Dresden University of Technology

Department of Computer Science

Institute of Computer Engineering

Chair of Computer Architecture

Hauptseminar
Rechnerarchitektur und Programmierung

Evaluating Visualisation Methods to Scalibly Display Structural and Runtime Differences in Parallel Programs

Author: Ronny Brendel (Mat.-No.: 3392512)
Professor: Prof. Dr. Wolfgang E. Nagel
Tutor: Matthias Weber

Dresden, 19th November, 2018

Contents

1 Introduction

With the stagnating per-core performance, the number of cores in processors increase. To exploit the potential performance of modern systems, software needs to be tailored to use the system's parallelism. Due to the always rising complexity of software and the push for parallelization, much execution speed is lost due to synchronisation time, load imbalances and other software inefficiencies. Therefore, performance optimisation has become an integral part of the software development process. Before and after resolving a specific performance problem, one always analyses the software, searching for the root cause and afterwards analysing the results of the chosen optimisation.

Comparison in general is an important part of analysis. One wants to see the difference an optimisation makes, or compare the performance of multiple versions of a program. It is also interesting to compare program runs on different hardware platforms, or compare threads of execution inside the same run to detect where, e.g., load imbalances arise.

Aside from visually comparing, automatic performance comparison can be used to aid the user in various ways. For example an automatic preselection of structurally similar processes can be presented for further timing difference inspection. Furthermore, comparison information can be used to improve today's performance analysis tools by, e.g., omitting to display redundant information about millions of similar threads of execution.

Current analysis tools mostly present information in an isolated manner to the user and make no to few use of comparison techniques.

In this work we present early results of our exploration of possibilities on how to visualise comparisons of program executions. This includes a simple attempt at visualising profiles and a more complex variation on the profile theme.

2 Background & Related Work

In this section we give an overview about current ways to measure, visualise and compare program performance information.

2.1 Measuring Performance Data

A wealth of different measures can be gathered during a program run. Most notable are timing, number of function invocations, memory usage and hardware performance counters like the current number floating point operations per second the CPU is putting through.

There are two large groups of performance data: profiles and traces. Profiles are tables of per-function accumulated information. They are coarse, small in size and give a good overview of the program's behaviour. Traces, in contrast, record threads of execution in great detail including each function call and its timing. This enables more detailed analysis than profiles can offer, but comes at the cost of additional measurement and analysis overhead. Profiles are usually presented in a static way (Figure 1) whereas the sheer size of traces (multiple tens or hundreds of mebibytes per thread of execution). Figure 2 shows an example visualisation of a trace containing multiple vertically arranged threads of execution. Lines between threads mark inter-process communication. For Process 8 the full function call stack over time is displayed.

Figure 1: Excerpt of a C++ program's profile

% Time	Self Seconds	Calls	Name
7.75	0.11	69659721	`QList::Node::t`
4.93	0.07	30581879	`std::move`
2.11	0.03	10224420	`QListData::isEmpty`
2.11	0.03	2368468	`QMapNode::lowerBound`
2.11	0.03	2367640	`Measure_record`
1.41	0.02	7220479	`std::swap`
0.70	0.01	591910	`handleEnter`

In order to gather the information displayed, one uses *instrumentation* or *sampling*. Instrumentation invokes the measurement routines every time a function call is started and on each return from a function call. When sampling, the measurement code is invoked in fixed intervals, instead.

Tools that implement performance measurement include VampirTrace [8], which uses the Open Trace Format [7] (OTF) for writing traces to files. The

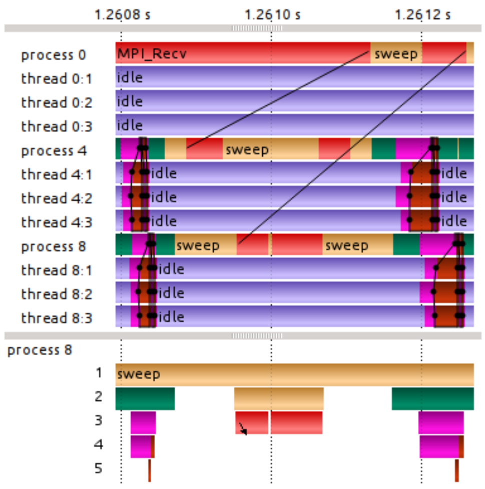


Figure 2: Example trace

Scalasca [2] toolkit provides similar functionality. Today both measurement infrastructures are in the process of being replaced by the co-developed Score-P [1] framework, including the successor trace file format OTF 2.

2.2 Visualising Performance Data

The Vampir toolkit [8] offers many different visualisations, from which the user can pick the appropriate for his investigations.

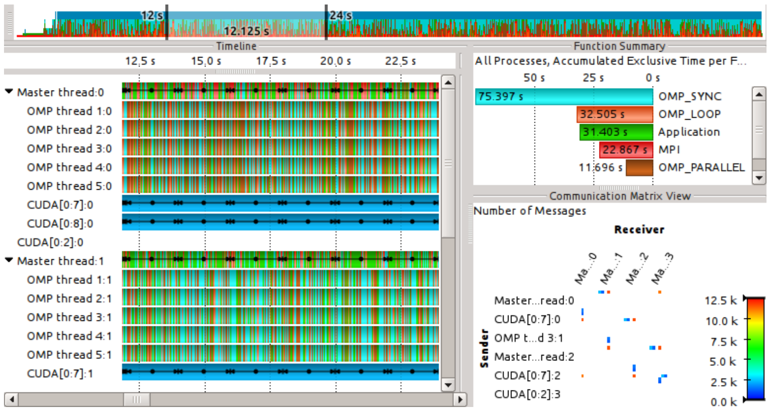


Figure 3: Showcase of some Vampir displays

Figure 3 shows a number of these, so called, *displays*. On the left side the *Master Timeline* is situated. It shows the currently active function, coded in colour, over time of each thread of execution. Arrows and black lines encode inter-process communication behaviour. At the top, the *Zoom Toolbar* is shown. Its primary use is quick navigation through the trace. It displays the same information as the Master Timeline, but sorts colours vertically to show which function is active over time across all processes. This way a good quantitative overview can be gained and phases in the program's execution can be seen. In the top-right, a traditional profile is displayed. In this screenshot functions are grouped to show a quantitative overview of how much time is spent in which set of functions. In the bottom-right, the *Communication Matrix* is situated. It encodes statistics about which thread of execution (horizontal) communicates with which threads (vertical) in colour. In this example the number of messages exchanged between all processes is displayed. In each display of Vampir the user can zoom in to inspect details.

Other tools which visualise traces are the Intel Trace Analyzer [5] and Cube [12]. The Intel Trace Analyzer has similar, but more limited, functionality to vampir. Cube (part of Scalasca) offers automatic detection of performance issues and visualizes using a specialized three-column table-esque layout.

2.3 Comparing Performance Data

One simple way to help users comparing traces, is to display multiple program runs side-by-side (Figure 4). Vampir is able to display an arbitrary

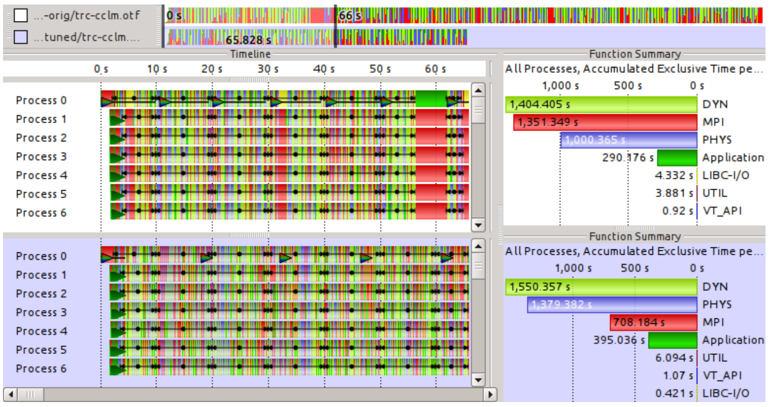


Figure 4: Vampir displaying two traces simultaneously

number of traces at the same time. The user can manually shift timelines left and right to aid visual comparison.

The *Complete Compressed Call Graph* [6] approach aims at reducing trace file size by representing the stack over time of a thread of execution as a tree of function calls including timing information. Sub-trees of equal structure and similar timing will then be merged into one. This way a directed acyclic graph, which mimics the original tree and has fewer nodes, arises. This format could then be used to analyse and visualise similarities and differences in a trace or between multiple traces.

eGprof [10] calculates textual differences between two GNU gprof [3] profiles. It also contains a simple visualisation of differences between call trees. In contrast to call stack over time, it is a compressed tree visualising all actually used call stack configurations enriched by accumulated profile information.

Open|SpeedShop [11], too, offers comparison between multiple profiles.

Cube uses performance information about multiple program runs to display differences with its profile-esque graphical user interface. It can also combine information about multiple runs to theoretical runs and offer information about not-yet tried program configurations.

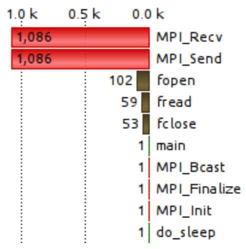
A more detailed comparison of traces has been done by Weber et al. [13]. The article describes how to compare two call stacks over time using alignment algorithms from bioinformatics. In a follow up article [14], a similarity metric based on alignment of traces is demonstrated. Furthermore, a Vampir display prototype which shows the timing difference between multiple threads of execution over the course of the program's execution is presented.

Generally speaking, trace comparison is still young and performance analysis tools make very limited use of comparison techniques. Therefore, we want to explore new ways of visualising, and generally leveraging, differences in traces.

3 Ideas & Evaluation

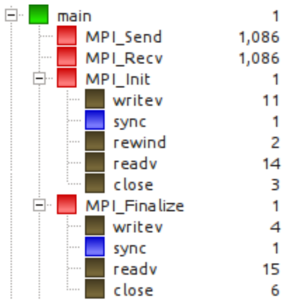
Because visualising differences between traces is much ground to cover, we place reasonable restrictions on the scope of our work. We only use information about the function call stack and its timings. That excludes communication information, performance counters and resource usage statistics. We only compare threads of execution inside the same program run. This way we can a make number of assumptions and do not need to cope with problems that arise when program configurations and environments differ. Recently there has been a push for performance analysis and visualisation during a program's execution. For the sake of simplicity we assume that a program run is completed when we start our analysis. Lastly, we restrict our visualisation attempts to profil-ish information.

By profil-ish we roughly mean information accumulated into a number of buckets. There are three basic ways to classify information. Traditional profiles accumulate information for each function (Figure 5). Call trees gather information for each call stack configuration (Figure 6). Accumulated measures are put into different buckets depending from which source a function has been called.



1,086	MPI_Recv
1,086	MPI_Send
102	fopen
59	fread
53	fclose
1	main
1	MPI_Bcast
1	MPI_Finalize
1	MPI_Init
1	do_sleep

Figure 5: Simple example profile showing the number of function invocations



```
main                  1
  MPI_Send        1,086
  MPI_Recv        1,086
  MPI_Init            1
    writev           11
    sync              1
    rewind            2
    readv            14
    close             3
  MPI_Finalize        1
    writev            4
    sync              1
    readv            15
    close             6
```

Figure 6: Simple example call tree showing the number of function invocations

Profiles are simple to compare, but lack structural data. For example if messages get sent and received by two processes in similar quantities but vary in purpose, no difference between the two processes can be detected. A call tree gives context to functions, in that it considers the source where functions are called from. In turn, it is harder to compare trees than it is to compare profiles. Simple differential comparison will not work for our purpose. For example if you compare two processes which are the same but some functions have been inlined, the call trees would then have less nodes in between and simple differential comparison will not yield that these processes are similar.

Because of these draw-backs of profiles and call trees we devise a hybrid called *call matrix* (Figure 7). In a call matrix the information is accumulated

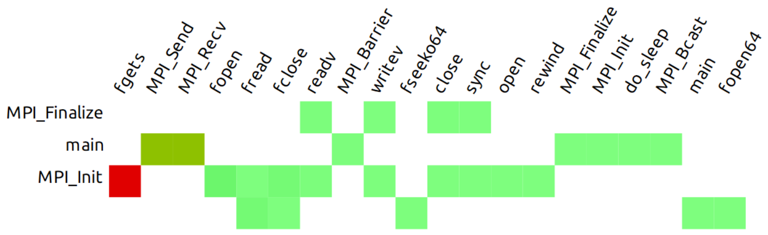


Figure 7: Simple example call matrix showing the number of function invocations coded in colour

in buckets of calling function and called function. This way we retain some context of the circumstances of a function call, and we can compare them element-wise. Furthermore, one can compute the transitive closure of a call matrix which helps dealing with the inlining problem described above.

The performance information we are using is function call timing and invocation counts. When measuring timing we distinguish between exclusive and inclusive time. Exclusive time, is the time spent in a function, where inclusive time also includes time spent in sub-calls (Figure 8).

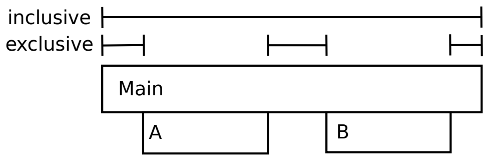


Figure 8: Exclusive and inclusive time

We take these three measures and derived values like minimum, maximum, average and standard deviation. Minimum and maximum are not very useful measures when looking at function call timings, because due to system noise and scheduling effects they will most likely be very different from most other measurement points. The average is much influenced by outliers. Furthermore, we think the standard deviation is hard to interpret intuitively.

During our research we came to prefer the median and quantiles, instead. Imagine all measurement points (e.g. exclusive execution time of each function call of one particular function) sorted from smallest to largest. The median is then the element in the middle of that list, or the average of the two middle elements if the list is of even length. Quantiles generalize the idea of the median to arbitrary points in the list. The second percentile for example is then the element situated at the two percent position in the sorted measurement point list. In comparison to average the median is a more robust measure, and one can say half of the measurement points are less and half are larger than the median. Likewise 98% of all measurement points are larger than the second percentile.

Medians and quantiles can nicely be drawn into box plots (Figure 9). Box

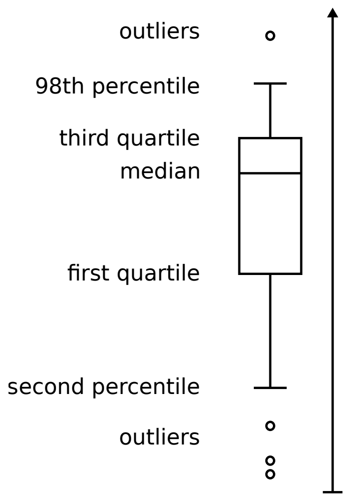


Figure 9: Box plot

plots vary in definition of the middle, box borders, and whiskers. We use 2%, 25% (first quartile), 50% (median), 75% (third quartile) and 98% for our visualisation. That means half of all function calls have execution times inside of the box. 96% of function calls have execution times between the two whiskers. Using average and standard deviation, no such claims could be made. Furthermore, using the standard deviation for the box, there would be no distinction between upper and lower box border.

In the following two subsections, we present our humble attempt at visualising profiles and call matrices using box plots.

3.1 Profile

Figure 10 shows a simple visualisation of a box plot-based profile. It depicts

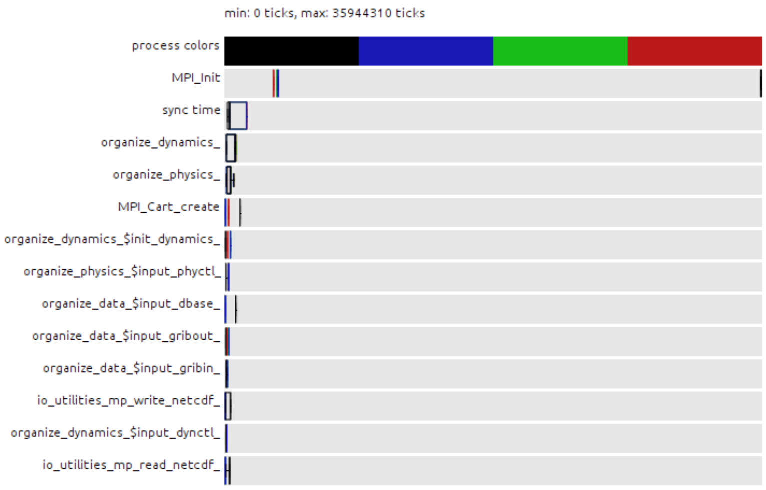


Figure 10: Profile showing the exclusive time distributions of four processes simultaneously

the exclusive time of each function used in the COSMO-SPECS weather simulation software [4]. Each of the four processes have one box per function drawn in a distinct colour. The colours for each process, left to right, can be seen in the first column. At first glance, one can see that `MPI_Init` takes very different amounts of time on different processes. Hardly anything else can be seen for other functions. For example, if a function is called on a certain process at all, is hidden. To reveal more detail, one needs to zoom in.

Figure 11 shows exemplary the seven most-called functions. One can see

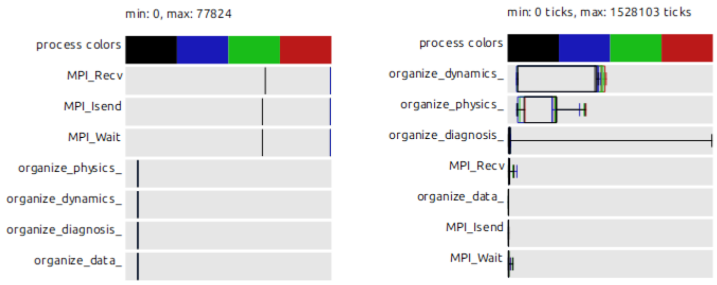


Figure 11: Invocation count (left) and exclusive time (right) profile of the seven most called functions, four processes

that `MPI_Isend`, `MPI_Recv` and `MPI_Wait` have different function invocation counts on different processes. In contrast, the `organize_*` functions have similar invocation counts across all processes. `organize_dynamics_` and `organize_physics_` have similar exclusive time distributions across processes. Furthermore, the right whisker of the black box plot of `organize_diagnosis_` is very far extended to the right, which means that at least two percent of function calls take considerably longer than most others on process one.

Figure 12 presents the same information as Figure 10, but for 64 processes. Again, one can see that `MPI_Init`'s execution time distribution is

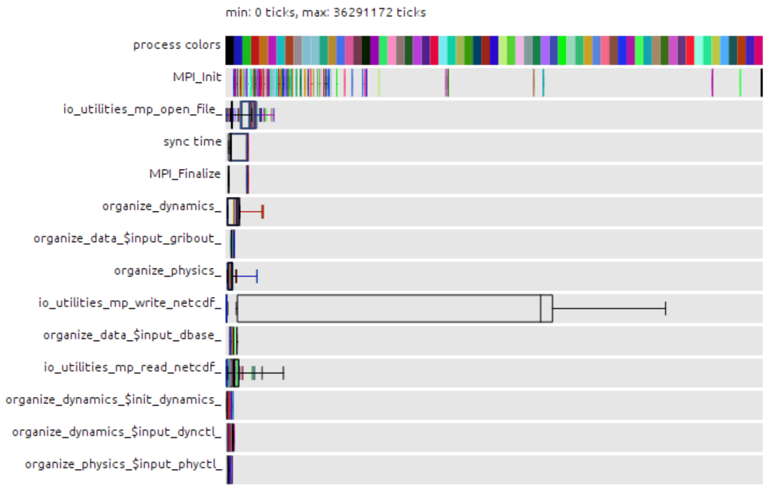


Figure 12: Profile showing the exclusive time distributions of 64 processes simultaneously

very different across processes. Calls to `io_utilities_mp_write_netcdf_` take long and very differently long on process one (black large box plot).

Figure 13 is, again, a profile of the seven most-called functions. Processes call `MPI_Isend`, `MPI_Recv`, `MPI_Wait` differently often. The processes can be divided into three sets according to their invocation counts. The rest of the interpretation is the same as for the four process case (Figure 11).

From a box plot, one cannot tell how measurement points underlie. If there is only one call to a function, the box plot collapses to just one vertical line. For very low invocation counts it degenerates. A common approach to mitigate this problem to adjust the width of the box plot according to the number of measurement points. A thinner box plot has less significance to it than a broader.

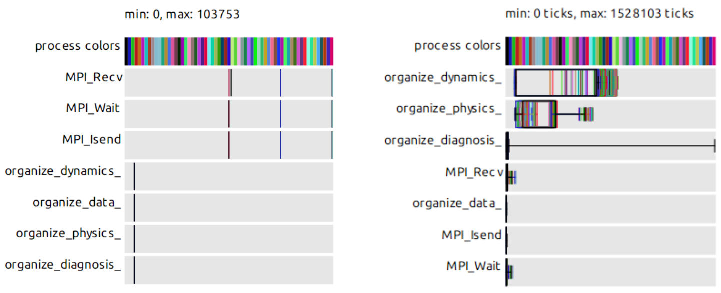


Figure 13: Invocation count (left) and exclusive time (right) profile of the seven most called functions, 64 processes

3.2 Call Matrix

Because the call matrix (Figure 7) shows the calling functions vertically and called functions horizontally, we need to use the third dimension to encode the distribution of execution times. Therefore, we encode box plots as squares, and reserve different areas for different quantities. The quantities themselves are encoded in colour. Figure 14 how this is done. The area of

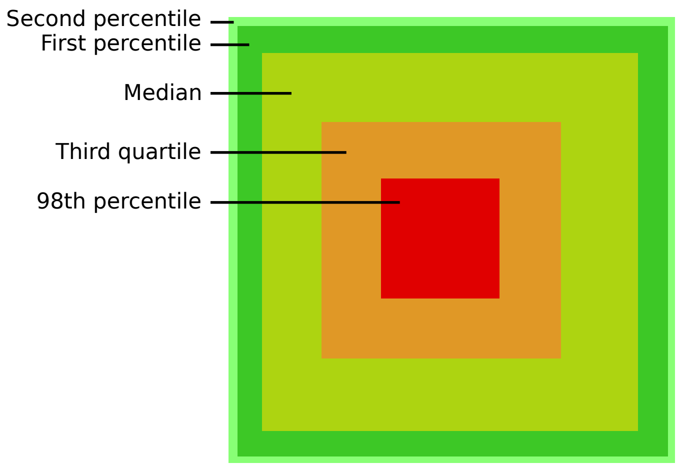


Figure 14: Colour-coded box plot

the square is divided onto 14 equal parts of which, from the lowest to the largest box plot value, 1, 3, 6, 3 and 1 part(s) are given. I.e., the second and the 98th percentile cover an area of the same size. Quartiles cover three times as much area. Finally the median covers twice the area of a quartile. The colour encoding of the specific quantity is then done via a linear gradient from green (low) to red (high) (Figure 15).

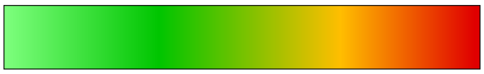


Figure 15: Linear gradient from bright green over orange to red

To demonstrate our prototype, we begin with a simple example walk-through of a ping pong parallel program. It repeatedly sends messages between processes. During the walk-through, we stick to displaying the inclusive execution time of functions. Figure 16 shows the complete call matrix of the example program. The nameless calling function is a virtual root function and shows which functions are called initially without any underlying other visible function. One can see that most time is spent in `main`. The functions which themselves call functions are `main`, `MPI_Init`

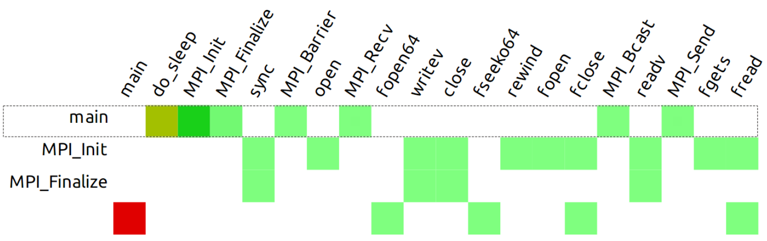


Figure 16: Call matrix of a simple program showing the inclusive time

and `MPI_Finalize`. This does not mean they don't call any functions, just that low level functions themselves have not been augmented with instrumentation and therefore no information about their innards is available.

We now select the `main` function and focus the view to display only the functions which `main` calls (Figure 17). Since we only display a single column, one could use, instead of the call matrix layout, a profile view. That profile would then show profile data of function calls that have been issued by `main` directly. Investigating Figure 17, one can see that much time is spent sleeping, initialising and shutting down the MPI inter-process communication library.

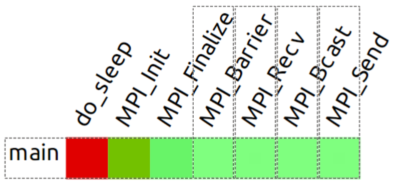


Figure 17: Call matrix limited to functions called by `main`

We, therefore, restrict the view to only those function that do real work (Figure 18). For the first time in our inspection, visible colour-coded box

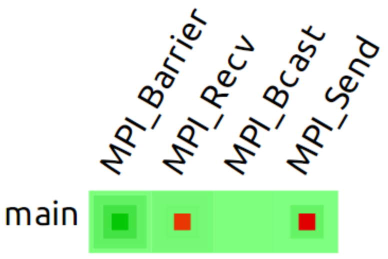


Figure 18: Call matrix limited to interesting functions called by `main`

plots can be observed. Before this point, the low number of calls and breadth of execution times stretched the linear gradient too thin. One can now see that `MPI_Barrier` takes slightly different amounts of time and that at least two percent of all `MPI_Send` and `MPI_Recv` calls take considerably longer than 75% (the third quartil) of all calls.

Going one step further, we now draw multiple call matrices into one view (Figure 19). We do this by partitioning each square into n, where n is

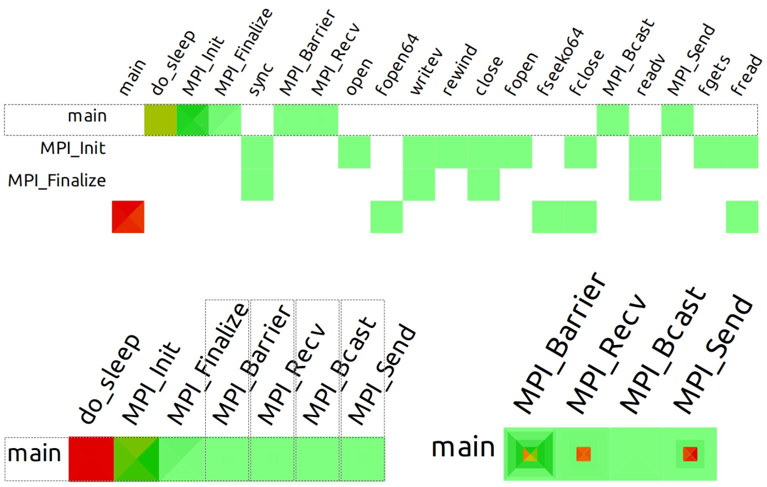


Figure 19: Call matrix for four processes and inclusive time

the number of threads of execution, equally sized slices. Slicing starts in the top-left corner and continues clock-wise. The relative areas for these slices of squares are preserved. So the partitioned colour-coded box plot still has correct proportions. In Figure 19, one can see that the execution time distribution of `MPI_Barrier` varies between the four processes. Other execution times vary only slightly.

The linear gradient depends on the lowest and highest value displayed and is, therefore, often stretched very thin so that in order to see differences in execution times, one needs to select fewer functions.

Having seen a simple example, we now present a real world code, namely the LINPACK benchmark, using 32 processes (Figure 20). Although this trace has relatively few functions it is already very confusing. Normal computer programs have thousands of functions. To alleviate this problem multiple options are available. One could group functions and introduce a view that can collapse multiple functions into their groups. Another possibility is to pre-select interesting functions and display only these.

If one restricts the view to fewer functions, pictures like Figure 21 emerge. One can see that every process sends and receives messages using MPI routines. Some of the processes have different sources where MPI functions are

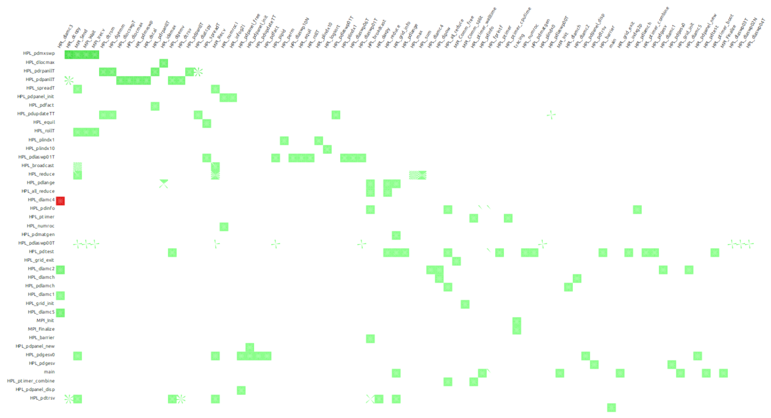


Figure 20: Call matrix of the LINPACK benchmark on 32 processes

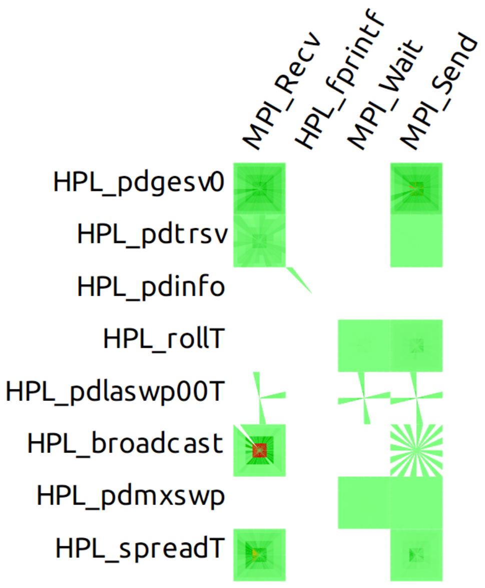


Figure 21: Call matrix limited to fewer interesting functions showing exclusive time

called from. That means some processes have different purposes. Only the first process calls `HPL_fprintf`, which is likely a function for printing out status information. One can see that patterns of function use emerge. In our research, we never observed great variations in the function caller/callee relation ships, which is what one expects because the relation ships only emerge from the static source code and are mostly insensitive to runtime behaviour. This sparks the hope that we can use this information to automatically group processes by introducing a call matrix-based similarity measure. In contrast to previous work, it would be conceptionally much simpler and computationally quicker. Some programs show no pattern at all, i.e. every called function is called by same functions on every process.

Figure 22 summarises some other patterns that one can observe when examining a call matrix.

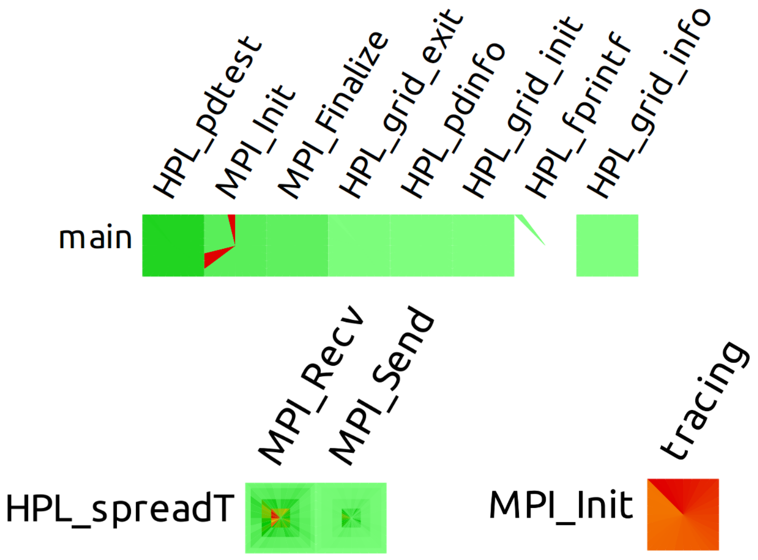


Figure 22: Selected parts of the LINPACK call matrix showing exclusive time

- `MPI_Init` takes very long on three processes.
- `MPI_Recv`'s execution time is very unevenly spread.
- `tracing` starts a bit later on consecutive processes, but all calls finish at the same time. This way a circular gradient emerges.

Concluding the presentation of the call matrix, we want to motivate one more thing. In Figure 23, one can see that multiple functions send and

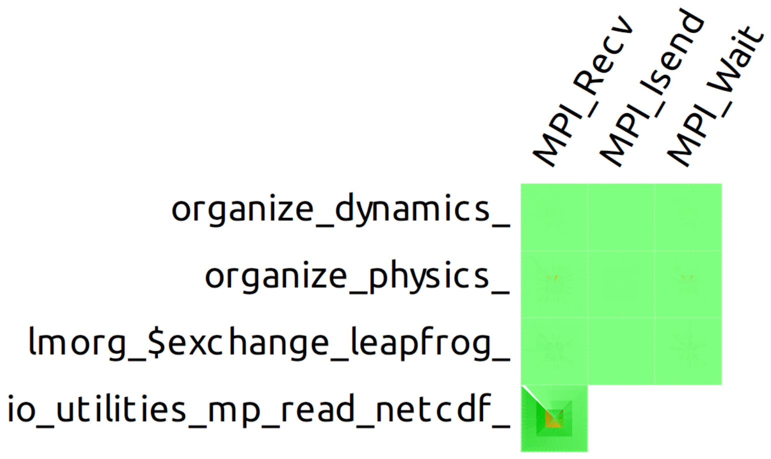


Figure 23: Selected calling functions for send, receive and wait in COSMO-SPECS

receive messages. The execution times of the receive calls issued by `io_utilities_mp_read_netcdf_` have wildly varying execution times. Restricting the view to even fewer callers (Figure 24), one can see that the execution times for sending and receiving message is very differently distributed depending on which function issues them. In a traditional profile,

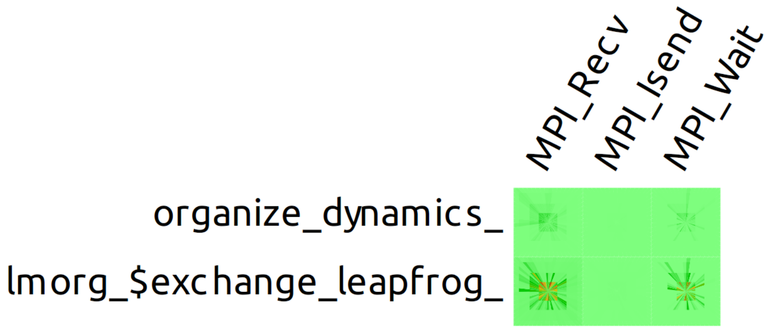


Figure 24: Send and receive statistics of two calling functions in COSMO-SPECS

all send and receive calls are put into one statistic. Therefore, the differences it makes which function issues the communication is evened out by information which would, in the call matrix case, be put into different buckets. Using a call matrix one could detect the execution time differences and also know where exactly they are issued from. We, therefore, conclude that filtering profiles by calling functions is something to consider.

4 Conclusion & Future Work

Summarising our work on comparing profil-ish information we think that box plot-based profiles are a viable way to get a qualitative overview of a program's runtime behaviour. Call matrices, in its current state, are not suited to be displayed to end users. It is still a promising technology which can on the one hand be improved to fit the users needs. On the other hand it is a simple and helpful tool for automatically detecting similar threads of execution. Aside from comparing, we conclude that median, quantiles are better suited than minimum, maximum, average and standard deviation to be used in profiling software. The downside is that more work is required to compute them. Lastly, we think that filtering profiles by calling functions or call stack configurations should be considered for implementation in profilers.

Our short-term goal is to take a stab at comparing and visualising call trees and develop a similarity metric based on call matrices and maybe profiles. In the long run we want to visualise comparisons of the whole call stack over time of multiple threads of execution.

Bibliography

[1] 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.

[2] 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.

[3] 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.

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

[5] Intel trace analyzer and collector. `https://software.intel.com/intel-trace-analyzer`.

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

[7] 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.

[8] 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.

[9] 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.

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

[11] 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.

[12] 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.

[13] 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.

[14] 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.