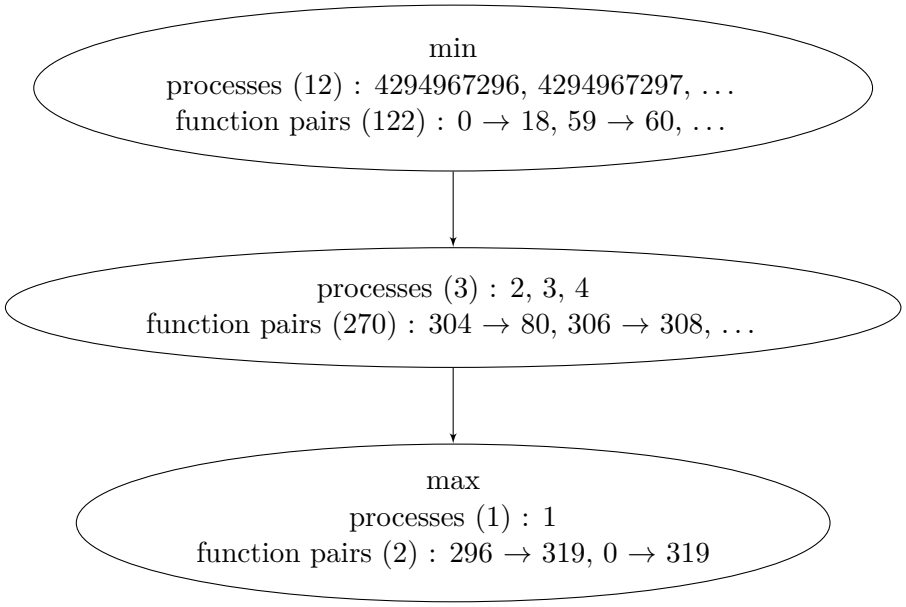

Figure 4.11: Concept lattice of the transitively closed function pair sets for a BT fluid dynamics simulation run utilising four processes and twelve threads

values. Remember, similarity is symmetric and returns 100% when comparing something to itself.

Transitively closing all function pair sets and constructing the lattice anew yields the concept lattice (Figure 4.11) from which the subsumption measure for each pair of processes/threads is subsequently computed. Now, the threads are indeed identified as having a subset of the processes' function pairs, since their node is reachable along upward-edges starting at the processes. Table 4.3 shows the resulting subsumption measure matrix. The cells represent the likelihood of the process/thread in the row doing at least as much work as the one in the corresponding column. The main diagonal is always 1. Process 1 has all function pairs, possibly using intermediate steps, of every other process and thread (100%). One can therefore conjecture that process 1 does at least as much work as the other processes and threads. Processes 2–4 subsume the threads (100%). The opposite is unlikely since the subsumption measure is 31%. Process 1 is more likely to subsume processes 2–4 than the opposite being the case (99% $<$ 100%), but they are very similar. In fact, process 1 calls the extra function `print_results`.

Mind, that the call trees of the threads are not sub-trees of the processes'. Therefore,

Table 4.3: Subsumption measure matrix for a BT fluid dynamics simulation run utilising four processes and twelve threads

	Process 1	Process 2, 3, 4	Threads
Process 1	1	1	1
Process 2, 3, 4	0.99	1	1
Threads	0.31	0.31	1

Table 4.4: Function-pair-based subsumption measure matrix for the four master threads of a GROMACS simulation run

	MT:0	MT:1	MT:2	MT:3
Master thread:0	1	1	0.63	0.5
Master thread:1	0.73	1	0.63	0.5
Master thread:2	0.05	0.07	1	1
Master thread:3	0.01	0.01	0.24	1

Table 4.5: Function-based subsumption measure matrix for the four master threads of a GROMACS simulation run

	MT:0	MT:1	MT:2	MT:3
Master thread:0	1	1	0.99	0.98
Master thread:1	0.76	1	0.99	0.98
Master thread:2	0.24	0.31	1	1
Master thread:3	0.08	0.10	0.33	1

straightforward call tree comparison does not yield the above results. The same holds when using function pair sets. Taking the transitive closure of each set is key.

Another use for this approach is to detect whether differences between processes across two runs, or equally inside one run, are due to inlining or different levels of recording granularity. For example, rarely are the full execution details needed for every process. One representative might be enough. To evaluate the effectiveness of the subsumption techniques, the molecular dynamics simulator GROMACS is used. The program run executes on four processes, each spawning five threads and three CUDA streams. Each process has its own executable with different built-in instrumentation. From process 1 to 4, the recording detail decreases. The first process records the most, the second one a little less. The third one only records OpenMP, MPI and CUDA calls. Process 4 observes only MPI and CUDA. To provide more context, Figure 4.12 shows an excerpt of the call tree and function call stack over time for each process, called *master thread* from here on.

Creating the lattice and computing the subsumption measure matrix (Table 4.4) does not yield what one would expect. Master thread 0 is identified to subsume master thread 1, but master thread 2 and 3 still show substantial differences (63% and 50%). Due to an implementation detail in the trace recording software Score-P [5], a new artificial main function called `PARALLEL` is introduced when only recording OpenMP, MPI and CUDA events or less. This function lies at the lowest call stack level and calls each recorded function. Therefore, master thread 2 and 3 are not correctly identified as subsumed by master thread 0 and 1. This problem can easily be fixed via string matching. The preferred way would be to remove this artificial function from Score-P.

Calculating the subsumption measure for the functions instead of function pairs yields the expected results (Table 4.5). The four upper-right cells are not 100% due to this one extra function.

Figure 4.12: Call tree excerpts and Vampir Process Timelines of the four master threads with decreasing detail from left to right / top to bottom. Application-specific functions are depicted in green. OpenMP regions and functions are brown. MPI functions are red. CUDA functions turquoise.

As demonstrated using three real-world applications, the subsumption approach is an effective tool for detecting spawned threads, different levels of recording granularity and inlining. This information is valuable for further comparative performance analysis. The next section evaluates how the introduced similarity techniques can aid performance visualisation.

4.4 Visualisation

The function and function pair similarity measures are intended to be a low-overhead, early filtering mechanism that automatically clusters processes according to their structure. As an auxiliary tool, the subsumption measure finds out what causes differences discovered by the similarity measure. It answers questions like: "Are processes 2 to 4 different from process 1, because process 1 additionally prints debug output and writes to log files?", or "Are threads 1–4 spawned from process 1?". The concept lattice is an efficient way of storing the information needed to determine the similarity or subsumption measure for each pair of processes.

Visualising the lattice itself works for cases with a sufficiently small number of nodes. Annotating the concept lattice with information about applied parallel programming paradigms and APIs yields a useful tool for program structure inspection. Using the WRF weather simulation, Figure 4.13 shows what this new tool could look like. Unfortunately not all application runs yield low lattice node counts, as is demonstrated in the previous sections. No mechanism for reducing the number of nodes to arbitrarily small levels whilst preserving important information exists, yet. Therefore, visualising the lattice in an automated way is not recommended. Instead, the introduced techniques should

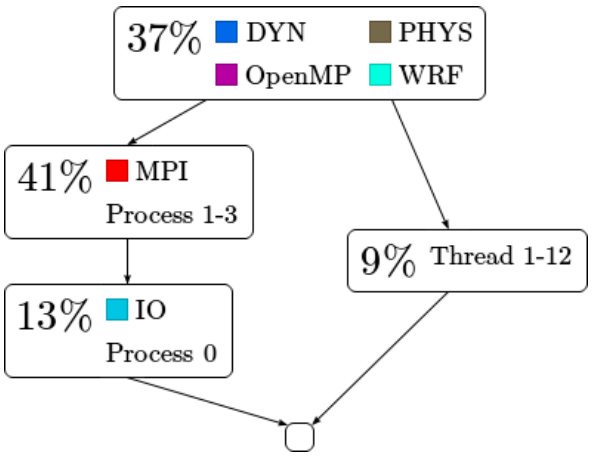

Figure 4.13: User-friendly version of Figure 3.12 on page 21, based on the concept lattice of function pairs for a WRF weather simulation run, annotated with function groups and condensed process names. The run utilises four processes and twelve threads. Percentages depict the amount of new function pairs a node introduces.

be incorporated into existing visualisations or used to develop new ones on top of it. The following two sections demonstrate the applicability of the methods to aid visualisation. First, they are applied to improve an existing Vampir display. Second, a novel profile view is proposed which leverages the introduced clustering.

4.4.1 Improving Vampir's Process Summary

Vampir's Process Summary display is first introduced in Section 2.3. Figure 4.14 gives another example of its visualisation. Aside from the Process Summary at the bottom, the Master Timeline and Function Summary are depicted to provide some context. Subsequently, an overview of the display, which is described in greater detail by Brunst [8], is given. The Process Summary presents a clustered per-process Function Summary. The k-means algorithm [17, 52, 55, 74] determines the clustering. It generates a clustering of p d-dimensional points for a target number of clusters k in $\mathcal{O}(p^{dk+1} \log p)$ steps. The display uses a fixed number of dimensions, which are the ten functions having the overall largest accumulated exclusive time. Each process is a point in this ten-dimensional space with coordinates corresponding to the accumulated exclusive times of the respective functions in this process. The number of clusters is, by default, determined by the available vertical space in the chart. Users can also set this number manually. For each cluster the display shows (from left to right) the number of processes in this cluster, a simple graphical representation of the position of these processes and a stacked bar chart. The stacked bar chart has one bar for each of the ten functions. Each bar has a staircase representation marking the minimum, average and maximum accumulated exclusive time of all processes in this

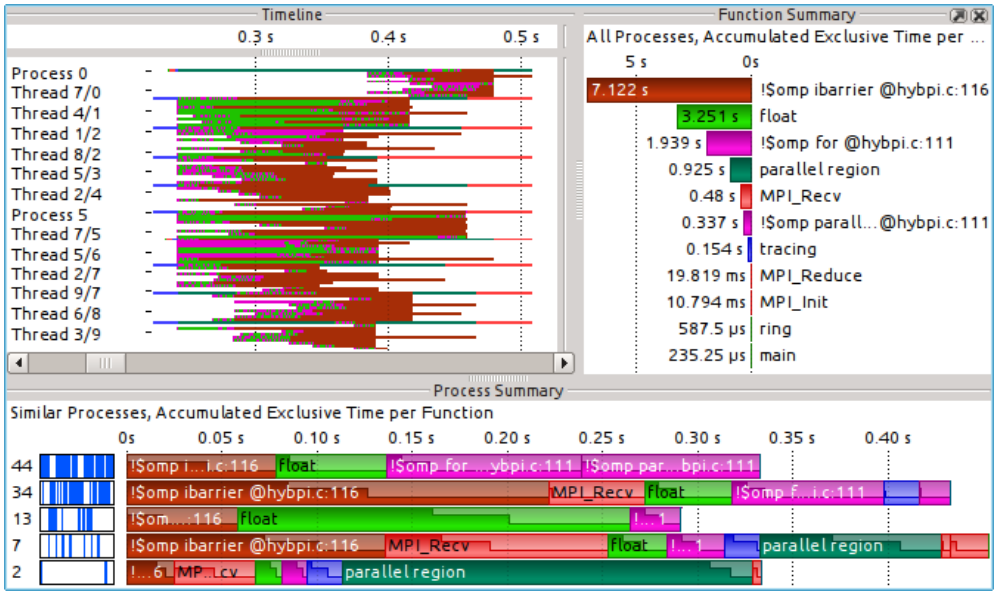

Figure 4.14: Vampir's Master Timeline, Function Summary and Process Summary for an example program computing decimal places of π using MPI and OpenMP. The run utilises ten processes and 90 threads.

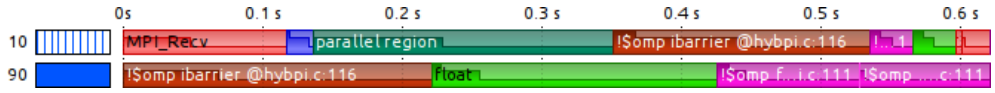

Figure 4.15: Vampir's Process Summary using the new methods to generate clusters for a π example program run

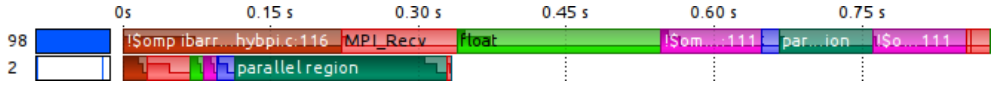

Figure 4.16: Vampir's Process Summary using the current clustering to create two clusters for a π example program run

cluster. The added length of all bars is of little interest, since it is the summed maxima from different processes. In conclusion, the Process Summary presents an overview of the time each process spends in the ten most time-consuming functions. It gives insight into a program's structure by scalably clustering processes according to their profile. The clustering minimises variation of the shown numbers in each cluster.

The effects of replacing the current clustering approach with the new techniques is illustrated, subsequently. First, an application calculating the decimal places of π, utilising ten processes and 90 threads, is used. It employs MPI and OpenMP to run in parallel. Figure 4.15 shows the result of applying function pair similarity grouping to the Process Summary. All ten processes are in one cluster, whereas the threads form the second cluster. Using fixed two clusters, the current approach yields the clustering depicted in Figure 4.16. Interestingly, it mixes threads and processes, even though they do not call the same functions. Mind that `MPI_Recv` and `tracing` are absent on the threads. This occurs, because the variations in execution time of the OpenMP barrier and parallel region are larger than the difference between non-existent (zero) and actual `MPI_Recv` and `tracing` timings. If one process in a cluster does not call a function, the corresponding bar's minimum defaults to zero. The current approach, thus, hides some of the variation in OpenMP barrier and parallel region timings of cooperating and functionally equal processes/threads. Using the new methods, on the other hand, reveals more relevant variations, gives better minima and overall yields a more meaningful clustering.

Figure 4.17 shows the clustering for a large FD4 trace using the current approach. For such a highly parallel, homogeneous application the resulting clusters seem to be random. Applying the new methods (Figure 4.18) reduces the number of clusters to two. One cluster contains the first and last process. The other processes form the second cluster. In this case, it is not obvious what the more meaningful approach is. Displaying more or fewer clusters does not reveal any more information. Showing a limited number of clusters might be the preferred choice, since the minimum, average and maximum taken over all similar processes reveals variations between all of them, instead of the variation of a seemingly random subset of processes. A second argument for limiting the number of clusters is that the user might be confused by looking at the many similar bar charts.

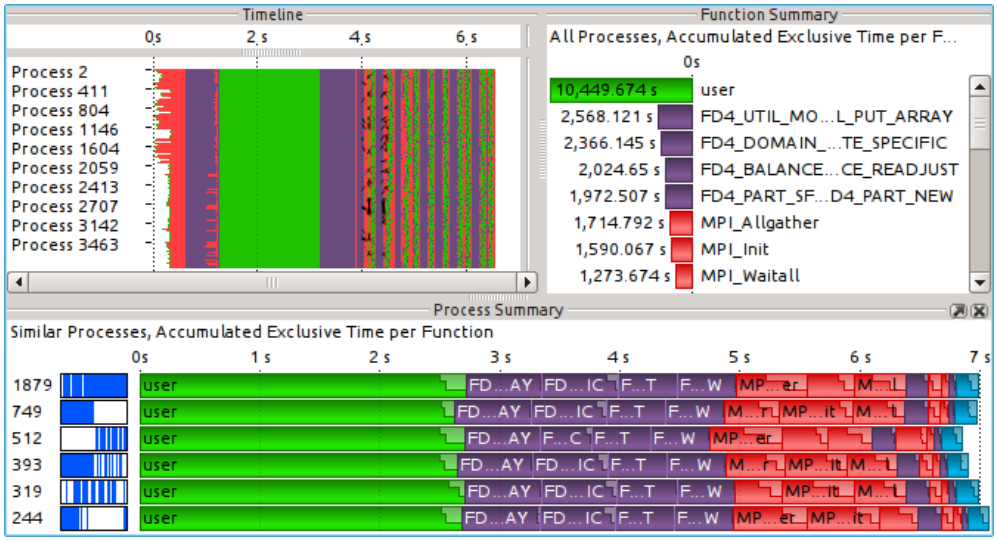

Figure 4.17: Vampir's Master Timeline, Function Summary and Process Summary for an FD4 run utilising 4,096 processes.

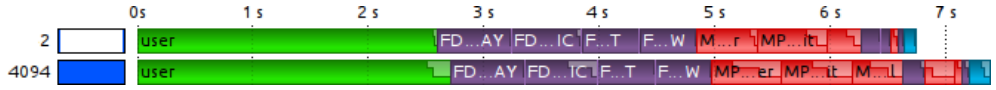

Figure 4.18: Vampir's Process Summary using the new methods to generate clusters for an FD4 trace

Most eye-catching is the apparent difference in total stacked bar length, which gives no insight.

The current approach aims to minimise time variation inside each cluster. This can yield unnatural clusters as demonstrated for the π-example. The new method, instead, groups processes according to their function call behaviour.

Currently, processes are clustered considering the ten most time-consuming functions. This means it disregards variation in other functions, possibly overlooking important differences. The new methods have the advantage of incorporating every function's structure.

The current method determines the number of clusters disregarding information about the analysed program. The new approach, on the other hand, yields a natural maximum cluster count. This number can be reduced by using a fixed or user-specified similarity threshold. An implementation should use a default threshold, because many traces contain small differences which cause processes to be split into multiple groups. For example in Figure 4.18 the difference between the two clusters is 3%. They practically do the same work. For programs with sufficiently regular function call patterns, the new methods yield an intuitive clustering. As shown in Section 4.2.3 and 4.2.4, they are not suited for all types of applications.

The current approach employs k-means, of which the execution time is bounded by $\mathcal{O}(p^{dk+1} \log p)$ for p processes, d dimensions and k clusters. The number of dimensions is fixed. The number of clusters is limited by the vertical screen space. The process count is

not limited. Vampir's implementation is well optimised and generally fast. A worst-case clustering time of about seven seconds has been measured. The new approach is expected to be quicker in most scenarios, i.e. bounded by $\mathcal{O}(p)$, especially for large process counts. As described in Section 4.2.3, there exist applications for which calculating the clustering takes very long—longer than the current approach. Therefore, in an actual implementation one has to take care to remain responsive under all circumstances, which is an achievable engineering goal.

After demonstrating the effectiveness of the introduced techniques to improve an existing visualisation, the next section offers an innovative take on displaying profiles. Aside from incorporating the new clustering, further novel ideas are presented.

4.4.2 A Novel Profile View

Profilers play a central role in software performance optimisation. This section first gives an overview on current parallel profilers and motivates improvements. Next, a novel profile view is proposed, which leverages the developed clustering, adds novel features and improves usability over existing solutions.

Current Profilers Surveying today's profiling tools, this section focusses on their features and usability. Since parallel programming is becoming increasingly important, and the techniques introduced in this thesis aim to improve analysing parallel applications, concurrency-specific features are of special interest.

Although multithreading is used in many programs today, GNU gprof [29] does not account for it. It only observes the main thread and ignores all others [28]. The profiler included in the Google Performance Tools [27] supports multithreading by accumulating measurements of all threads. It has no parallelism-specific features. Neither profiler supports analysing multi-process applications. Two well-known profilers that explicitly enable inspecting multithreading and multi-process programs are Allinea MAP [3] and HPCToolkit [1]. Vampir [47], as a trace visualiser for massively parallel applications, offers profile displays as well.

Allinea MAP's main focus is usability. Figure 4.19 shows its user interface after profiling one of the tools created for this thesis. In addition to function timing, MAP records and displays various performance metrics like the number of floating-point operations and memory usage. Users are able to profile a program run and view results with little effort even on high performance computers. This is thanks to MAP integrating with MPI and job schedulers. It offers several sensibly constructed default views so users can quickly grasp the important characteristics of their program, and start investigating performance issues. In addition to providing accumulated values, it also displays metrics in a timeline-esque fashion. Values are always aggregated over all processes. It, thus, provides no means of displaying timing or performance counters for individual processes. MAP claims to scale well with increasing process counts and growing program execution time.

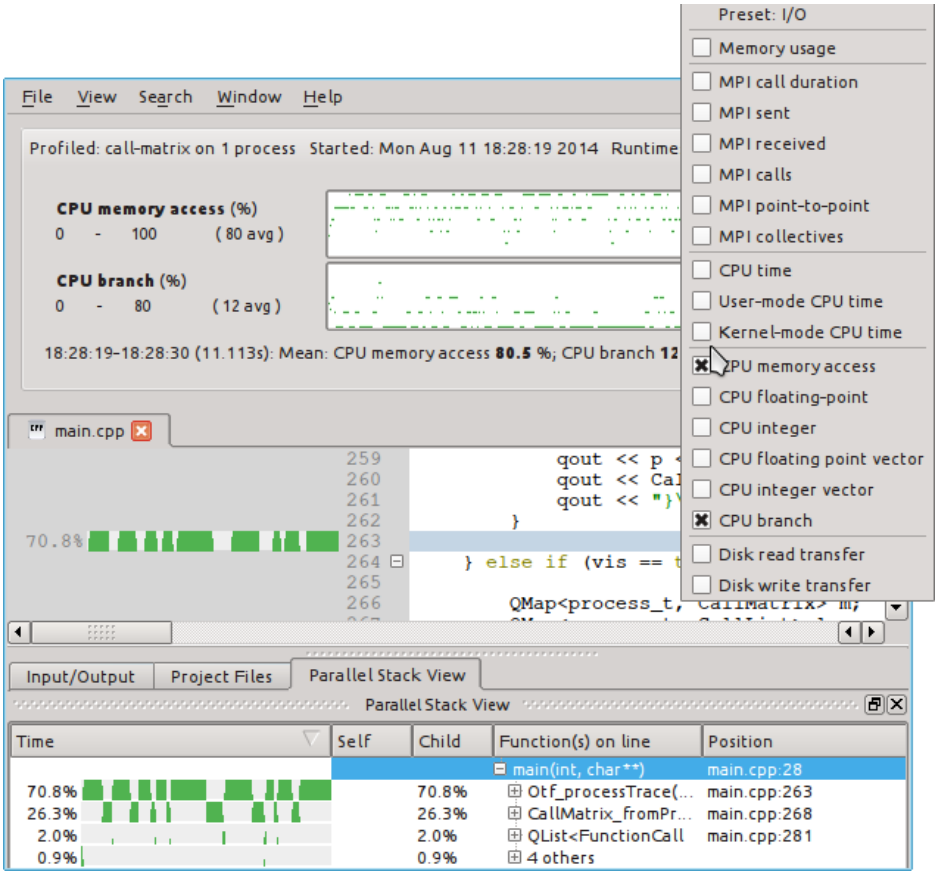

Figure 4.19: Allinea MAP presenting the results of profiling an application based on C++ [75] and Qt [65]. The top part shows performance metrics over time. Using the menu on the right side, users can choose from a great variety of metrics. The middle part displays the source code enriched with profile information. The bottom table shows the traditional profile plus an illustration of how much time is spent in each function over time.

HPCToolkit consists of a few command-line tools and a viewer program. Their use is reasonably simple. Figure 4.20 shows the user interface component displaying the profile of a simple MPI example program running on eight processes. One notable feature of HPCToolkit is that one can view the timing of any function on a per-process basis. This can be seen in the middle part of the screenshot. Furthermore, it supports recording performance counters and allows the user to combine and customise these to create new metrics. HPCToolkit includes a feature called *hot path analysis*, which unrolls the call tree to the point where functions spend less than 50% of the whole program runtime. This way users are navigated to likely points of interest. Generally, HPCToolkit's focus is on customisability and features, less on usability. For example users have a hard time reading timing information in scientific notation.

Vampir contains three profile-esque displays—the Function Summary, Process Summary and Call Tree. They all accumulate values and then present a condensed view. Figure 4.21 shows them displaying the results of a GROMACS run. The Function Summary presents

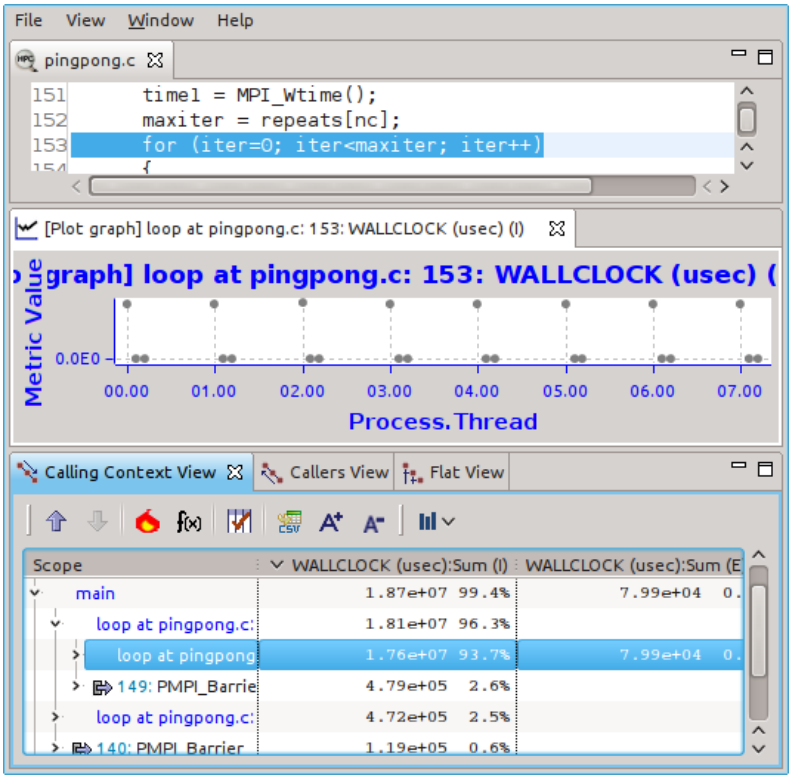

Figure 4.20: HPCToolkit showing the results of profiling an MPI ping-pong example. The source code view is at the top. The middle part shows how much time has been spent in a particular loop for all eight processes. The bottom part presents the profile with different information arrangements for each tab.

a traditional profile. The Process Summary, which has been the subject of the previous section, clusters per-process profiles to expose differences between them. The Call Tree unravels the flat profile, so statistics of functions called via different paths are distinguished. These three displays somewhat overlap in purpose and functionality. Call path profiling [32] is not supported. Statistics of calls to the same function in a given function but from different lines of code are aggregated into one value. This prevents exact attribution of profiling results to source code locations. Both MAP and HPCToolkit support call path profiling. For each Vampir display numerous filters are available. For example users can choose which functions and processes they want to view statistics about. Overall, Vampir has a highly flexible, powerful and yet simple to use user interface.

Having investigated the capabilities of these tools, there is arguably potential in developing a novel profile viewer. Parallel programming support and usability are the focus areas where it improves upon existing solutions. It should enable users to compare performance data for multiple runs and between processes of the same run. Instead of creating a standalone profiler, one could also replace Vampir's three profile displays with one holistically designed one. How to achieve these goals is the topic of the next section.

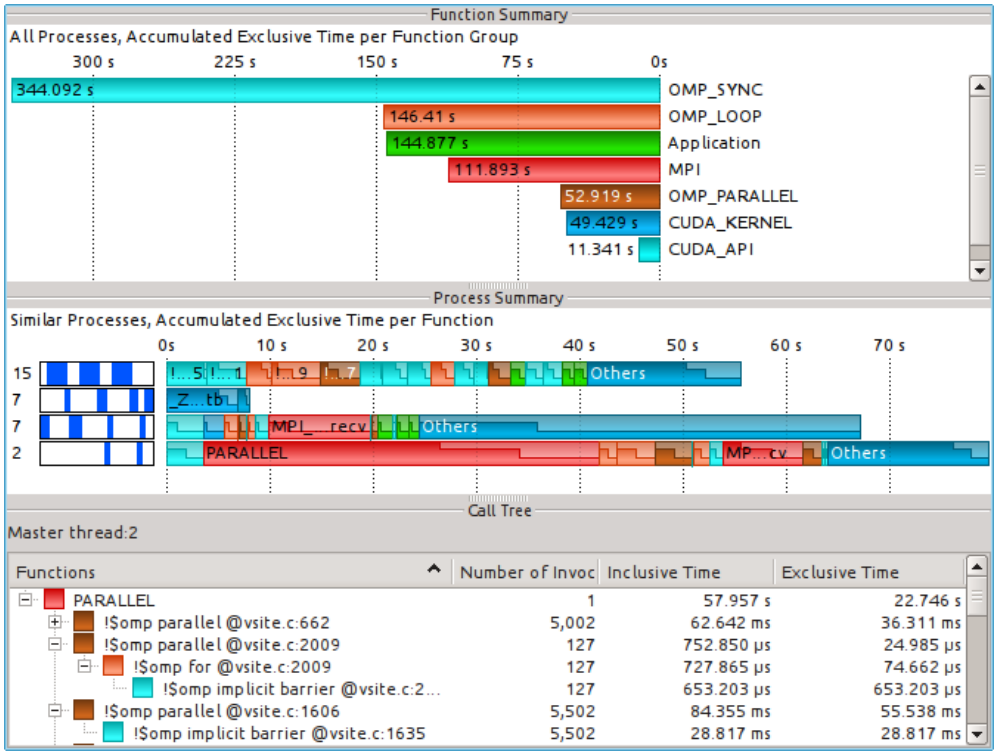

Figure 4.21: Vampir's three profile displays. At the top is the flat profile. The middle shows the Process Summary. Below is a call-tree-based profile.

Proposal The following proposition reflects, in part, my personal opinion of how a profiler for parallel programming should be designed. This opinion is based on eight years of work experience in the field of parallel performance analysis and visualisation.

Structurally similar processes more likely interact with each other than dissimilar ones. Two prime examples are programs using the fork-join concurrency model, and applications that compute in iterations between which they share information via scatter and gather operations. Therefore, the proposed profile view only presents information for similar processes. Not doing so might lead to unintuitive results, since functions used differently on some processes still influence measures, like sums and averages, computed over all processes. Averages, deviations and minima get distorted by zero values from functions that are not called on some processes. Functions wasting time on many processes overshadow others that spend as much time but on fewer processes. Initially, the profile view presents the user all process groups and similarity information, obtained via the introduced techniques, and lets him decide which processes to consider for analysis.

To display variation between the chosen processes, the proposed profile viewer uses box plots [56] instead of single values. Box plots (Figure 4.22) depict the second, 25th, 50th, 75th and 98th percentile, plus optionally outliers, of a given distribution. The pth *percentile* is the smallest value such that p percent of all data points are less than or equal to that value. The 50th percentile is also called the *median*. *Quartiles* are multiples of the 25th percentile. Inside the box of a box plot, half of all data points are

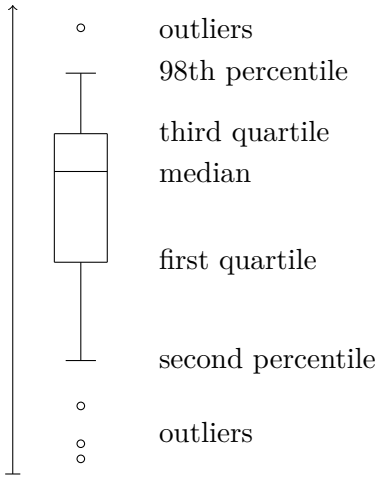

Figure 4.22: Box plot

located. 96% of all values lie between the whiskers. For a given function each process adds one data point to the distribution from which the box plot is derived. Possible values for display are the accumulated exclusive time, average exclusive time, accumulated inclusive time, invocation count, performance counters and many others. Accumulation and averaging is done over individual function calls. Each function, function group or call stack configuration, depending on what is desired, is depicted as one box plot. Imagine a traditional profile where bars are replaced with box plots to illustrate variation between processes. Via mouse hovering, users can find out which processes lie in specific parts of the box plot. Figure 4.23 shows what this could look like. Most performance analysis tools rely on minimum, average, maximum and some form of deviation for display. These values strongly depend on outliers and may, thus, mislead. Percentiles are robust to outliers, more intuitive and lead to a better understanding of a distribution. Conclusions like "50% of all processes spend one to two seconds in `MPI_Recv`." are impossible to draw from accumulated/selected values like minimum, average, maximum and deviation. Another drawback is that deviation is indifferent to whether values lie above or below average. Using percentiles, tendencies in variation can be displayed more clearly. The downside is that obtaining percentiles is more costly in terms of computation and space.

To ease access to the wealth of information contained in profiles, few well-defined starting points should be provided. Not all scenarios where inclusive time, exclusive time and invocation count can be displayed make sense intuitively. A call tree should, first and foremost, provide inclusive timing. One can then repeatedly unfold functions where much inclusive time is spent, to locate points of interest. If desired, the user can view additional information like exclusive time, invocation count and other performance metrics. To find out where the most time in a program run is spent, one should first look at the accumulated exclusive time displayed per function group. These groups can then be unfolded, e.g. by double-clicking, to inspect in which functions exactly the time is spent. To find candidates for inlining, it makes sense to display a flat list of functions with their respective invocation

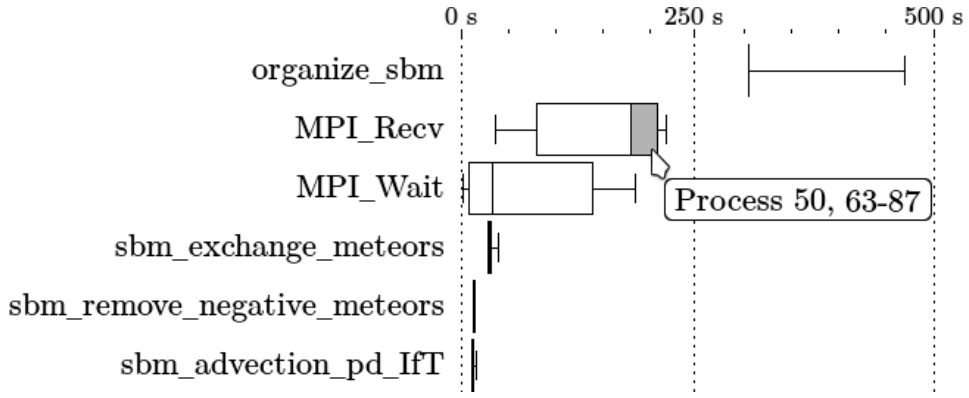

Figure 4.23: Excerpt of a box plot based-profile for a COSMO-SPECS run on 100 processes. The tooltip shows that process 50 and 63 to 87 are between the median and third quartile.

count, perhaps in conjunction with average and/or accumulated exclusive time. Other combinations of exclusive time, inclusive time, invocation count over functions, function groups or the call tree should only be supplied if the user desires.

Furthermore, it should be easy to find out which functions call a specific function, and which functions are called by it. This enables users to quickly investigate where inclusive time is spent further down the stack or unwind a function into multiple entries to inspect timing and other metrics distinguished by calling location.

Emphasis should be put on avoiding to provide confusing or useless information. Therefore, sensible sorting order defaults should be in place. Additional orders help users obtain interesting information more quickly. In general, the larger the accumulated exclusive time of a function, the more important it is. Repeatedly calling a function using the same arguments, modulo external influences and global state, mostly yields the same execution time. Therefore, large variations in accumulated exclusive time of a function between processes, especially when called via the same call path, hints at load imbalance. A sorting order based on both accumulated exclusive time and its variation should, thus, be provided. Another example for a helpful auxiliary order is sorting functions by invocation count paired with accumulated exclusive time. This helps users find candidates for inlining.

In addition to these central features, there are lots of smaller ones that are to be realised. Mind that smaller does not imply easier or quicker to implement. Much of what follows is part of that extra mile which makes software good.

- Users should be able to quickly search and filter for functions, processes and performance metrics everywhere. Searching for specific information in a given view can be tedious. For example, a simple search dialog (Ctrl+F) which highlights the searched word already helps.
- Proper undo and redo support is imperative. Neither HPCToolkit nor MAP have it. Users want to quickly go back and forth during an investigation and not having to remember the steps they take in order to do that.

- The user interface should be persistent. It ought to memorise the choices a user makes while navigating through different views, selecting functions, changing sort orders, moving up and down the stack. A view should be restored to its last state when being opened again. This includes storing an analysis session and restoring it once the same program is profiled again, or a previous profile is viewed. In general, the user interface should never present something unexpected or unintuitive.
- The user should be able to choose from viewing time in CPU cycles, seconds and percentages.
- Functions should automatically be grouped. Programs have hundreds and thousands of functions. Providing function groups improves overview and readability. VampirTrace [79] provides very few default function groups. The successor Score-P and its file format OTF 2 do not support the concept of function groups. One has to improvise function groups using auxiliary information about programming paradigms and parallelism-specific properties, as is done in Vampir. These groups have to be generated automatically. Libraries, source files, C++ classes are likely candidates. Membership can be identified via string matching.
- Functions should be easy to regroup. Automatically generating a grouping that is 100% correct and suits the users needs is hard if not impossible. Therefore, enabling easy and quick regrouping is key. One has to take extra care when functions are part of multiple groups, because it makes comparing values and applying percentages more difficult. A straightforward remedy is to only show mutually exclusive function groups.
- Cycle detection should be implemented. In recursive programs, straightforward calculation of inclusive timing, i.e. subtracting start from end time, leads to unintuitive results. For example, if A takes 5 seconds of which 3 are used in a sub-call to A, then A did not spend 8 seconds but 5. The inclusive time of a single function can, thus, exceed the execution time of the program. By using *cycle detection*, the inclusive time of a function is only considered if this function is not already present on a lower stack level. This feature has been overlooked during Vampir development, and will be part of the next release.

Weaving these loosely organised threads together into a coherent tool potentially yields the best parallel profiler available.

This chapter demonstrates the effectiveness of the introduced techniques for two visualisations. First, an existing one is improved. Second, a novel profile viewer is proposed. Its main contributions are greatly enhanced support for parallel programming and improved usability.

5 Conclusion & Future Work

This work introduces methods to cluster similar processes in sub-quadratic time. For this purpose, it presents two measures. One for detecting structural similarity between processes, and one to discern inlining and spawned threads from other differences. Their effectiveness and efficiency to aid performance analysis is demonstrated using 15 different applications with up to 65,000 processes. Due to their interesting characteristics, two of these applications are investigated in greater detail. The improvement of an existing visualisation and introduction of a promising, novel profile viewer exhibit the aptitude of the methods to assist performance visualisation.

The contributions of this thesis advance comparative performance analysis and visualisation. The introduced profile viewer improves over current ones, especially in support for parallel programming and usability. Most notably, it visualises variations in performance measures, like timing and hardware performance counters, between processes using box plots. With this, trends and outliers are easily spotted.

One of the investigated applications shows that the clustering approach is less effective for programs that exhibit irregular function call patterns. This can be detected and steps to alleviate the situation are proposed.

Three categories of future directions of this work can be identified. On the theoretical side, removing the requirement of van der Merwe's algorithm to retain duplicate attributes in the concept lattice would yield an appreciable speedup. For that, a modification of the algorithm is to be devised. To make the approach practical for every application, a method to reduce the lattice node count to arbitrary levels is needed. It must retain meaningful and interesting information and yield intuitive program run lattices. This reduction would also enable visualisation of the lattice itself. This in turn, annotated with information about used APIs and parallel programming paradigms, can yield valuable insight into a program's structure.

A second direction is to advance the implementation. To make it production-ready, the clustering should be parallelised, fall-backs must be implemented, and general optimisation is to be applied.

The third and last direction is to further software performance visualisation. The proposed profile viewer has the potential to be the best in the field and is, therefore, to be implemented as a standalone tool or as a component of an existing visualisation tool, such as Vampir. Leveraging the introduced techniques, novel visualisations to aid developers in understanding their programs and uncovering performance problems should be devised.

Bibliography

[1] Laksono Adhianto, Sinchan Banerjee, Mike Fagan, Mark Krentel, Gabriel Marin, John Mellor-Crummey, and Nathan R Tallent. HPCToolkit: Tools for performance analysis of optimized parallel programs. *Concurrency and Computation: Practice and Experience*, 22(6):685–701, 2010.

[2] Allinea DDT and MAP user guide — viewing stacks in parallel. `https://www.allinea.com/user-guide/DDTControllingProgramExecution.html#x8-9600018`, January 2015.

[3] Allinea MAP: Increase application performance. `http://allinea.com/products/map`, January 2015.

[4] ASC Sequoia benchmark codes – AMG. `https://asc.llnl.gov/sequoia/benchmarks/#amg`, January 2015.

[5] Dieter an Mey, Scott Biersdorff, Christian Bischof, Kai Diethelm, Dominic Eschweiler, Michael Gerndt, Andreas Knüpfer, Daniel Lorenz, Allen D. Malony, Wolfgang E Nagel, Yury Oleynik, Christian Rössel, Pavel Saviankou, Dirk Schmidl, Sameer S. Shende, Michael Wagner, Bert Wesarg, and Felix Wolf. Score-P: A unified performance measurement system for petascale applications. In *Proc. of the CiHPC: Competence in High Performance Computing, HPC Status Konferenz der Gauß-Allianz e.V., Schwetzingen, Germany, June 2010*, pages 85–97. Gauß-Allianz, Springer, 2012.

[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 *Parallel and Distributed Processing Symposium, 2007. IPDPS 2007. IEEE International*, pages 1–10. IEEE, 2007.

[7] David H Bailey, Eric Barszcz, John T Barton, David S Browning, Russell L Carter, Leonardo Dagum, Rod A Fatoohi, Paul O Frederickson, Thomas A Lasinski, Rob S Schreiber, et al. The NAS parallel benchmarks. *International Journal of High Performance Computing Applications*, 5(3):63–73, 1991.

[8] Holger Brunst. *Integrative Concepts for Scalable Distributed Performance Analysis and Visualization of Parallel Programs.* PhD thesis, Technische Universität Dresden, 2007.

[9] Vasily Bulatov, Wei Cai, Jeff Fier, Masato Hiratani, Gregg Hommes, Tim Pierce, Meijie Tang, Moono Rhee, Kim Yates, and Tom Arsenlis. Scalable line dynamics

in ParaDiS. In *Proceedings of the 2004 ACM/IEEE conference on Supercomputing*, page 19. IEEE Computer Society, 2004.

[10] Michael Bussmann, Heiko Burau, Thomas E Cowan, Alexander Debus, Alex Huebl, Guido Juckeland, Thomas Kluge, Wolfgang E Nagel, Richard Pausch, Felix Schmitt, et al. Radiative signatures of the relativistic Kelvin-Helmholtz instability. In *Proceedings of SC13: International Conference for High Performance Computing, Networking, Storage and Analysis*, page 5. ACM, 2013.

[11] Callgrind: A call-graph generating cache and branch prediction profiler. `http://valgrind.org/docs/manual/cl-manual.html`, January 2015.

[12] 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, 2010.

[13] Consortium for small scale modeling (COSMO). `http://www.cosmo-model.org`, January 2015.

[14] Parallel programming and computing platform | CUDA | NVIDIA. `http://www.nvidia.com/object/cuda_home_new.html`, January 2015.

[15] 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*, volume 96, pages 226–231, 1996.

[16] FFTE: A fast fourier transform package. `http://ffte.jp`, January 2015.

[17] Edward W Forgy. Cluster analysis of multivariate data: efficiency versus interpretability of classifications. *Biometrics*, 21:768–769, 1965.

[18] 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*, pages 243–252. ACM, 2010.

[19] Bernhard Ganter. Two basic algorithms in concept analysis. FB4–Preprint 831, TH Darmstadt, 1984.

[20] Bernhard Ganter, Rudolf Wille, and Rudolf Wille. *Formal concept analysis*, volume 284. Springer Berlin, 1999.

[21] The dresden formal concept analysis page. `http://tu-dresden.de/Members/bernhard.ganter/fca/index_html`, January 2015.

[22] Markus Geimer, Felix Wolf, Brian JN Wylie, Erika Ábrahám, Daniel Becker, and Bernd Mohr. The Scalasca performance toolset architecture. *Concurrency and Computation: Practice and Experience*, 22(6):702–719, 2010.

[23] Paul Gibbon, FN Beg, EL Clark, RG Evans, and M Zepf. Tree-code simulations of proton acceleration from laser-irradiated wire targets. *Physics of Plasmas (1994-present)*, 11(8):4032–4040, 2004.

[24] Robert Godin, Rokia Missaoui, and Alain April. Experimental comparison of navigation in a galois lattice with conventional information retrieval methods. *International Journal of Man-Machine Studies*, 38(5):747–767, 1993.

[25] Juan Gonzalez, Judit Gimenez, and Jesus Labarta. Automatic detection of parallel applications computation phases. In *Parallel & Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on*, pages 1–11. IEEE, 2009.

[26] Juan Gonzalez, Judit Gimenez, and Jesus Labarta. Automatic evaluation of the computation structure of parallel applications. In *Parallel and Distributed Computing, Applications and Technologies, 2009 International Conference on*, pages 138–145. IEEE, 2009.

[27] Google Performance Tools – fast, multi-threaded malloc() and nifty performance analysis tools. `https://code.google.com/p/gperftools`, January 2015.

[28] Howto: using gprof with multithreaded applications. `http://sam.zoy.org/writings/programming/gprof.html`, January 2015.

[29] Susan L Graham, Peter B Kessler, and Marshall K Mckusick. Gprof: A call graph execution profiler. In *ACM Sigplan Notices*, volume 17, pages 120–126. ACM, 1982.

[30] GROMACS. `http://www.gromacs.org`, January 2015.

[31] Verena Grützun, Oswald Knoth, and Martin 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):233–242, 2008.

[32] Robert J Hall. Call path profiling. In *Proceedings of the 14th international conference on Software engineering*, pages 296–306. ACM, 1992.

[33] HOMME | high-order methods modeling environment. `https://www.homme.ucar.edu`, January 2015.

[34] HPL – a portable implementation of the high-performance linpack benchmark for distributed-memory computers. `http://www.netlib.org/benchmark/hpl`, January 2015.

[35] Xuegang Hu, Xiang Wei, Dexing Wang, and Peipei Li. A parallel algorithm to construct concept lattice. In *Fuzzy Systems and Knowledge Discovery, 2007. FSKD 2007. Fourth International Conference on*, volume 2, pages 119–123. IEEE, 2007.

[36] Intel Hyper-Threading Technology. `http://www.intel.com/content/www/us/en/architecture-and-technology/hyper-threading/hyper-threading-technology.html`, January 2015.

[37] ASC Sequoia benchmark codes – IRS. `https://asc.llnl.gov/sequoia/benchmarks/#irs`, January 2015.

[38] Intel Trace Analyzer and Collector. `https://software.intel.com/intel-trace-analyzer`, September 2014.

[39] 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.

[40] Anil K Jain. Data clustering: 50 years beyond k-means. *Pattern Recognition Letters*, 31(8):651–666, 2010.

[41] CW Johnson, PB Thistlewaite, D Walsh, and M Zellner. Developing monitoring and debugging tools for the AP1000 array multiprocessor,". In *Proceedings of the Second Fujitsu-ANU CAP Workshop C*, volume 1, page C15, 1991.

[42] Guido Juckeland. *Trace-Based Performance Analysis for Hardware Accelerators*. PhD thesis, Technische Universität Dresden, 2012.

[43] Ian Karlin, Abhinav Bhatele, Jeff Keasler, Bradford L Chamberlain, Jonathan Cohen, Zachary DeVito, Riyaz Haque, Dan Laney, Edward Luke, Felix Wang, et al. Exploring traditional and emerging parallel programming models using a proxy application. In *Parallel & Distributed Processing (IPDPS), 2013 IEEE 27th International Symposium on*, pages 919–932. IEEE, 2013.

[44] Leonard Kaufman and Peter Rousseeuw. *Clustering by means of medoids*. North-Holland, 1987.

[45] Andreas Knüpfer. *Advanced Memory Data Structures for Scalable Event Trace Analysis*. PhD thesis, Technische Universität Dresden, 2008.

[46] Andreas Knüpfer, Ronny Brendel, Holger Brunst, Hartmut Mix, and Wolfgang E Nagel. Introducing the open trace format (OTF). In *Computational Science–ICCS 2006*, pages 526–533. Springer, 2006.

[47] Andreas Knüpfer, Holger Brunst, Jens Doleschal, Matthias Jurenz, Matthias Lieber, Holger Mickler, Matthias S Müller, and Wolfgang E Nagel. The Vampir performance analysis tool-set. In *Tools for High Performance Computing*, pages 139–155. Springer, 2008.

[48] Andreas Knüpfer, Bernhard Voigt, Wolfgang E Nagel, and Hartmut Mix. Visualization of repetitive patterns in event traces. In *Applied Parallel Computing. State of the Art in Scientific Computing*, pages 430–439. Springer, 2007.

[49] FD4: Four-dimensional distributed dynamic data structures. `http://wwwpub.zih.tu-dresden.de/~mlieber/fd4`, January 2015.

[50] Matthias Lieber, Ralf Wolke, Verena Grützun, Matthias S Müller, and Wolfgang E Nagel. A framework for detailed multiphase cloud modeling on HPC systems. In *PARCO*, pages 281–288, 2009.

[51] Zongtian Liu, Liansheng Li, and Qing Zhang. Research on a union algorithm of multiple concept lattices. In *Rough Sets, Fuzzy Sets, Data Mining, and Granular Computing*, pages 533–540. Springer, 2003.

[52] Stuart Lloyd. Least squares quantization in PCM. *Information Theory, IEEE Transactions on*, 28(2):129–137, 1982.

[53] Livermore Unstructured Lagrangian Explicit Shock Hydrodynamics (LULESH). `https://codesign.llnl.gov/lulesh.php`, January 2015.

[54] Piotr R Luszczek, David H Bailey, Jack J Dongarra, Jeremy Kepner, Robert F Lucas, Rolf Rabenseifner, and Daisuke Takahashi. The HPC Challenge (HPCC) benchmark suite. In *Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, page 213, 2006.

[55] James MacQueen et al. 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. California, USA, 1967.

[56] Robert McGill, John W Tukey, and Wayne A Larsen. Variations of box plots. *The American Statistician*, 32(1):12–16, 1978.

[57] J Michalakes, S Chen, J Dudhia, L Hart, J Klemp, J Middlecoff, and W Skamarock. Development of a next generation regional weather research and forecast model. In *Developments in Teracomputing: Proceedings of the Ninth ECMWF Workshop on the use of high performance computing in meteorology*, volume 1, pages 269–276. World Scientific, 2001.

[58] 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*, page 55. ACM, 2009.

[59] Message Passing Interface (MPI) forum. `http://mpi-forum.org`, January 2015.

[60] Patrick Njiwoua and E Mephu Nguifo. A parallel algorithm to build concept lattice. In *Proceedings of the 4th Groningen International Information Technology Conference for Students*, volume 107, 1997.

[61] OpenMP specifications. `http://openmp.org/wp/openmp-specifications`, January 2015.

[62] ParaDiS: Dislocation simulations. `http://paradis.stanford.edu`, October 2014.

[63] 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*, volume 44, pages 17–31, 1995.

[64] Uta priss' formal concept analysis homepage. `http://www.upriss.org.uk/fca/fca.html`, January 2015.

[65] Qt Project. `http://qt-project.org/`, January 2015.

[66] 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*, pages 46–55. ACM, 2008.

[67] Steve Reiniger. Bestimmung und Darstellung von Ähnlichkeiten in Programmspuren mit parallelen Programmablaufgraphen. Diploma thesis, Dresden, Technische Universität, 2013.

[68] Burkhard Rockel, Andreas Will, and Andreas Hense. The regional climate model COSMO-CLM (CCLM). *Meteorologische Zeitschrift*, 17(4):347–348, 2008.

[69] Martin Schulz and Bronis R de Supinski. Practical differential profiling. In *Euro-Par 2007 Parallel Processing*, pages 97–106. Springer, 2007.

[70] 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):105–121, 2008.

[71] Sameer S Shende and Allen D Malony. The TAU parallel performance system. *International Journal of High Performance Computing Applications*, 20(2):287–311, 2006.

[72] Fengguang Song, Felix Wolf, Nikhil Bhatia, Jack Dongarra, and Shirley Moore. An algebra for cross-experiment performance analysis. In *Parallel Processing, 2004. ICPP 2004. International Conference on*, pages 63–72. IEEE, 2004.

[73] single program multiple data — dictionary of algorithms and datastructures — national institute of standards and technology. `http://xlinux.nist.gov/dads//HTML/singleprogrm.html`, January 2015.

[74] Hugo Steinhaus. Sur la division des corps matériels en parties. *Bulletin de l'Académie Polonaise des Sciences*, 1:801–804, 1956.

[75] Bjarne Stroustrup et al. *The C++ programming language*. Pearson Education India, 1995.

[76] TOP500 supercomputer sites. `http://top500.org`, October 2014.

[77] Intel Turbo Boost Technology 2.0. http://www.intel.com/content/www/us/en/architecture-and-technology/turbo-boost/turbo-boost-technology.html, January 2015.

[78] Amos Tversky. Features of similarity. *Psychological Review*, 84(4):327–352, 1977.

[79] VampirTrace. http://tu-dresden.de/zih/vampirtrace, January 2015.

[80] Dean Van Der Merwe, Sergei Obiedkov, and Derrick Kourie. AddIntent: A new incremental algorithm for constructing concept lattices. In *Concept Lattices*, pages 372–385. Springer, 2004.

[81] David Van Der Spoel, Erik Lindahl, Berk Hess, Gerrit Groenhof, Alan E Mark, and Herman JC Berendsen. GROMACS: fast, flexible, and free. *Journal of computational chemistry*, 26(16):1701–1718, 2005.

[82] Matthias Weber, Ronny Brendel, and Holger Brunst. Trace file comparison with a hierarchical sequence alignment algorithm. In *IEEE 10th International Symposium on Parallel and Distributed Processing with Applications*, pages 247–254. IEEE, 2012.

[83] Matthias Weber, Kathryn Mohror, Martin Schulz, Bronis R. de Supinski, Holger Brunst, and Wolfgang E Nagel. Alignment-based metrics for trace comparison. In *Euro-Par 2013 Parallel Processing*, pages 29–40. Springer, 2013.

[84] Josef Weidendorfer. Sequential performance analysis with Callgrind and KCachegrind. In *Tools for High Performance Computing*, pages 93–113. Springer, 2008.

[85] The weather research & forecast model. http://wrf-model.org, January 2015.

[86] ARK | Intel Xeon processor E5–2690. http://ark.intel.com/products/64596/Intel-Xeon-Processor-E5-2690-20M-Cache-2_90-GHz-8_00-GTs-Intel-QPI, January 2015.

[87] Intel Xeon Phi product family. http://www.intel.com/content/www/us/en/processors/xeon/xeon-phi-detail.html, January 2015.