Available online at www.prace-ri.eu

Partnership for Advanced Computing in Europe

Profiling and Tracing Tools for Performance Analysis of Large Scale Applications

Jerry Eriksson[b], Pedro Ojeda-May[b], Thomas Ponweiser[a,*], Thomas Steinreiter[a]

[a]*RISC Software GmbH, Softwarepark 35, 4232 Hagenberg, Austria*
[b]*High Performance Computing Center North (HPC2N), MIT Huset, Umeå Universitet, 901 87 Umeå, Sweden*

Abstract

The usage of modern profiling and tracing tools is vital for understanding program behaviour, performance bottlenecks and optimisation potentials in HPC applications. Despite their obvious benefits, such tools are still not that widely adopted within the HPC user community. The two main reasons for this are firstly unawareness and secondly the sometimes inhibitive complexity of getting started with these tools. In this work we aim to address this issue by presenting and comparing the capabilities of four different performance analysis tools, which are 1) HPCToolkit, 2) Extrae and Paraver, 3) SCALASCA and 4) the Intel Trace Analyzer and Collector (ITAC). The practical usage of these tools is demonstrated based on case studies on the widely used molecular dynamics simulation code GROMACS.

Keywords: Performance Analysis, Profiling, Tracing, HPCToolkit, Extrae, Paraver, SCALASCA, ITAC, GROMACS

1. Introduction

Background of this work is a survey conducted by the PRACE project in the beginning of 2016. In order to develop links to European Centres of Excellence (CoEs) and to understand their needs with respect to high performance computing, a questionnaire has been developed and circulated to each of eight selected CoEs. The results of this survey are summarised in the PRACE-4IP deliverable D7.3 [1].

One finding of this survey which motivated the present work is that from the CoE's answers it is evident that still the use of profiling tools for code optimization is not that well established. The most favoured method for performance analysis among all CoEs is still manual code instrumentation (i.e. inserting code for time measurement) and console or log output – compare Figure 1. However, for a well-grounded and deep understanding of program performance behaviour and optimisation opportunities, the usage of specialised profiling tools is important already today and will get even more important in view of the ever-increasing complexity and heterogeneity of HPC systems on the road to Exascale.

With this work, we aim to raise the general awareness for the benefits of using specialised performance analysis tools and to lower the threshold for potential users looking to get started with such tools. We do so by highlighting and comparing the capabilities of four different profiling and tracing toolsets, which are 1) HPCToolkit, 2) Extrae and Paraver, 3) SCALASCA and 4) the Intel Trace Analyzer and Collector (ITAC). We emphasise that the selection of exactly these four toolsets is not to be understood as a special recommendation. Instead, it has more the character of a random sample which should give potential users an idea on what level of insight the use of profiling and tracing tools can provide. We encourage potential users also to consider other tools, as for example those listed in section 3 of PRACE-3IP deliverable D7.2.1 [2].

[*] Corresponding author. *E-mail address*: thomas.ponweiser@risc-software.at

For demonstrating the practical usage of the above-mentioned tools, the application code GROMACS [3] has been selected due to its high relevance for the molecular dynamics community and in particular for the three CoEs NOMAD, BioExcel and E-CAM. Of course most information in this document (like tool descriptions, hints, best practices, etc.) is not specific to applications with GROMACS and therefore may be also useful for an even broader audience.

Apart from that, in particular our investigations on Extrae and Paraver may be of interest for the BioExcel CoE, as they expressed interest to use this toolset more frequently in future [1]. Secondly, also the PoP CoE, in particular the members who are developers of the Scalasca and Extrae/Paraver toolsets, may be interested on this project's findings and conclusions on Scalasca and Extrae/Paraver.

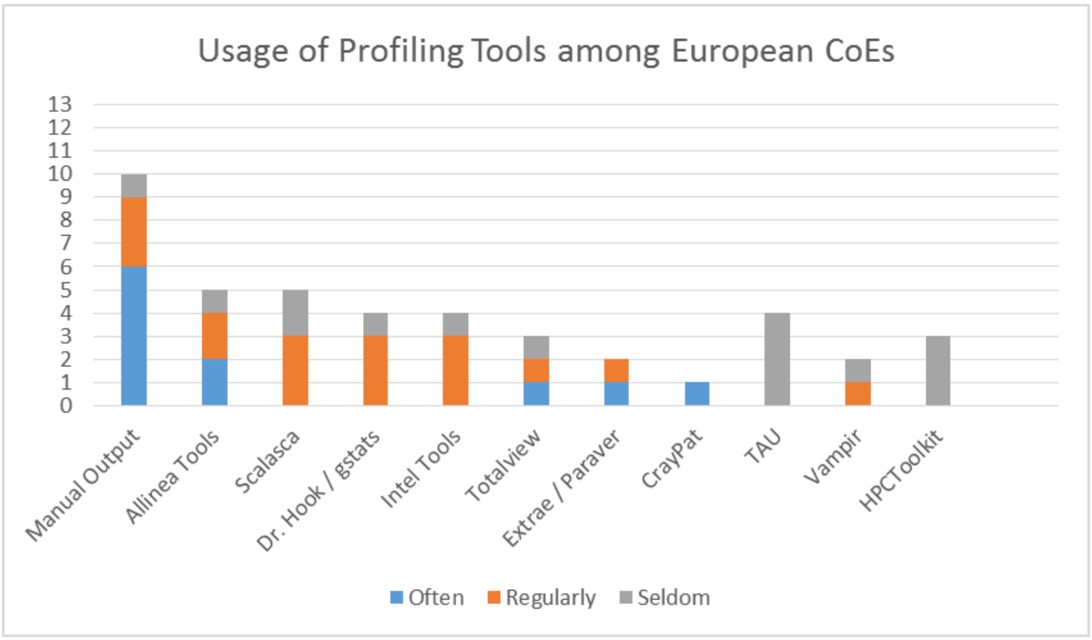


Figure 1: Survey result on the usage frequency of profiling tools among European Centres of Excellence.
Data from 13 points of contact from BioExcel (1), CoeGSS (1), E-CAM (1), ESiWACE (9) and MaX (1) is displayed.
Manual performance measurement and console output is the most favoured method. Dedicated profiling tools are used less frequently.

1.1. Document Structure

This white paper is structured as follows: In the remainder of this section, we shortly introduce the molecular dynamics package GROMACS. Sections 2 to 5 are devoted to each of the tools considered in this work, which are HPCToolkit, Intel Trace Analyzer and Collector (ITAC), Extrae/Paraver and SCALASCA. In each of these sections, we give a brief tool overview and introduction as well as an exemplary case study for demonstrating one selected tool feature. Additionally, where applicable, we give complementary hints and best practices for the usage of the tools, based on our own experience which are either missing or difficult to find in currently available documentation. Section 6 provides an overall overview by briefly comparing the tools with respect to their features and limitations. Finally, Section 7 concludes with a summary of our findings.

1.2. GROMACS

The *GROningen MAchine for Chemical Simulations (GROMACS)* [3] is one of the most widely used open-source and free software codes in chemistry, used primarily for dynamical simulations of biomolecules. It provides a rich set of simulation schemes, preparation and analysis tools. GROMACS is one of the fastest and most scalable molecular dynamics software packages currently available. Its outstanding performance is due to the implementation of novel methodologies and algorithms for dealing with computationally intensive tasks.

For instance, electrostatic interactions, which are the most expensive part in Biomolecular simulations, are solved through the *Particle Mesh Ewald (PME)* method [4]. Here, particle-particle interactions up to a predefined cut-off distance, which are also called *short-range interactions*, are computed explicitly while the rest of the interactions, i.e. so-called *long-range interactions*, are computed in reciprocal space through Fourier transformations.

In distributed computing, Fourier transformations by nature involve heavy communication which is often a limiting factor for scalability. For this reason, GROMACS partitions the set of compute processes into a *particle-particle (PP) group* for computing short-range interactions, and a comparatively smaller *PME group* for computing long-range interactions. If the number of processes in the PP and PME groups is well balanced, this method greatly improves performance and scalability as compared to computing short- and long-range interactions on all processes.

In general PP interactions represent 80% of the total execution time of non-bonded interactions computation and the rest is spent on reciprocal space computations. On modern GPU-accelerated systems, GROMACS achieves a good scaling behaviour by computing short-range interactions on the GPUs and long-range interactions on CPUs.

GROMACS is part of the PRACE Unified European Application Benchmark Suite (UEABS) [5] and has been highly optimised over the last years. In the course of this work, as expected, we did not uncover any definite optimisation potential of GROMACS by using any of the selected profiling and tracing tools.

GROMACS has been the topic of numerous investigations within PRACE, e.g. [6]-[9]. These projects focused mainly on implementing certain improvements for GROMACS as well as on detailed performance and scalability analyses with conclusions for best practices for running GROMACS on different architectures. However, with the exception of [8], where SCALASCA has been used for performance analysis, none of these white papers report on the use of dedicated profiling and tracing tools.

2. HPCToolkit

2.1. Tool Description

HPCToolkit [10] is a suite of tools for program performance analysis developed at Rice University, Houston, Texas. It is primarily designed for analysis of pure or threaded (i.e. hybrid) MPI applications, but can just as well be used for sequential or threaded applications.

For performance measurement, HPCToolkit uses statistical sampling. It supports both asynchronous sampling based on timers and hardware-counters as well as synchronous sampling of standard I/O routines (i.e. `fread`, `fwrite`, etc.) and memory management routines (i.e. `malloc`, `realloc`, `free`, etc.). Full call path unwinding together with binary analysis allows HPCToolkit to attribute performance data to program modules, source files, functions, loop constructs and individual lines of source code.

Analysis of fully-optimised binaries is possible and even recommended. For attribution of metrics to source code, the application has to be compiled so that it includes debug symbols, which is usually the only required change to the build procedure. For reasonably chosen sampling rates, HPCToolkit typically adds very low overhead (below 5%) to the program execution.

Tool Name	HPCToolkit
Website	http://hpctoolkit.org/
Main Features	• Performance measurement ○ Statistical sampling • Analysis of fully-optimised binaries ○ Low runtime overhead ○ Metrics associated to full call path • Profile viewer ○ Top-Down, Bottom-Up and Flat view ○ Rich context information (e.g. call-sites and source code display) ○ User-defined (derived) metrics • Trace viewer ○ Timeline view ○ Visualization based on functions and call path depth
Documentation	Very good and comprehensive User Manual
License	Open Source (BSD License)
Supported Platforms	Linux-x86_64, Linux-x86, Linux-Power

Table 1: HPCToolkit overview

2.2. Basic Workflow

The basic workflow for program performance analysis with HPCToolkit consists of the following four steps:

1. Building the application
2. Measuring application performance
3. Post-processing measurements
4. Measurement presentation and analysis

Depending on static or dynamic linkage being used for building the application, steps 1 and 2 need to be adapted appropriately. The instructions given in this section apply for dynamic linkage, which is in general the more typical case. For the sake of completeness, the instructions for the case of static linkage (which is typically required e.g. on Cray systems) are found in the appendix, section 8.1.1.

2.2.1. Building the application

HPCToolkit principally works without any changes to the build procedure, i.e. with fully optimised executables without debug symbols. In this case (if binary analysis works) performance metrics are still associated to function names and their call stack, but no association to source code is available in the profile viewer. For this reason, it is generally recommended to include debug symbols when compiling the application (the usual compiler switch for doing so is `-g`).[1]

2.2.2. Measuring Application Performance

For collecting performance measurements, the `hpcrun` launch wrapper has to be used, which inserts profiling code to the application at runtime via `LD_PRELOAD`. For a typical MPI application the application launch command has the following form:

```
[mpi-launcher] [mpi-args] hpcrun [hpcrun-args] [application] [application-args]
```

For instance, when measuring the performance of an MPI application `myapp` running with 256 MPI ranks, the launch command will be something like the following:

```
mpirun -n 256 hpcrun -e PAPI_TOT_CYC@4100100 -o myapp-256.m ./myapp --some-arg
```

For full documentation of `hpcrun` and its options we refer to its man page (`man hpcrun`) and the HPCToolkit user manual. In the following we briefly describe the three most important options:

`-e event[@period]`	This option specifies a so-called sampling source. A full list of available sampling sources can be queried with `hpcrun -L`. For example `-e PAPI_TOT_CYC@4100100` specifies that sampling (i.e. recording the applications stack trace) will take place roughly every 4 million CPU cycles. Multiple sampling sources may be specified, however in general not all combinations of hardware counters will be supported by the underlying hardware. HPCToolkit will report an error in such a case. Note that for synchronous event sources (such as `IO` and `MEMLEAK`) no sampling period has to be specified (all calls to I/O resp. memory management library functions will be intercepted).
`-t`	This option enables trace data collection. In addition to collecting aggregated profiling data, also the temporal behaviour of the program is recorded.
`-o [output-path]`	This option defines the name of the output directory for the collected performance data.

2.2.3. Post-Processing Measurements

Correlation of runtime performance data (i.e. dynamic call paths referring to optimised binaries) with the original source code is usually done in two steps:

First, the tool `hpcstruct` can be used in order to recover structural information from optimised binaries, such as loops or functions, which have been transformed or inlined by the compiler. This information is stored in a so-

[1] Note that for some compilers, `-g` implies turning off certain optimizations. For this reason, it is strongly recommended to pass all optimization flags after the `-g` flag, as the precedence for compiler options is usually right-to-left.

called program structure file, which can be reused for multiple analyses as long as the binary executable remains unchanged. For example, `hpcstruct ./myapp` will generate a program structure file named `myapp.hpcstruct`.

Second, `hpcprof` or `hpcprof-mpi` are used for attributing the performance measurements to source code, optionally using augmented information from the program structure file. The result of this step is a so-called experiment database. Note that `hpcprof` and `hpcprof-mpi` behave very similarly. Usage of the latter one is recommended for MPI applications. The typical command line for `hpcprof-mpi` has the following form:

```
[mpi-launcher] [mpi-args] hpcprof-mpi -S [structure file] -I [path-to-source-code]/+ -o
[ouput path] [measurements-directory]…
```

For instance, consider the following command line:

```
mpirun -n 16 hpcprof-mpi -S myapp.hpcstruct -I ~/myprog/src/+ -o experiment.db myapp-
256.m
```

This will launch `hpcprof-mpi` with 16 MPI ranks (the number of ranks for `hpcprof-mpi` may be different from the number of ranks used for running the application). Using the previously generated measurements directory `myapp-256.m` and the program structure file `myapp.hpcstruct`, an experiment database will be created, which is located in a directory named `experiment.db`. The source code of the application, searched in the directory `~/myprog/src` will be included (the suffix '`/+`' indicates a recursive search in subdirectories). Experiment databases are therefore self-contained, which is handy when moving databases between systems and in cases where the source code is frequently changed.

2.2.4. Measurement presentation and analysis

For visual analysis of generated experiment databases, HPCToolkit provides two portable (Java-based) and easily installable (i.e. unpack and run) tools, `hpcviewer` for profile analyses and `hpctraceviwer` for trace analyses. The self-contained experiment databases can easily be copied to a local workstation computer and inspected with one of these tools.

2.3. Tool Features

In this section we give a short overview of the graphical user interfaces of HPCToolkit's profile viewer `hpcviewer` and trace viewer `hpctraceviewer`, which is intended to serve as a starting point for novices. For a full reference of the tool features, we refer to the HPCToolkit user manual.[2]

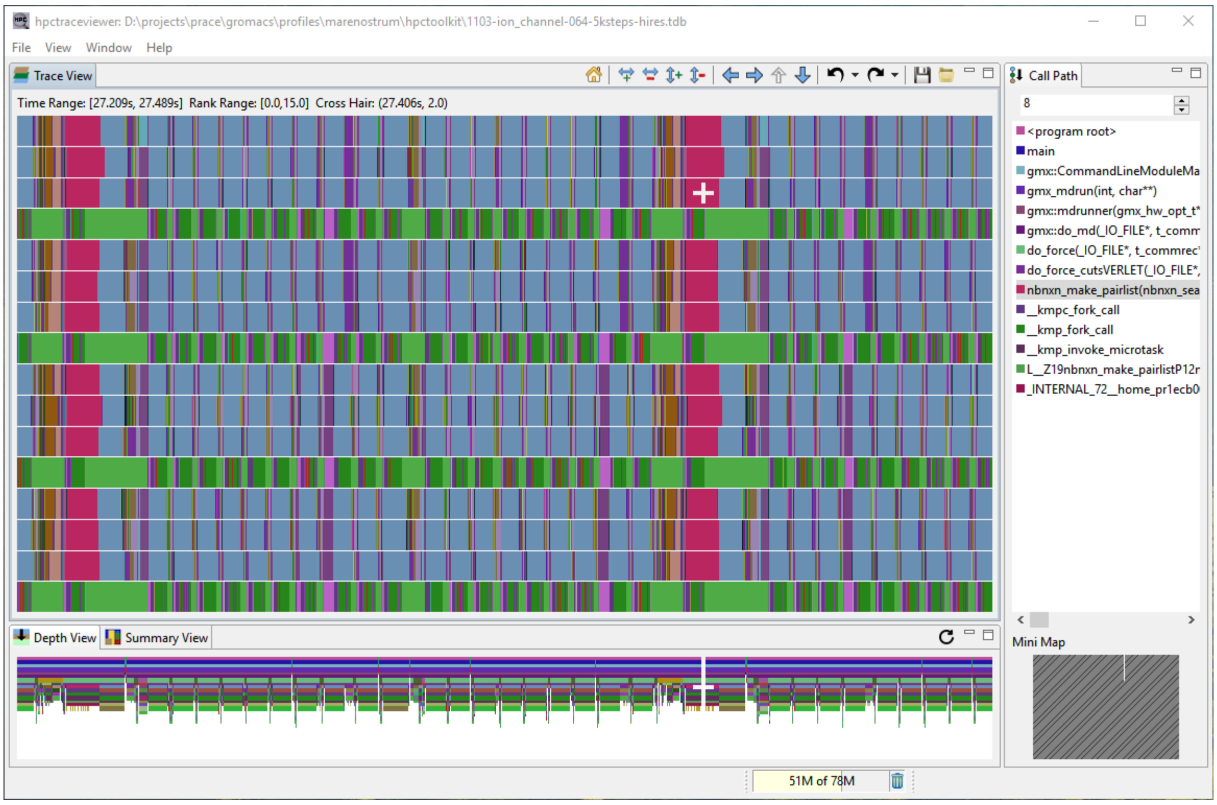
2.3.1. Trace viewer

Figure 2 shows a trace visualization of a pure MPI execution of GROMACS with 64 processes. The **Trace View** pane shows the activity of the first 16 MPI processes (*Rank Range: [0.0, 15.0]*), in a time period of 280 milliseconds (*Time Range: [27.209s, 27.489]*). Each row corresponds to one MPI process and different colours encode the different functions being executed over time, where time is flowing from left to right. The partitioning of processes in a PP and PME group as done by GROMACS (compare section 1.2), can be seen clearly from the different coloring patterns of the individual rows: Each fourth (greenish) row corresponds to a process of the PME group, while the other (bluish) rows correspond to processes of the PP group.

The level of detail of the trace visualization can be adjusted in the **Call Path** pane. Currently, the visualization shows all functions being executed at call path level 8. According to the current selection by the white cross hair in the *Trace View* pane (*Cross Hair: [27.406s, 2.0]*), the *Call Path* pane shows the full call stack of the execution thread 2.0 (i.e. MPI rank 2, thread 0), at execution time 27.406 (seconds). The **Depth View** pane visualises full call stack of the currently selected execution thread, i.e. thread 2.0, over time.

Navigation (temporal and among processes) within the trace is easily possible by either using toolbar buttons or directly by dragging a selection in one of the visualization panes.

[2] http://www.hpctoolkit.org/manual/HPCToolkit-users-manual.pdf

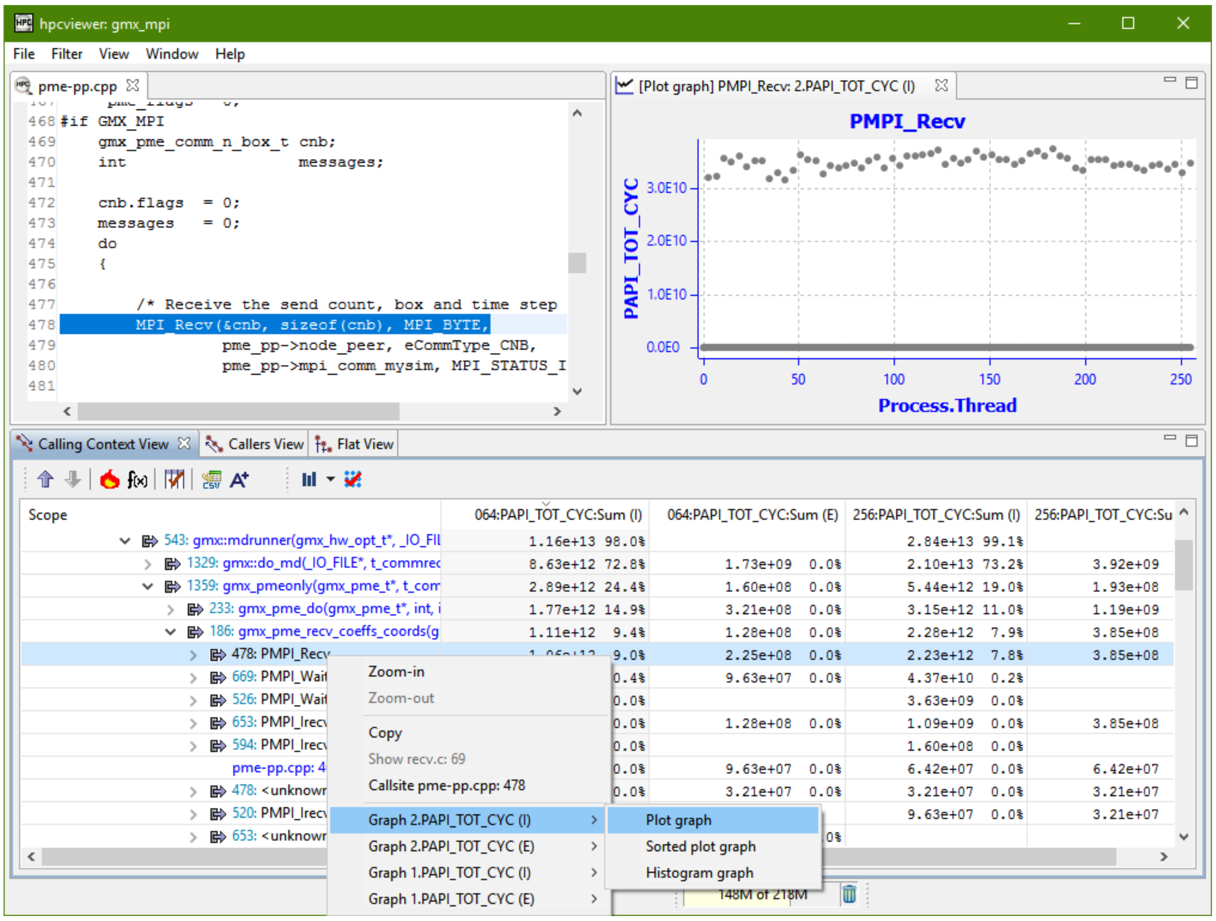

hpctraceviewer: D:\projects\prace\gromacs\profiles\marenostrum\hpctoolkit\1103-ion_channel-064-5ksteps-hires.tdb
File View Window Help
Trace View
Call Path
Depth View
Summary View
Mini Map
51M of 78M

Time Range: [27.209s, 27.489s] Rank Range: [0.0,15.0] Cross Hair: (27.406s, 2.0)

Call Path: 8

- <program root>
- main
- gmx::CommandLineModuleMa
- gmx_mdrun(int, char**)
- gmx::mdrunner(gmx_hw_opt_t*
- gmx::do_md(_IO_FILE*, t_comm
- do_force(_IO_FILE*, t_commrec
- do_force_cutsVERLET(_IO_FILE*,
- nbnxn_make_pairlist(nbnxn_sea
- __kmpc_fork_call
- __kmp_fork_call
- __kmp_invoke_microtask
- L__Z19nbnxn_make_pairlistP12r
- _INTERNAL_72__home_pr1ecb0

Figure 2: The HPCToolkit trace viewer



```
#if GMX_MPI
    gmx_pme_comm_n_box_t cnb;
    int                  messages;

    cnb.flags   = 0;
    messages    = 0;
    do
    {

        /* Receive the send count, box and time step
        MPI_Recv(&cnb, sizeof(cnb), MPI_BYTE,
                 pme_pp->node_peer, eCommType_CNB,
                 pme_pp->mpi_comm_mysim, MPI_STATUS_I

```

Scope	064:PAPI_TOT_CYC:Sum (I)	064:PAPI_TOT_CYC:Sum (E)	256:PAPI_TOT_CYC:Sum (I)	256:PAPI_TOT_CYC:Su
543: gmx::mdrunner(gmx_hw_opt_t*, _IO_FIL	1.16e+13 98.0%		2.84e+13 99.1%	
1329: gmx::do_md(_IO_FILE*, t_commrec	8.63e+12 72.8%	1.73e+09 0.0%	2.10e+13 73.2%	3.92e+09
1359: gmx_pmeonly(gmx_pme_t*, t_com	2.89e+12 24.4%	1.60e+08 0.0%	5.44e+12 19.0%	1.93e+08
233: gmx_pme_do(gmx_pme_t*, int, i	1.77e+12 14.9%	3.21e+08 0.0%	3.15e+12 11.0%	1.19e+09
186: gmx_pme_recv_coeffs_coords(g	1.11e+12 9.4%	1.28e+08 0.0%	2.28e+12 7.9%	3.85e+08
478: PMPI_Recv	1.06e+12 9.0%	2.25e+08 0.0%	2.23e+12 7.8%	3.85e+08
669: PMPI_Wait	0.4%	9.63e+07 0.0%	4.37e+10 0.2%	
526: PMPI_Wait	0.0%		3.63e+09 0.0%	
653: PMPI_Irecv	0.0%	1.28e+08 0.0%	1.09e+09 0.0%	3.85e+08
594: PMPI_Irecv	0.0%		1.60e+08 0.0%	
pme-pp.cpp: 4	0.0%	9.63e+07 0.0%	6.42e+07 0.0%	6.42e+07
478: <unknown	0.0%	3.21e+07 0.0%	3.21e+07 0.0%	3.21e+07
520: PMPI_Irecv			9.63e+07 0.0%	3.21e+07
653: <unknown		0%		

Figure 3: The HPCToolkit profile viewer

2.3.2. Profile viewer

Figure 3 shows the graphical user interface of HPCToolkit's profile viewer. The currently opened experiment database contains data of two GROMACS executions, one with 64 processes and one with 256 processes, where respectively 16 and 64 processes have been used for PME. For each of the two executions, the hardware counter for total CPU cycles (`PAPI_TOT_CYC`) has been monitored, which is essentially proportional to wall clock time.

A good starting point for most analyses is the **Calling Context View** in the lower half of the screen. Here, functions and their attributed metrics are displayed with respect to their full dynamic calling context. Looking at the first metric column, *064:PAPI_TOT_CYC:Sum (I)*, we can see that the functions `gmx::do_md` and `gmx_pmeonly` have high contributions to the total number of CPU cycles of the execution with 64 processes. Note that `gmx::do_md` is executed by processes of the PP group only, while `gmx_pmeonly` is executed by PME ranks only (compare section 1.2). In fact, the respective percentages of 72.8% and 24.4% for `gmx::do_md` and `gmx_pmeonly` quite accurately reflect the proportion of PP to PME ranks, which is 3:1 (compare also Figure 2).

Expanding the call path below the function `gmx_pmeonly`, we can see that PME ranks spend quite a significant amount of time with receiving data (i.e. `PMPI_Recv`) within the function `gmx_pme_recv_coeffs_coords`. The percentage of CPU cycles for this receive operation (9.0%) has to be seen in relation to the total number of CPU cycles coming from the processes of the PME group (25%). In terms of execution time, this means that PME ranks spend about 36% (= 4·9%) of their time with waiting and receiving data at this particular call site. Note that the function `gmx_pme_recv_coeffs_coords` can also be seen clearly in the trace view (Figure 2) as light-green areas within the execution timeline for PME processes. A possible conclusion here could be that the work load for PME ranks is too low and choosing a smaller number of PME ranks could potentially improve performance.[3]

Apart from the *Calling Context View*, also the *Callers View* and the *Flat View* are available. The **Callers View** is a bottom-up view which allows it to look upward along call paths. This view is particularly useful for functions that occur in more than one context, such as communication library routines. Finally the **Flat view** allows it to inspect performance metrics according to the static program structure, i.e. at the level of load modules, source files, functions and individual lines of source code.

In all of the views (*Calling Context View*, *Callers View* or *Flat View*), selecting a given node will open the corresponding position in the source code (for functions this is the start of their definition), if available. In order to navigate to the precise call site of a given function, the context menu can be used. In the top-left pane in Figure 3 for example, the call site of the above-mentioned critical MPI receive operation can be seen (also shown is the corresponding context menu entry, *Callsite pme-pp.cpp: 478*).

The last feature to be presented here is `hpcviewer`'s support for visualisations of different metrics among processes.[4] The top-right pane in Figure 3 shows such a visualization in the case of the execution with 256 processes. For the previously mentioned critical MPI receive operation, the total CPU cycles per process are displayed. It can be seen that the receive times for PME ranks (i.e. every fourth process) vary between (estimated) 3.2×10^{10} and 3.8×10^{10} CPU cycles. The other processes (i.e. PP ranks) do not execute this particular receive operation and thus have a contribution of 0.0 CPU cycles.

2.4. Case-Study: Analysing Scaling Overheads

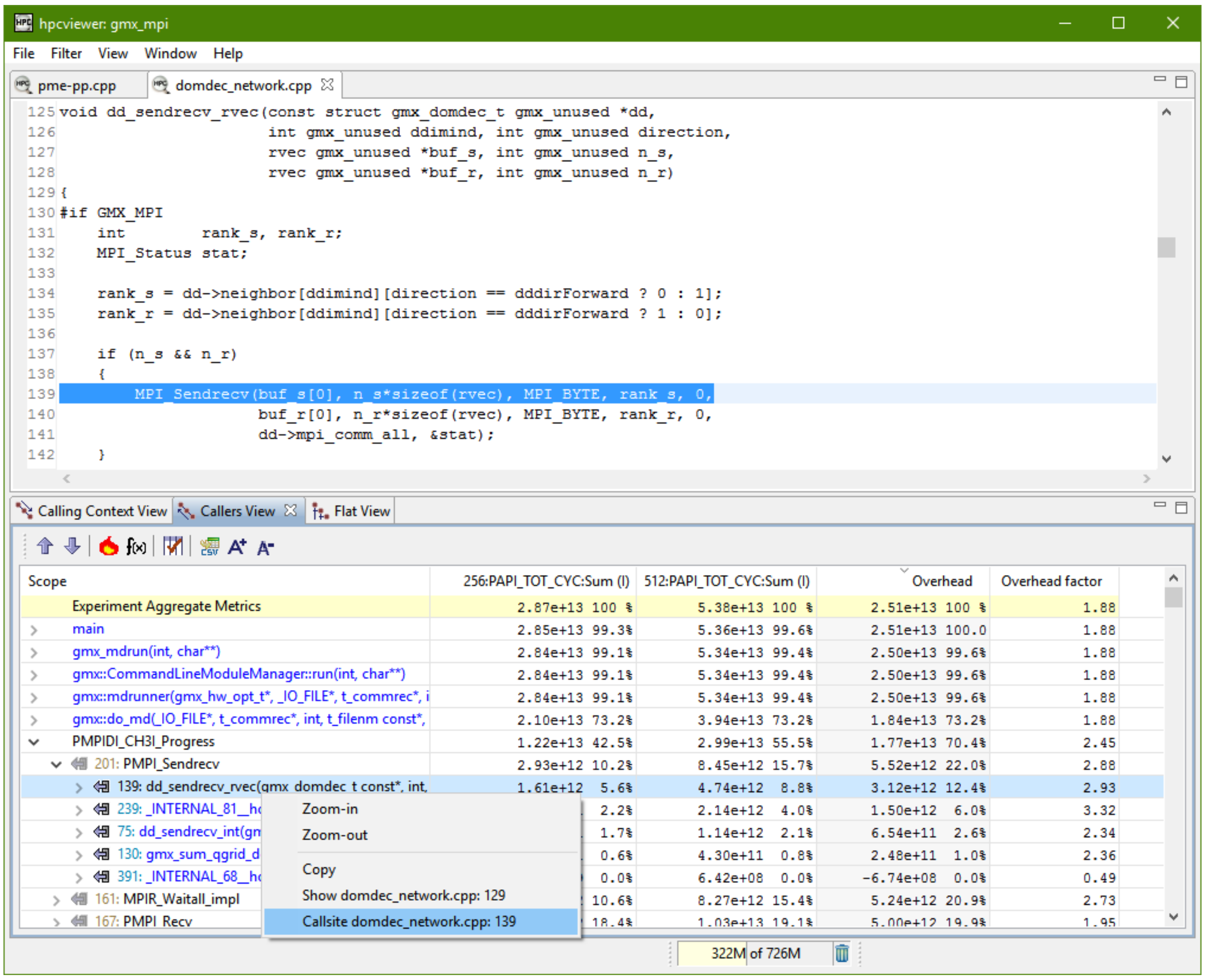
In this section we briefly demonstrate how *derived metrics*, i.e. user-defined metrics, which are based on expressions using available performance metrics, can be used for pinpointing scalability bottlenecks. For a more comprehensive and excellent discussion on the use of derived metrics for other purposes, we refer the reader to the HPCToolkit user manual, Chapter 4, *Effective Strategies for Analyzing Program Performance*.

The following analysis is based on a strong scaling experiment, i.e. two executions of GROMACS 2016 with 256 and 512 processes respectively. The underlying benchmark case is test case A for GROMACS (`ion_channel`) from the *PRACE Unified European Application Benchmark Suite* (UEABS).[5] For demonstration purposes, the number of processes is deliberately chosen quite high (scalability limit of the case) and all tuning options provided by GROMACS have been left at their default setting. For this reason, the results presented in this section are not representative for typical production runs and thus are not necessarily an indication for any inherent scalability or load balancing problem of GROMACS.

[3] It is recommended to use the GROMACS `tune_pme` tool for selecting a good balance between PP and PME ranks before running the actual simulation. This has not been done in the above case.

[4] For this to be supported, `hpcprof-mpi` (not `hpcprof`) has to be used for preparing the experiment database.

[5] http://www.prace-ri.eu/ueabs/#GROMACS



```
125 void dd_sendrecv_rvec(const struct gmx_domdec_t gmx_unused *dd,
126                       int gmx_unused ddimind, int gmx_unused direction,
127                       rvec gmx_unused *buf_s, int gmx_unused n_s,
128                       rvec gmx_unused *buf_r, int gmx_unused n_r)
129 {
130 #if GMX_MPI
131     int        rank_s, rank_r;
132     MPI_Status stat;
133
134     rank_s = dd->neighbor[ddimind][direction == dddirForward ? 0 : 1];
135     rank_r = dd->neighbor[ddimind][direction == dddirForward ? 1 : 0];
136
137     if (n_s && n_r)
138     {
139         MPI_Sendrecv(buf_s[0], n_s*sizeof(rvec), MPI_BYTE, rank_s, 0,
140                      buf_r[0], n_r*sizeof(rvec), MPI_BYTE, rank_r, 0,
141                      dd->mpi_comm_all, &stat);
142     }
```

Scope	256:PAPI_TOT_CYC:Sum (I)	512:PAPI_TOT_CYC:Sum (I)	Overhead	Overhead factor
Experiment Aggregate Metrics	2.87e+13 100 %	5.38e+13 100 %	2.51e+13 100 %	1.88
main	2.85e+13 99.3%	5.36e+13 99.6%	2.51e+13 100.0	1.88
gmx_mdrun(int, char**)	2.84e+13 99.1%	5.34e+13 99.4%	2.50e+13 99.6%	1.88
gmx::CommandLineModuleManager::run(int, char**)	2.84e+13 99.1%	5.34e+13 99.4%	2.50e+13 99.6%	1.88
gmx::mdrunner(gmx_hw_opt_t*, _IO_FILE*, t_commrec*, i	2.84e+13 99.1%	5.34e+13 99.4%	2.50e+13 99.6%	1.88
gmx::do_md(_IO_FILE*, t_commrec*, int, t_filenm const*,	2.10e+13 73.2%	3.94e+13 73.2%	1.84e+13 73.2%	1.88
PMPIDI_CH3I_Progress	1.22e+13 42.5%	2.99e+13 55.5%	1.77e+13 70.4%	2.45
201: PMPI_Sendrecv	2.93e+12 10.2%	8.45e+12 15.7%	5.52e+12 22.0%	2.88
139: dd_sendrecv_rvec(gmx_domdec_t const*, int,	1.61e+12 5.6%	4.74e+12 8.8%	3.12e+12 12.4%	2.93
239: _INTERNAL_81__hc	2.2%	2.14e+12 4.0%	1.50e+12 6.0%	3.32
75: dd_sendrecv_int(gn	1.7%	1.14e+12 2.1%	6.54e+11 2.6%	2.34
130: gmx_sum_qgrid_d	0.6%	4.30e+11 0.8%	2.48e+11 1.0%	2.36
391: _INTERNAL_68__hc	0.0%	6.42e+08 0.0%	-6.74e+08 0.0%	0.49
161: MPIR_Waitall_impl	10.6%	8.27e+12 15.4%	5.24e+12 20.9%	2.73
167: PMPI_Recv	18.4%	1.03e+13 19.1%	5.00e+12 19.9%	1.95

Figure 4: `hpcviewer`: Analysing scaling losses with derived metrics.

As described in section 2.2.2, for both GROMACS executions, we collected performance measurements based on the hardware counter for total CPU cycles, i.e. `PAPI_TOT_CYC`, with `hpcrun`. The two generated measurement directories, named `m256` and `m512` respectively, were combined into one single experiment database with `hpcprof-mpi`, as follows (for brevity, the MPI launcher and arguments for `-I` and `-S` are omitted here; for a description see section 2.2.3):

```
… hpcrpof-mpi -I … -S … -o experiment.db m256 m512
```

After opening the generated experiment database (`experiment.db`) in `hpcviewer`, the following steps have been carried out:

1. For better overview, *exclusive* metrics (*…PAPI_TOT_CYC:Sum (E)*), have been hidden (Toolbar button *Hide/Show metrics*,) and the remaining *inclusive* metrics have been renamed to *256:PAPI_TOT_CYC:Sum (I)* and *512:PAPI_TOT_CYC:Sum (I)* respectively (*View → Show metric properties*).[6]
2. For quantifying scaling losses and attributing them to individual functions, we are using two derived metrics: *Overhead* is defined as the difference of the metrics *512:PAPI_TOT_CYC:Sum (I)* and *256:PAPI_TOT_CYC:Sum (I)* and *Overhead factor* as their quotient, i.e. *Overhead* quantifies the absolute increase of CPU cycles spent by each function while *Overhead factor* quantifies the relative increase,

[6] An *exclusive* metric for a scope refers to the quantity of the metric measured for that scope alone; an *inclusive* metric for a scope represents the value measured for that scope as well as costs incurred by any functions it calls. In `hpcviewer`, inclusive metric columns are marked with *(I)* and exclusive metric columns are marked with *(E)*.

comparing the execution with 512 processes to that with 256 processes.[7] For a short description of the user interface for defining a derived metric, see the appendix, section 8.1.2.

3. For pinpointing the call sites of MPI procedures which incur high scaling overhead, we are using the *Callers View*. By clicking the column header of our derived *Overhead* metric, we sort functions according to this metric in descending order.

As can be seen in Figure 4, in the first row of the *Callers View* (*Experiment Aggregate Metrics*), by going from 256 to 512 processes, the total number of CPU cycles increased from $2.87 \cdot 10^{13}$ to $5.38 \cdot 10^{13}$. The absolute increase, which we call overhead or *scaling loss*, is $2.51 \cdot 10^{13}$ CPU cycles and the relative increase is 88% (*Overhead factor* 1.88).

A significant portion of the total scaling loss, namely 70.4%, is caused by the routine `PMPIDI_CH3I_Progress`, one of MPI's core routines executed during idleness. Expanding descendant nodes, it can be seen much of this extra waiting time (22.0% out of the total 70.4%) is induced by `PMPI_Sendrecv`, where in turn 12.5% (out of 22.0%) are due to the GROMACS function `dd_sendrecv_rvec`.

Apart from `dd_sendrecv_rvec`, five more functions with high scaling overhead have been identified in a similar way. By looking up these functions in the *Flat View*, an overall statement on their individual contributions to the total scaling loss can be made. Moreover, by expanding descendant nodes, overhead-causing MPI routines can be accurately quantified, as shown in Table 2. Note that in order to conclude on overhead contributions in terms of run-time, the percentages given in Table 2 have to be seen in relation to the proportion of PP and PME ranks, which is 3:1. This means for example that the routine `gmx_pme_recv_coeffs_coords`, executed by PME ranks (i.e. each fourth rank) only, contributes to effectively 40% (= 4 · 10.0%) of the extra execution time of the PME ranks.

Function	**Group**	**Total**	**Sendrecv**	**Recv**	**Waitall**	**Alltoall**	**Other**
`gmx_pme_recv_f`	PP	**20,4%**		20,4%			
`dd_sendrecv2_rvec`	PP	**18,5%**			18,3%		0,2%
`dd_sendrecv_rvec`	PP	**16,1%**	14,6%	1,5%			
`gmx_pme_recv_coeffs_coords`	PME	**10,0%**		9,7%	0,3%		
`fft5d_execute`	PME	**7,3%**				7,3%	
`pme_dd_sendrecv`	PME	**7,1%**	6,6%	0,5%			

Table 2: Relative contributions to the total scaling loss by functions and causing MPI operations (strong scaling from 256 to 512 processes).

2.5. Concluding Comments on HPCToolkit

Positive

- Good scalability: Performance monitoring can be restricted to a subset of processes; no problems with monitoring 1000+ processes.
- Very good documentation: Extensive user's manual, man pages and command line help.
- Painless installation procedure:
    - GUI tools (`hpcviewer`, `hpctraceviewer`): Java-based platform-independent (extract and run).
    - Core tools (`hpcrun`, `hpcprof`, etc.): Well documented build from source (all required external libraries contained in tar-ball).
- Good support: Problem reports are answered quickly on HPCToolkit mailing list.

Tool limitations

- MPI communication analysis (senders / receivers / sizes of messages etc.) is not supported as it does not lie in the focus of HPCToolkit.

Opportunities for improvements:

- `hpcviewer`:
    - Derived metrics are lost when closing an experiment database and need to be re-defined after re-opening it.

[7] The total sum of CPU cycles of all processes is essentially a measure total computational work or resource usage. Hence for ideal strong scaling, the metrics *256:PAPI_TOT_CYC:Sum (I)* and *512:PAPI_TOT_CYC:Sum (I)* should be equal.

    - Flat View: Search feature for faster lookup of a certain function of interest in the *Flat View* would be helpful.
- `hpctraceviewer`:
    - A possibility to highlight the execution of a certain function of interest irrespective of the call-stack level at which it occurs would be beneficial.
- Temporal filtering of samples (e.g. delayed start of performance measurement) is currently only possible using HPCToolkit's measurement API (requires changing the source-code and re-compilation).

3. Intel Trace Analyzer and Collector

3.1. Tool Description

Intel Trace Analyzer and Collector (ITAC) [11], formerly known as Vampirtrace, is a commercial tool for analysing MPI communication developed by the Intel Corporation. The tool focuses mainly on MPI communication which includes the tracing and analysing of MPI function calls and MPI messages being transferred. Complementary to ITAC, the profiling and tracing tool Intel VTune can be used for intra-node performance analysis. Although these two tools are most sensibly used in combination, this section is dedicated to ITAC only, as it appears to be not as widely known as Intel VTune.

ITAC intercepts MPI function calls in order to collect performance data. Besides Intel MPI, ITAC supports all major MPICH2-based MPI implementations.

The focus is primarily on analysing how the application communicates via MPI. Analysing other parts of the application is not covered at all by ITAC. For that, Intel recommends Intel VTune, which is a "regular" profiler.

Besides MPI communication pattern analysis, ITAC has two other notable features. The first one is the *idealization* of a profile run, where it tries to simulate an ideal MPI System with zero latency and infinite bandwidth. By comparing these idealised profiles with the original ones, one can estimate if performance problems are due to the underlying MPI system and/or hardware.

The second feature is MPI correctness checking. Here, ITAC checks every MPI Function call at runtime and tries to detect programming errors like buffer overflows, incorrect data types, possible deadlocks and data corruption.

Tool Name	Intel Trace Analyzer and Collector (ITAC)
Website	https://software.intel.com/en-us/intel-trace-analyzer
Main Features	• MPI Communication Behavior Analysis • Supports C, C++, Fortran ○ Performance measurement through instrumentation or MPI call interception ○ filter by event type and time • Event viewer ○ global, all ranks, all Messages ○ color MPI Messages ○ show messages sizes, send/receive code location • Supports Intel Xeon Phi • Supports crashing MPI Applications • MPI Correctness Checking
Documentation	Very detailed, also shorter tutorials available
License	Commercial (starting at $2,949)
Supported Platforms	Intel processors (MPICH-2 based MPI system required)

Table 3: ITAC overview

3.2. Basic Workflow

1. Ensure that Intel MPI and ITAC are correctly installed and that the environment is set up.

```
source [path-to-itac]/bin/itacvars.sh
```

2. (Optional) To get code location support, compile your program with the Intel compiler and specify the flag `-trace`.

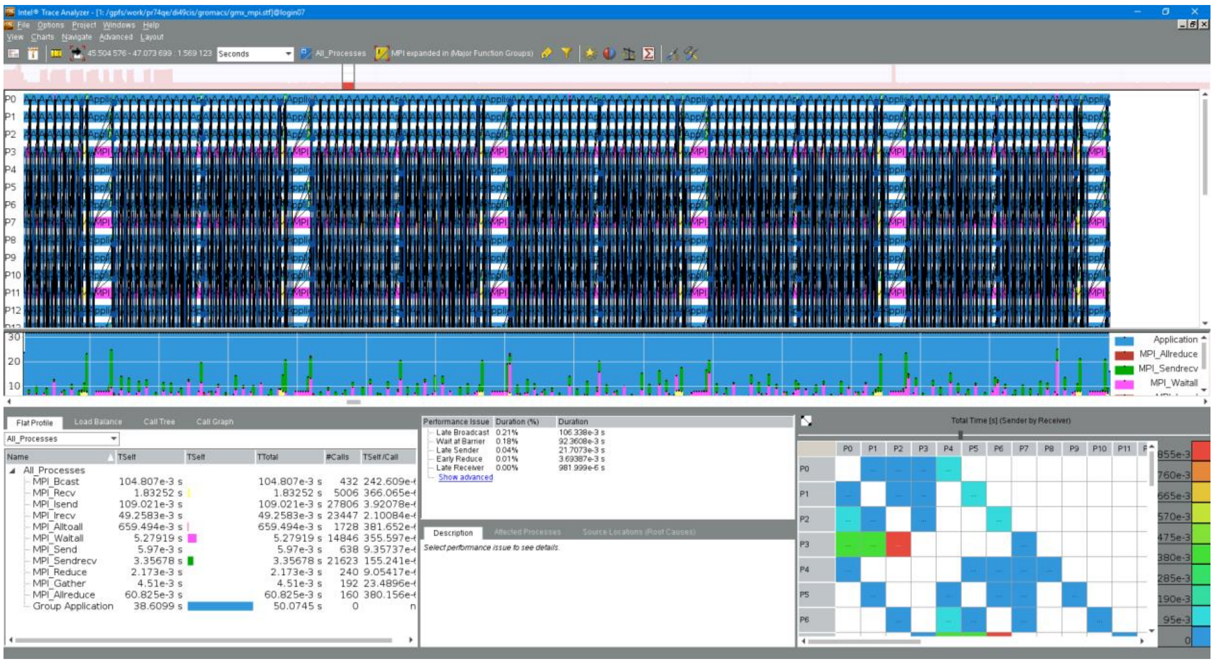
```
mpiicc -trace myApp.c
```

3. For launching the application, add the flag `-trace` to the MPI launcher `mpiexec`:

```
mpiexec -trace -n 4 ./myApp
```

4. After successful execution, multiple `*.stf` files have been created, which can be transferred to a local machine and can be opened via `traceanalyzer` (the ITAC GUI), see also Figure 5:

```
traceanalyzer myApp.stf &
```



Name	TSelf	TSelf	TTotal	#Calls	TSelf /Call
All_Processes					
MPI_Bcast	104.807e-3 s		104.807e-3 s	432	242.609e-(
MPI_Recv	1.83252 s		1.83252 s	5006	366.065e-(
MPI_Isend	109.021e-3 s		109.021e-3 s	27806	3.92078e-(
MPI_Irecv	49.2583e-3 s		49.2583e-3 s	23447	2.10084e-(
MPI_Alltoall	659.494e-3 s		659.494e-3 s	1728	381.652e-(
MPI_Waitall	5.27919 s		5.27919 s	14846	355.597e-(
MPI_Send	5.97e-3 s		5.97e-3 s	638	9.35737e-(
MPI_Sendrecv	3.35678 s		3.35678 s	21623	155.241e-(
MPI_Reduce	2.173e-3 s		2.173e-3 s	240	9.05417e-(
MPI_Gather	4.51e-3 s		4.51e-3 s	192	23.4896e-(
MPI_Allreduce	60.825e-3 s		60.825e-3 s	160	380.156e-(
Group Application	38.6099 s		50.0745 s	0	n

Performance Issue	Duration (%)	Duration
Late Broadcast	0.21%	106.338e-3 s
Wait at Barrier	0.18%	92.3608e-3 s
Late Sender	0.04%	21.7073e-3 s
Early Reduce	0.01%	3.69387e-3 s
Late Receiver	0.00%	981.999e-6 s
Show advanced		

Select performance issue to see details.

Figure 5: The Intel Trace Analyzer with several views enabled: Event Timeline, Message Quantity, Message Profile and Performance Assistant.

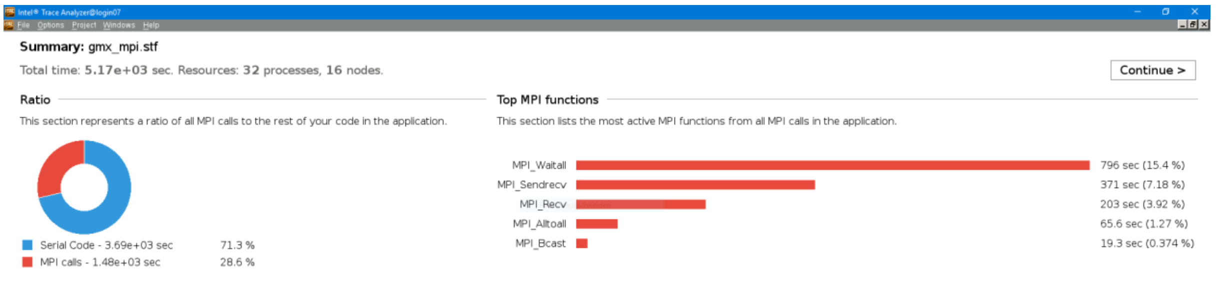
3.3. Tool Features

In the following, we present the core features of the Intel Trace Analyzer (the ITAC GUI). When opening a trace file, a **Summary Page** as seen in Figure 6 is shown which gives a compact overview of the runtime, showing the proportion spent in MPI calls and ranking the most consuming MPI functions. Knowing this, one can figure out if MPI itself is actually an issue, and if yes, where to start analysing more deeply.

The **Event Timeline view** – see Figure 7 – shows the timeline and all MPI ranks. Time spent in application code is colored blue. Time spent in MPI Functions is shown in other colors, which can be defined by the user. The MPI messages are represented as thin black lines. In this view, one can observe the dependencies between the ranks when communicating and thereby find out who is waiting for whom.

The **Message Profile view** – see Figure 8 – shows a matrix of all MPI ranks in sending and receiving role. Displayed is the amount of data (Bytes) transferred for each rank. The content is colored like a heat map, where red indicates more data than blue. Here one can identify imbalances where certain nodes must transfer more data than others and therefore might slow down the whole system.

Individual messages shown in the Event Timeline can be inspected in a **Message Detail view** as shown in Figure 9. It uncovers information of the message such as sender and receiver ranks, message size and involved MPI functions. If enabled, ITAC can even show the source code location of the participating MPI functions and the call stacks at the moment of their invocation.



Summary: gmx_mpi.stf

Total time: **5.17e+03** sec. Resources: **32** processes, **16** nodes.

Ratio

This section represents a ratio of all MPI calls to the rest of your code in the application.

🟦 Serial Code - 3.69e+03 sec	71.3 %
🟥 MPI calls - 1.48e+03 sec	28.6 %

Top MPI functions

This section lists the most active MPI functions from all MPI calls in the application.

MPI_Waitall	796 sec (15.4 %)
MPI_Sendrecv	371 sec (7.18 %)
MPI_Recv	203 sec (3.92 %)
MPI_Alltoall	65.6 sec (1.27 %)
MPI_Bcast	19.3 sec (0.374 %)

Figure 6: Intel Trace Analyzer - Summary page

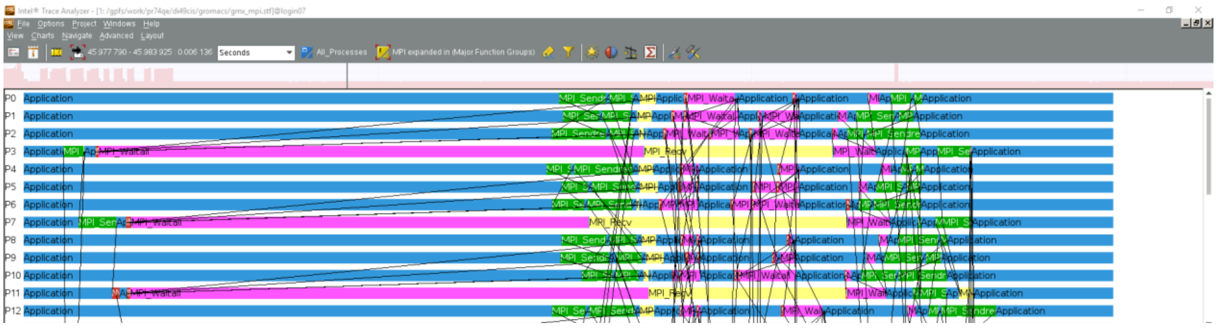


Figure 7: Intel Trace Analyzer - Event Timeline view

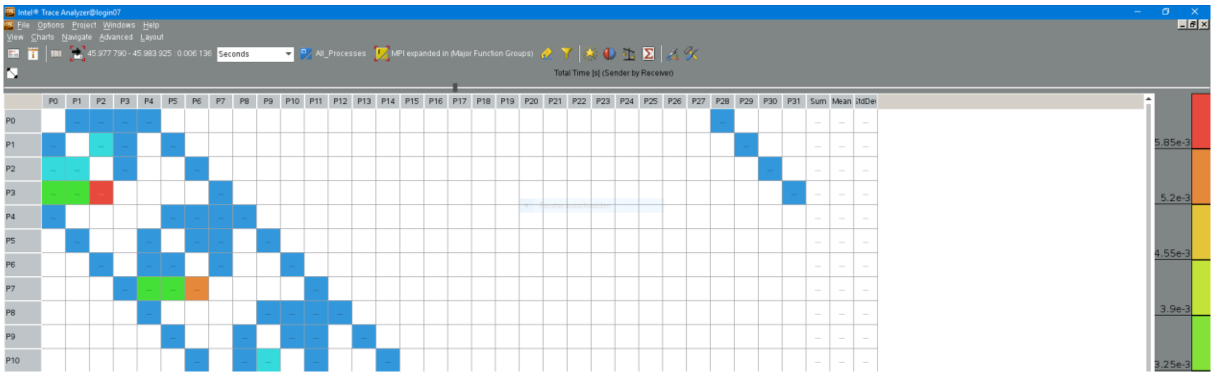
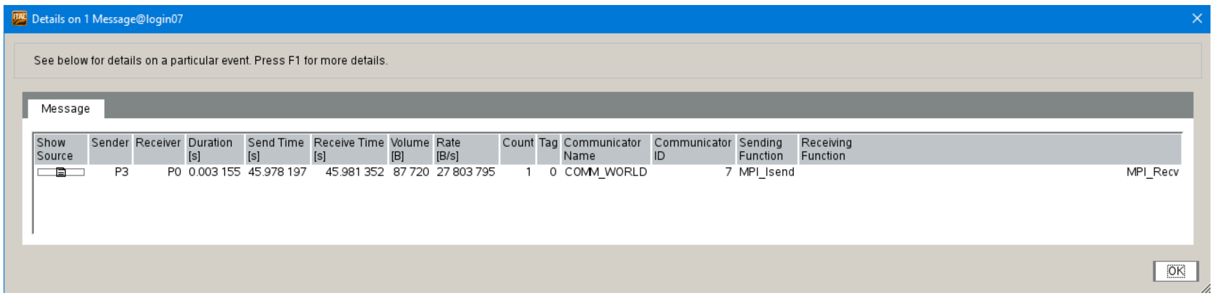


Figure 8: Intel Trace Analyzer - Message Profile view



Details on 1 Message@login07

See below for details on a particular event. Press F1 for more details.

Message

Show Source	Sender	Receiver	Duration [s]	Send Time [s]	Receive Time [s]	Volume [B]	Rate [B/s]	Count	Tag	Communicator Name	Communicator ID	Sending Function	Receiving Function
	P3	P0	0.003 155	45.978 197	45.981 352	87 720	27 803 795	1	0	COMM_WORLD	7	MPI_Isend	MPI_Recv

Figure 9: Intel Trace Analyzer - Message Detail view

3.4. Case Study: Analysing Late Senders and Late Receivers

Collecting Performance data for later Analysis

In this Case Study, we use Intel Trace Analyzer and Collector Version 9.1.2 with Intel MPI Version 5.1.3 and ICC Version 16.0.4. Recompiling the program using the Intel compiler with the `-trace` flag enables collecting performance data including code locations.

Using ITAC User Interface to analyse the communication

Enabling the *Event* view, one can see every MPI process in the system and the messages sent over time, represented as thin lines. Zooming in on the time scale, the program iterations become visible. Those iterations consist of a computing phase without any MPI communication and a communication phase, where most of the execution time is spent inside MPI routines.

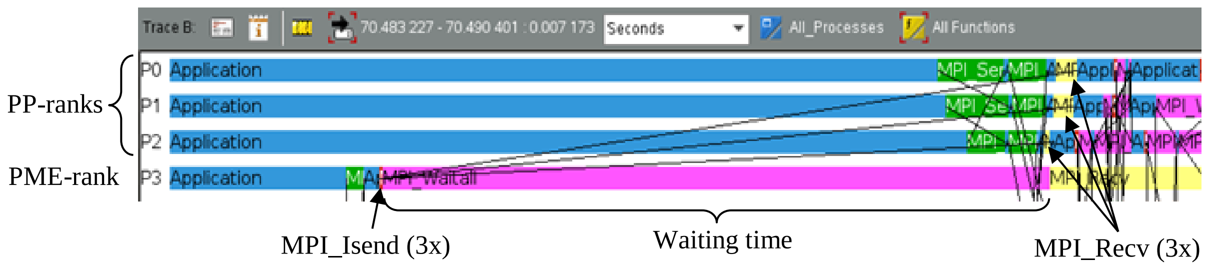


Figure 10: A late receiver problem: Three PP-ranks receive messages from a PME-rank lately causing waiting time for the PME rank.

Figure 10 shows the beginning of a communication phase. MPI processes of the PME group (PME ranks) send data to processes of PP group (PP ranks) – for an explanation on PME and PP ranks compare Section 1.2. Before the PP ranks can receive the data, some computation and PP-internal communication has to be done. In the meantime, the sending PME rank is waiting for the PP ranks to receive the data. This situation is also called “late receiver”, where data is available, but not received yet. In general, late receiver problems can sometimes be mitigated by receiving data earlier in time with non-blocking receive operations, which can potentially improve the overlap of communication and computation. However in this particular situation, doing so just would shift the PME-rank’s waiting time from `MPI_Waitall` to the subsequent `MPI_Recv` operation. Realistically, the root cause of the problem here is just the suboptimal proportion of PP and PME ranks.

3.5. Concluding Comments on Intel Trace Analyzer and Collector

Positive

- ITAC can be used even without recompiling with the original binary. Although source code location is not supported in this mode, ITAC still can show the MPI communication pattern of the application.
- The Summary view in addition to the *Performance Assistant* supports even novices in the field by identifying problematic areas (Late Sender/Late Receiver) for the MPI communication.
- ITAC has a very low performance overhead (2%, as measured on SuperMUC, when running GROMACS with 256 processes)

Tool limitations

- The tool focuses only on MPI communication. It is not possible to get information like CPU load or where the function hotspots are (ITAC’s companion Intel VTune can be used for this purpose).

4. Extrae and Paraver

4.1. Tool Description

Extrae [12] and Paraver [13] are two tools for performance analysis developed by the Barcelona Supercomputer Center (BSC), and constitute the core tools in their approach for performance analysis. There are also very useful complementary tools such as Dimemas, Clustering, Tracking and Folding.

The philosophy behind Paraver is that visual inspections should be the base – or at least the first step – to capture different program characteristics and behavior. The tool’s main focus is therefore to support the user in recognising patterns and getting global overview. These visual inspections are supposed to reveal hot-spots where to improve the code. However, since these views may be misleading and may not reveal sufficient detailed information, Paraver is complemented with features to display detailed numbers and statistics for the whole execution as well as for hot-spots.

The designers of Paraver used a modular structure to achieve their goal for high flexibility. A requirement was that Paraver should be independent of programming models and also the metrics to present. For this reason, a generic trace file format is used, allowing Paraver to easily adopt to newly defined metrics. Further, the metrics can be combined as desired by the experienced user giving enormous opportunities to display the information as desired. This is carried out by the configuration files (`.cfg`). For the most typical basic performance analyses (IPC, cache misses etc.), the Paraver tool provides predefined configuration files. For advanced users the flexibility of Paraver enables the possibility to create new configuration files.

For code instrumentation Extrae is most often used. Extrae is a dynamic instrumentation package to trace programs compiled and run with various programming models. Important examples include pthreads, OpenMP, MPI, CUDA, and combinations of these. Extrae generates trace files that can be visualised with Paraver and provides its own API (`extrae.xml`) to allow the user to manually instrument their application. In the file `extrae.xml`, it also possible to easily choose hardware counters using the PAPI library. The output from an execution is a set of three textual files containing the application activity (`.prv`), the labels associated to the numerical values (`.pcf`) and the resource usage (`.row`). There are also alternatives to capture trace-data (https://tools.bsc.es/tools_manuals) for experienced users.

Tool Name	Extrae and Paraver
Website	https://tools.bsc.es
Main Features	• Paraver is not tied to any programming model. Thus MPI, OpenMP, MPI + OpenMP, OpenMP + Cuda are handled. • Features: ○ Concurrent comparative analysis of several traces ○ Customizable semantics of the visualised information. ○ Cooperative work, sharing views of the tracefile ○ Building of derived metrics • Also very detailed ○ The instructions per cycle executed by each thread. ○ Cache misses by each tread ○ Average ready task queue length. ○ The instructions per cycle executed by each thread ○ The load balance of the different parallel loops
Documentation	Documentation is available at https://tools.bsc.es/tools_manuals and e.g. https://pop-coe.eu/further-information/learning-material
License	Open Source (LGPL)
Supported Platforms	x86 CPU, NVidia GPU, Intel Xeon Phi, ARM

Table 4: Extrae and Paraver Overview

4.2. Basic Workflow

1. Ensure that the files `extrae.xml` and `trace.sh` are in your working directory. For different analysis types (MPI OpenMP, GPUs, etc.), there are predefined versions of `extrae.xml` which normally do not need to be modified. In `trace.sh` you should specify the correct library to use. Normally the selection of one of the default libraries is sufficient.
2. When invoking the executable, the `trace.sh` script must be added as in the following example:

```
mpirun ./trace.sh gmx_mpi [parameters]
```

3. After successful execution, three files (`.prv`, `.pcf`, `.row`) will have been created. These files should be copied to your local machine. Of course, it is possible use Paraver on the login node on the cluster, but it is not recommended.
4. Start Paraver and load the trace.
5. To visualise the data a certain configuration file (`.cfg`) is selected.

4.3. Tool Features

Below four examples are given. The first three present basic views of Paraver. A horizontal timeline represents the behavior of the application along time for certain processes (vertical). For regular problems, *vertical* lines are often wanted, meaning that the execution is well-balanced with negligible synchronization problems. In the following examples, GROMACS is used as an application and the number of cores chosen is eight. Moreover, the three views have been synchronised, each of them showing the same short time period of 4 milliseconds.

Example 1: MPI Calls

In Figure 11, MPI calls for this time period are presented. The different colours display the specific MPI call used. For example, yellow stands for `MPI_Broadcast`, green for `MPI_All_reduce`, brown for `MPI_Irecv`, and so forth. Also in the figure, communications lines are shown using thin yellow lines. The light blue parts display the computation parts of the execution. Paraver offers an information panel showing the value of the colours.

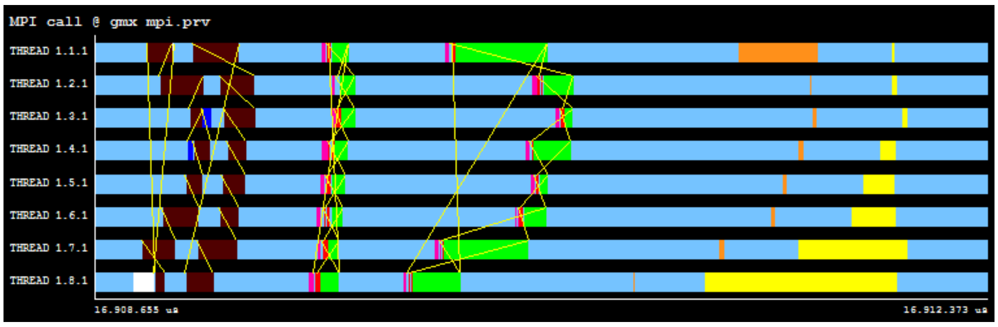


Figure 11: Analysing MPI calls with Paraver

Example 2: Analysing MFLOPs

In Figure 12, the number of MFLOPs (million floating point operations per second) are shown for the corresponding period. The color scale ranges from light green (low MFLOPs – potentially problematic) to dark blue (high MFLOPs – good). Also here an information panel is available that shows the exact range of MFLOPs-values for each color. Note how well regions with low MFLOPs align with regions corresponding to MPI operations (compare Figure 11).

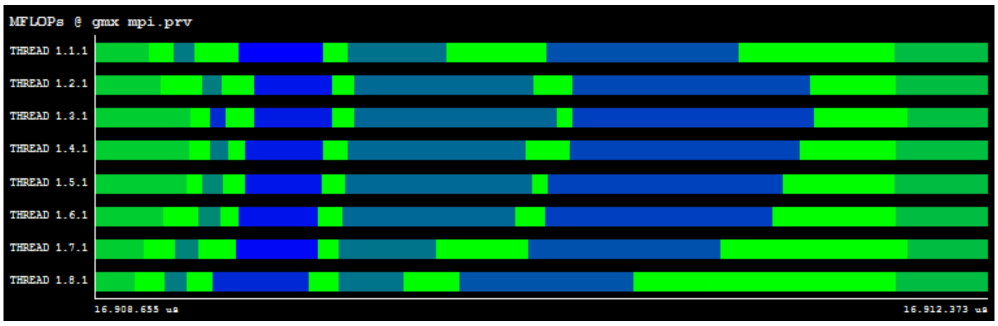


Figure 12: Analysing MFLOPS with Paraver

Example 3: Instructions per cycle (IPC)

A very important and useful view is the study of the number of *instructions per cycle* (IPC), see Figure 13. As above, the color scale again ranges from light green (low IPC – potentially problematic) to dark blue (high IPC – good). In addition, the black areas correspond to MPI Communications (compare Figure 11).

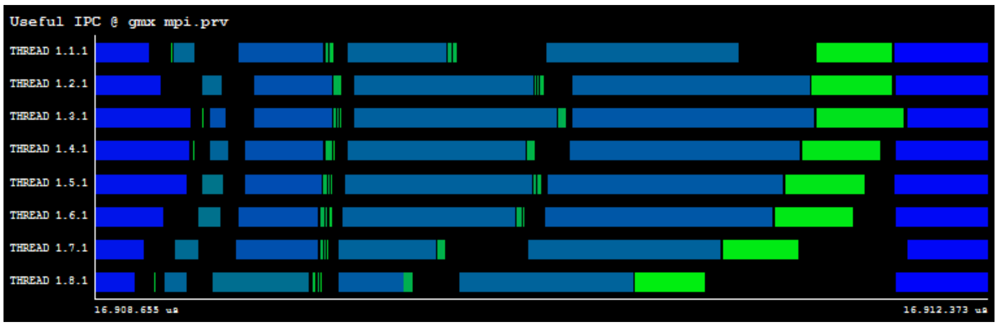


Figure 13: Analysing instructions per cycle (IPC) with Paraver

Example 4: Histogram of Instructions per cycle (IPC)

The timelines view may not always give the whole picture, and it is often useful to complement it with a study of certain metrics in a histogram. For example, Figure 14 shows for each process a histogram for IPC. In this representation, each row corresponds to one process and the x-axis corresponds to IPC (left = low; right = high). Again the color scale ranges from light green (low relative execution time with this particular IPC) to dark blue (high relative execution time with this particular IPC). Thus for highest efficiency with respect to IPC, the rightmost parts should be as close to dark blue as possible.

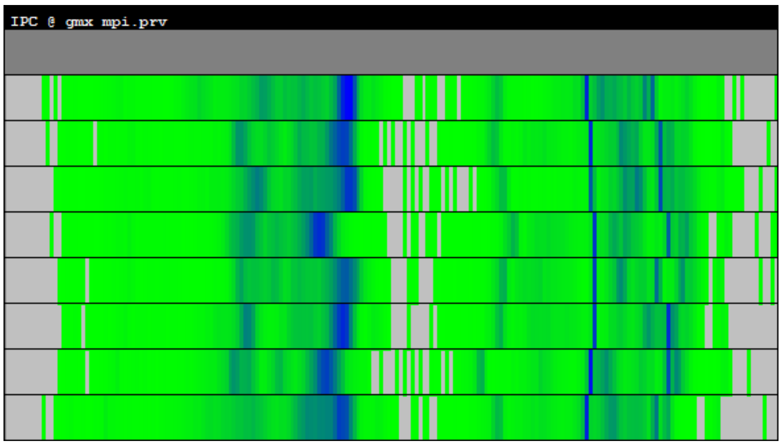


Figure 14: Paraver showing a histogram for instructions per cycle

4.4. Case Study – GROMACS

Execution with 16 cores - Using 12 PP ranks and 4 PME Ranks

Figure 15 shows a Paraver analysis of a GROMACS execution with 16 cores on the `ion_channel` test case. The figure gives a very good overview of the program's MPI communications (upper part) and its corresponding performance in terms of instruction per cycles (lower part).

The upper part of the figure shows the MPI Communication. It can be immediately seen that twelve of the processes (PP ranks) have completely different communication patterns than the other four (PME ranks). (See Section 1.2 for description of PP and PME processes)

The lower figure shows that the twelve PP ranks often have less IPC (light blue), than the four PME ranks (dark blue). However, black regions (i.e. time spent in MPI operations) occur more frequently for PME ranks, which might be an indication that the balancing between PP and PME ranks is not optimal and PME ranks are under-utilised. The good vertical alignment of computation phases indicates that the problem is well synchronised.

It can be seen that during the initialization phase, the four PME processes that compute do that with quite low IPC (light green) while PP ranks are waiting in `MPI_Irecv` (white).

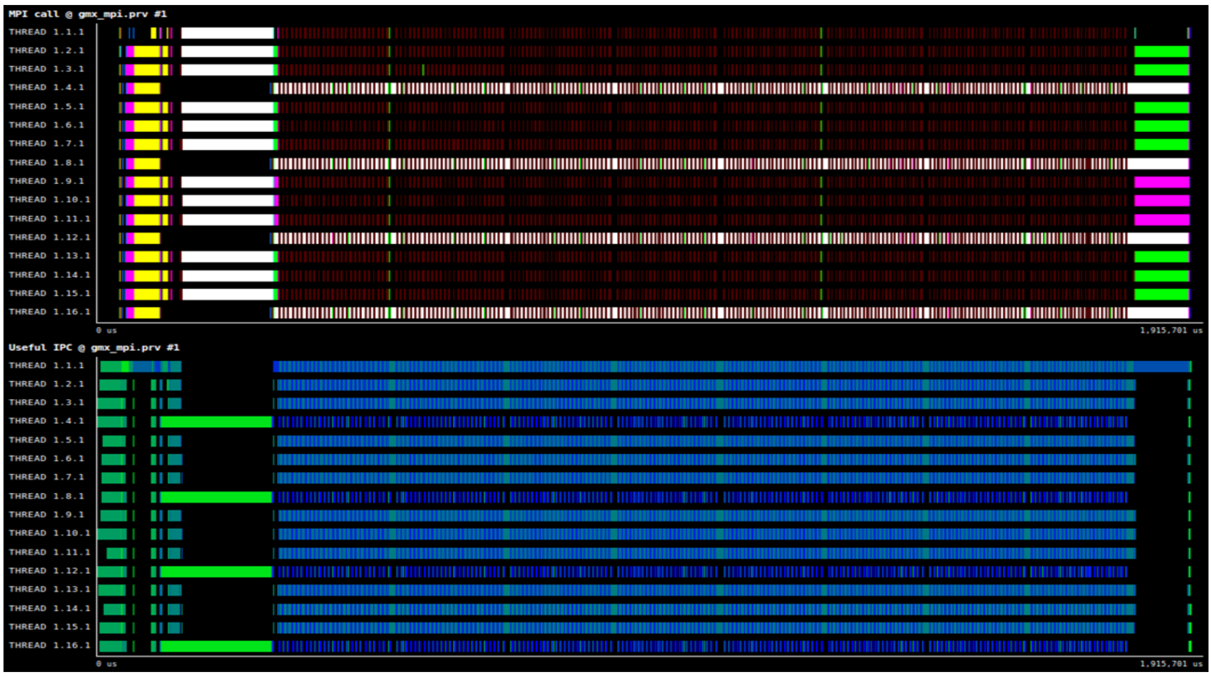


Figure 15: Paraver visualising MPI routines (upper part) and performance in terms of instructions per cycle (lower part) of a GROMACS execution with 16 MPI ranks.

Execution with 8 cores - Using 4 PP ranks and 4 PME Ranks

Here, we show differences of the parallel behaviour when increasing the number of PME Ranks. Figure 16 shows the significant difference of MPI Communication Pattern (upper part) and also the different IPC (lower part) between the set of nodes. In the figure a zoomed-in part is shown. This constitutes a very good example of how to study the behaviours in more detail. We can see that GROMACS involves a lot of MPI communications of various kinds. The view below shows the IPC varies strongly with MPI communication patterns.

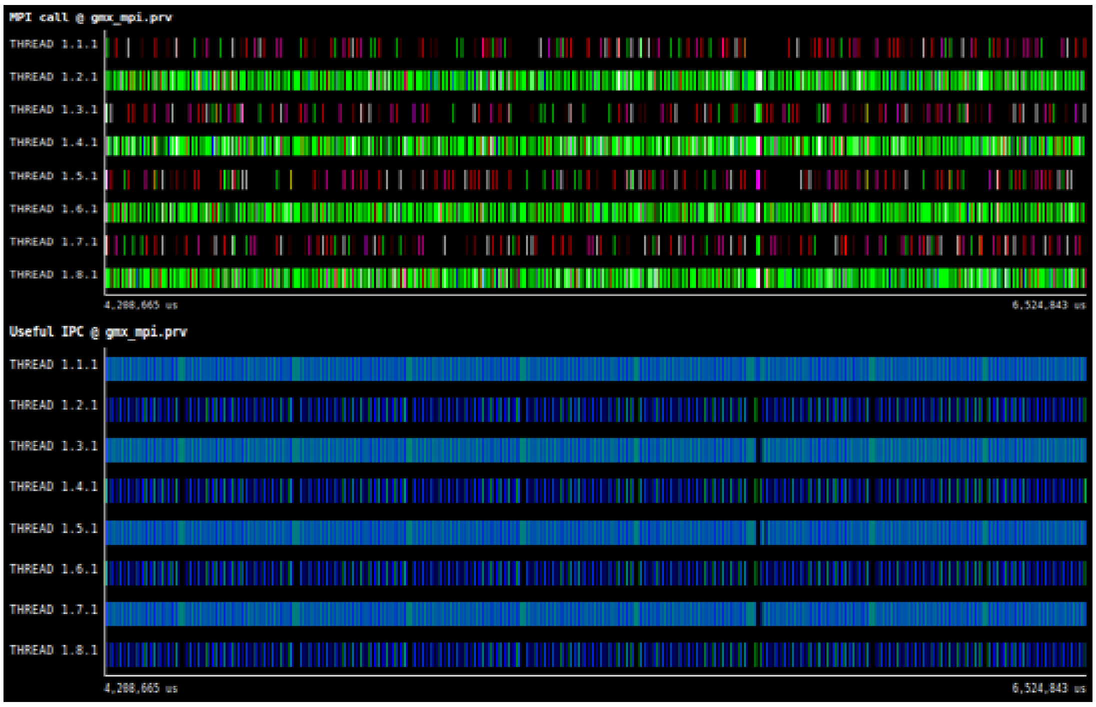


Figure 16: A Paraver visualisation of a GROMACS execution, where the number of PME ranks has been chosen too high (every 2nd rank is PME).

4.5. Concluding Comments on Extrae and Paraver

Positive

- Paraver offers an enormous amount of possibilities to study the performance in detail.
- The capturing of trace data has a low overhead.
- Large and multiple traces are quite easy to handle.
- It is very easy to zoom, synchronise windows, and to compare different traces.
- A tutorial with examples (traces, and `.cfg`) is included in the Paraver package.
- Very good support; Problem reports are answered quickly.

Opportunities for improvements

- It is not obvious how to organise the configuration files. Since everyone can create such files, there are many of them out there. Very often equal files do have different names.
- It is not so easy to interpret the information from the histograms. The metrics for the different dimensions can be modified, but is not obvious how interpret them.
- 3D-visualizations as alternative to 2D colour-coded visualizations might be helpful.
- The learning curve for the more advance features from Extrae and Paraver is quite steep, such as developing own advanced configuration files.
- Due to Paraver's extreme flexibility, it could be difficult to find your own strategy for analyzing a program (except for the very basic analysis which is included in the tutorial).
- Manuals are of limited help for unexperienced users. Paraver's manual is outdated.

5. SCALASCA

5.1. Tool Description

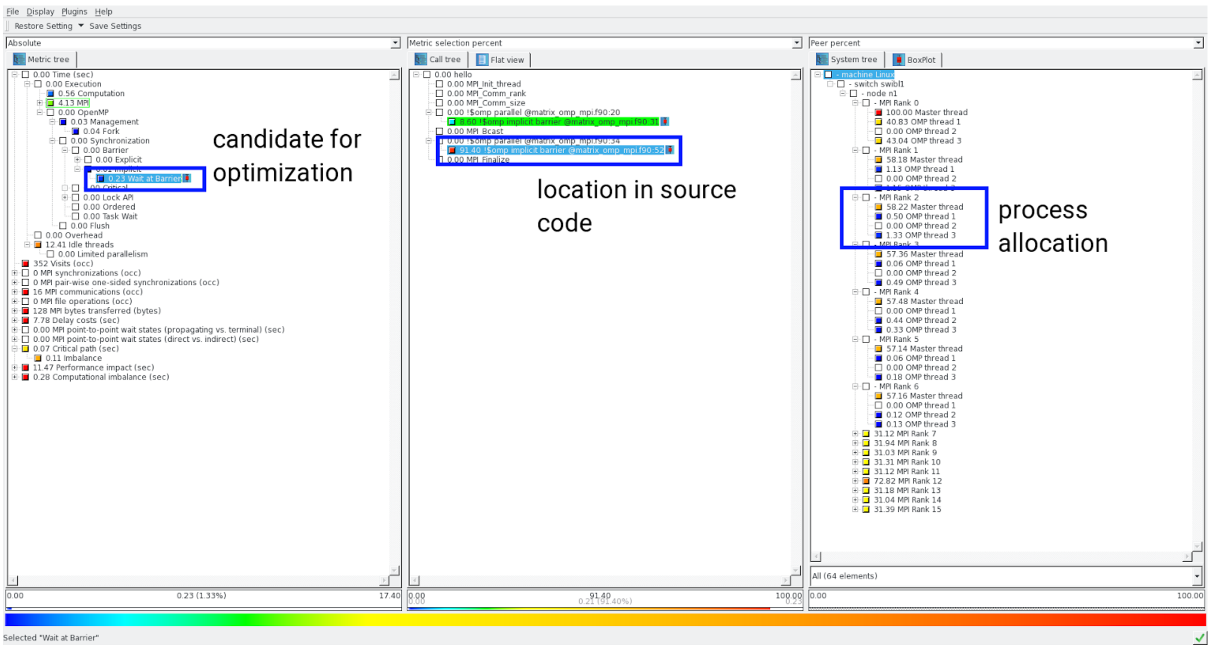
The *SCalable performance Analysis of LArge Scale Applications* (SCALASCA) toolset [14] is designed to analyse the performance of modern software supporting MPI, OpenMP, POSIX threads and hybrids of them on HPC architectures. It also supports Intel Xeon Phi coprocessors in native mode. SCALASCA is in fact an analysis tool for trace files. These are obtained through the Score-P measurement system which is an independent software. The output files' format of Score-P (`OTF2` format) is compatible with other tools such as Periscope, TAU, and Vampir. Score-P is able to monitor execution times of events during the simulation by using hardware counters from perf, rusage, and PAPI. Users can also monitor software counters using a plugin interface (https://github.com/score-p/). Statistics of events are obtained through the PMPI libraries for MPI calls and POMP for OpenMP directives.

Tool Name	SCALASCA (SCalable performance Analysis of Large SCale Applications)
Website	http://www.scalasca.org
Main Features	• Supported programming models: MPI, OpenMP, MPI + OpenMP, and hybrids of them. It is also supported by Intel Xeon Phi coprocessors (native mode). • Features: ○ Trace file is compatible with other common HPC tools. ○ Metrics can be associated to individual functions or to groups (MPI, OMP, COM, and USR). ○ Trace file can be filtered to avoid sampling of irrelevant routines. ○ Efficient/accurate statistics of MPI communications time, Computational load balance, Critical Path, among others. ○ Excellent scalability up to more than one million processes.
Documentation	Full documentation on-line. There are several tutorials available as well as videos.
License	Open source (Scalasca License, BSD-Style)
Supported Platforms	CPU, Intel Xeon Phi (native mode)

Table 5: SCALASCA Overview

SCALASCA follows the so-called postmortem philosophy where relevant data is gathered during runtime and it is summarised when the program finishes without user intervention. In SCALASCA routines/functions can be analysed individually or within a certain context by using groups. Basically, SCALASCA distinguishes among four different groups, i.e., MPI calls (MPI), OpenMP constructs (OMP), user defined functions (USR), and routines which contain MPI/OpenMP instructions (COM) in the same call path. The latter is suitable to preserve the context of MPI/OpenMP instructions.

The trace tool in SCALASCA is especially useful to locate events which lead to unbalanced workloads across processes. Such events are for instance *Late Sender (LS)* or *Late Receiver (LR)* where sending or receiving of messages is not properly synchronised, causing a decrease in overall software performance.



```
0.00 Time (sec)
  0.00 Execution
    0.56 Computation
    4.13 MPI
    0.00 OpenMP
      0.03 Management
        0.04 Fork
      0.00 Synchronization
        0.00 Barrier
          0.00 Explicit
          0.01 Implicit
            0.23 Wait at Barrier
        0.00 Critical
        0.00 Lock API
        0.00 Ordered
        0.00 Task Wait
      0.00 Flush
  0.00 Overhead
  12.41 Idle threads
    0.00 Limited parallelism
352 Visits (occ)
0 MPI synchronizations (occ)
0 MPI pair-wise one-sided synchronizations (occ)
16 MPI communications (occ)
0 MPI file operations (occ)
128 MPI bytes transferred (bytes)
7.78 Delay costs (sec)
0.00 MPI point-to-point wait states (propagating vs. terminal) (sec)
0.00 MPI point-to-point wait states (direct vs. indirect) (sec)
0.07 Critical path (sec)
  0.11 Imbalance
11.47 Performance impact (sec)
0.28 Computational imbalance (sec)
```

```
0.00 hello
  0.00 MPI_Init_thread
  0.00 MPI_Comm_rank
  0.00 MPI_Comm_size
  0.00 !$omp parallel @matrix_omp_mpi.f90:20
    8.60 !$omp implicit barrier @matrix_omp_mpi.f90:31
  0.00 MPI_Bcast
  0.00 !$omp parallel @matrix_omp_mpi.f90:34
    91.40 !$omp implicit barrier @matrix_omp_mpi.f90:52
  0.00 MPI_Finalize
```

```
- machine Linux
  - switch swibl1
    - node n1
      - MPI Rank 0
        100.00 Master thread
        40.83 OMP thread 1
        0.00 OMP thread 2
        43.04 OMP thread 3
      - MPI Rank 1
        58.18 Master thread
        1.13 OMP thread 1
        0.00 OMP thread 2
        1.15 OMP thread 3
      - MPI Rank 2
        58.22 Master thread
        0.50 OMP thread 1
        0.00 OMP thread 2
        1.33 OMP thread 3
      - MPI Rank 3
        57.36 Master thread
        0.06 OMP thread 1
        0.00 OMP thread 2
        0.49 OMP thread 3
      - MPI Rank 4
        57.48 Master thread
        0.00 OMP thread 1
        0.44 OMP thread 2
        0.33 OMP thread 3
      - MPI Rank 5
        57.14 Master thread
        0.06 OMP thread 1
        0.00 OMP thread 2
        0.18 OMP thread 3
      - MPI Rank 6
        57.16 Master thread
        0.00 OMP thread 1
        0.12 OMP thread 2
        0.13 OMP thread 3
      31.12 MPI Rank 7
      31.94 MPI Rank 8
      31.03 MPI Rank 9
      31.31 MPI Rank 10
      31.12 MPI Rank 11
      72.82 MPI Rank 12
      31.18 MPI Rank 13
      31.04 MPI Rank 14
      31.39 MPI Rank 15
```

0.00 0.23 (1.33%) 17.40 | 0.00 91.40 0.21 (91.40%) 100.00 | 0.00 100.00

Figure 17: SCALASCA GUI

5.2. Basic Workflow

Here we describe the most common steps a user needs to do when working with SCALASCA:

1. In order to perform an analysis with SCALASCA, one needs to instrument the code with SCALASCA/Score-P measurement libraries. Since version 2.0 this is achieved with the Score-P instrumentation tool which replaces the compiler's name in use. For instance, a code compiled with the instructions

```
mpif90 -o exe hello.f90
```

   should be then compiled as follows:

```
scorep mpif90 -o exe hello.f90
```

2. Thus, an application using a Makefile for compilation must be changed accordingly. It is important to mention that Score-P supports CMake applications, such as GROMACS. Here, Score-P wrapper can be used instead of the plain compiler (see instrumentation instruction in Appendix section) by using environment variables. **We emphasise that the same compilers used to build Score-P must be used with the software being profiled.** One issue which can arise from the wrong choice of compilers is that SCALASCA/Score-P is only able to collect statistics of MPI calls and not from user defined functions.

3. Perform a summarization analysis first to detect most relevant routines (`scalasca -analyze`, `scalasca -examine -s`). This analysis is highly recommended as in most of the cases routines, which are not important for analysis, should be excluded, otherwise this can lead to unnecessarily big performance data files.
4. Filter routines which are highly visited and are executed quickly with respect to the total elapsed time. These are commonly USR functions. Filtering step is done through filter files, for instance in the following example we exclude two USR unwanted routines:

```
SCOREP_REGION_NAMES_BEGIN
EXCLUDE
        USR 237,925,541,724 216,733,085,472 23787.12    29.5      0.11 gmx_sum4_hpr
        USR   6,765,227,456   5,047,099,336   669.85     0.8      0.13
subc_bb_dist2_simd4
SCOREP_REGION_NAMES_END
```

5. Perform a full trace analysis (`scalasca -analyze -t`). Visualise the results with the CUBE4 graphical tool. Score-P collects statistics of individual events at runtime. This is useful to discover the causes of overheads observed during the analysis summarization experiment.
6. In both stages of the performance analysis, users can visualise the results of the experiments with this tool. It consists of three vertical panels, see Figure 17. The left-most panel displays the *Metric tree*, which includes execution times of user defined functions (computation), MPI/OpenMP related calls/constructs, critical path, and computational imbalance among others. The middle panel shows the routines within the same metric context (*Call tree*). This information is essential to locate bottlenecks at code level. Finally, the right-most panel displays the workload of individual (MPI/OpenMP) processes.

In the following examples we will perform different analyses which display important features of SCALASCA.

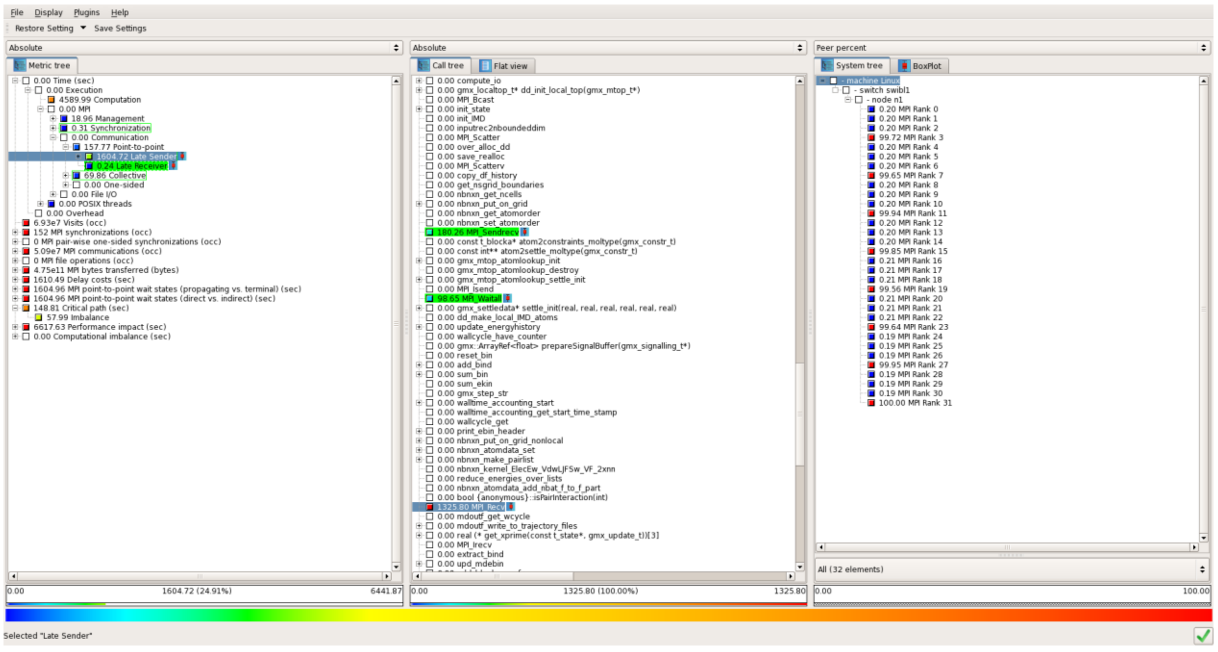
5.3. Case Study: Finding Late Sender Problems and Candidates for Optimisation

In the present example, a solvated protein system consisting of 33,583 atoms was analysed using 32 processes with the pure MPI GROMACS implementation. Also PME algorithm was used and solved with 8 MPI ranks (PME ranks). The remaining 24 processes were used to solve the real space part of electrostatics (PP ranks). As can be seen from the CUBE4 snapshot in Figure 18, left hand column, bottom right, the total execution time was 6441.87 seconds, of which 4589.99 seconds (71%) are spent in computation (Node *Computation* in the *Metric tree*). Interestingly, Late Sender (LS) time represents 24.91% of the total execution time. Looking more in detail, we observe that `MPI_Recv` is the routine which is consuming most of this time (highlighted red in the *Call tree* pane). It turns out that a major contributor to the waiting time in `MPI_Recv` is the function `gmx_pme_recv_coeffs_coords`, which we also saw in the case study with HPCToolkit, section 2.4.

The analysis of the workload of processors (right-most panel) shows that PME ranks (in red) are waiting for a message to be received while PP processors (in blue) are performing other tasks, possibly computations. This tells us that for this particular arrangement of PP and PME ranks, the former finish their task slower than the latter and therefore they need to wait for synchronization when the information for forces/positions are required to dynamically integrate Newton's equations. Even though, we have not used the `tune_pme` tool available in GROMACS to optimise the number of PME ranks, we monitored the execution times by manually changing the number of PME ranks, this is presented in Table 6. Here we can see that for the best performance of GROMACS (32 MPIs and 4 PME ranks) the percentage of LS execution time is still 11% which is far beyond to be neglected.

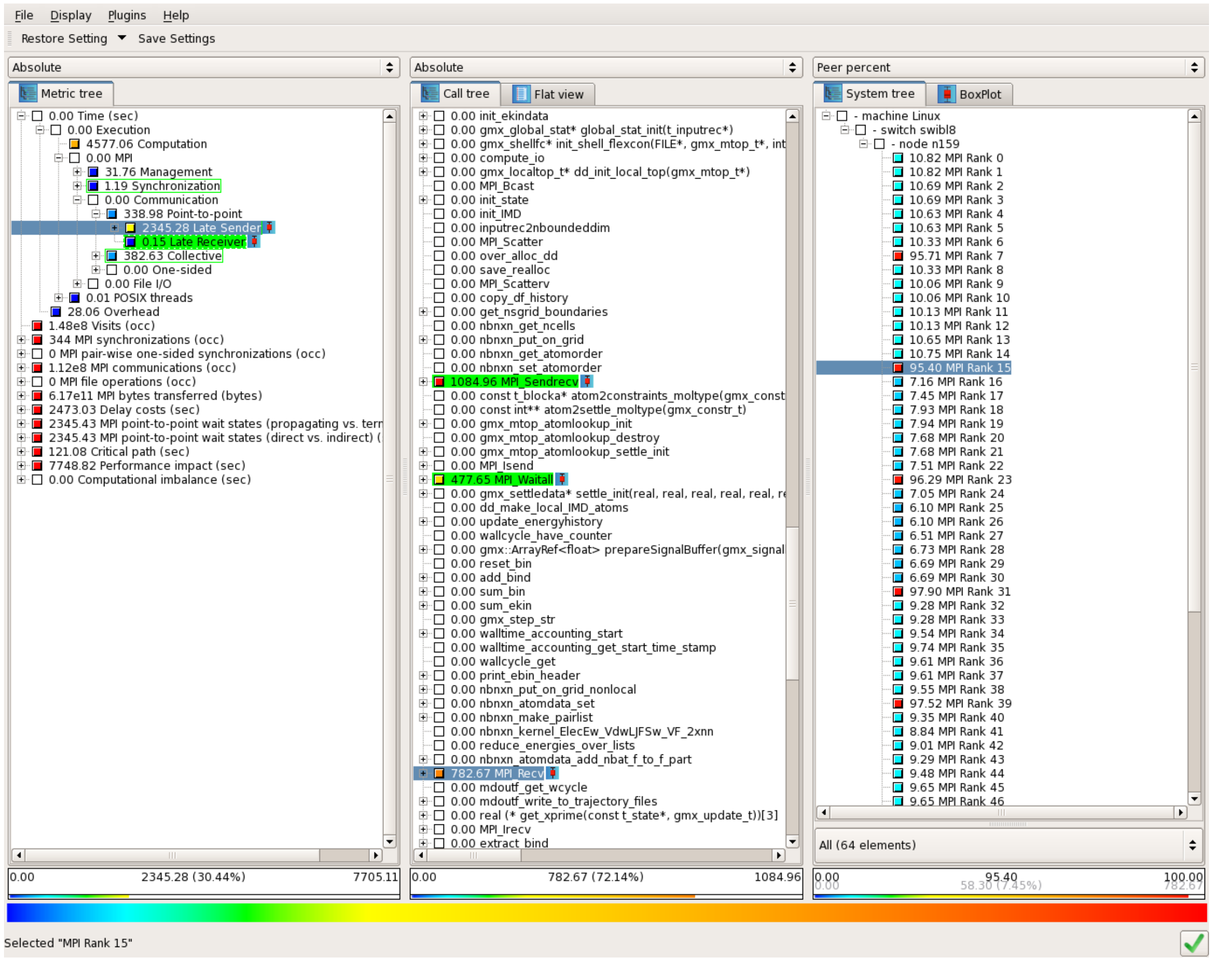
PP ranks	**PME ranks**	**Percentage LS time**	**Performance**
24	8	24	38.12 ns/day
28	4	11	45.57 ns/day
32	0	6	44.54 ns/day

Table 6: Percentages of waiting times due to Late Sender (LS) problems and performance of GROMACS when varying the proportion of PP and PME ranks.



```
File  Display  Plugins  Help
Restore Setting ▾  Save Settings

Absolute
Metric tree
0.00 Time (sec)
  0.00 Execution
    4589.99 Computation
    0.00 MPI
      18.96 Management
      0.31 Synchronization
      0.00 Communication
        157.77 Point-to-point
          1604.72 Late Sender
          0.24 Late Receiver
        69.86 Collective
        0.00 One-sided
      0.00 File I/O
    0.00 POSIX threads
  0.00 Overhead
6.93e7 Visits (occ)
152 MPI synchronizations (occ)
0 MPI pair-wise one-sided synchronizations (occ)
5.09e7 MPI communications (occ)
0 MPI file operations (occ)
4.75e11 MPI bytes transferred (bytes)
1610.49 Delay costs (sec)
1604.96 MPI point-to-point wait states (propagating vs. terminal) (sec)
1604.96 MPI point-to-point wait states (direct vs. indirect) (sec)
148.81 Critical path (sec)
  57.99 Imbalance
6617.63 Performance impact (sec)
0.00 Computational imbalance (sec)
0.00          1604.72 (24.91%)          6441.87

Absolute
Call tree | Flat view
0.00 compute_io
0.00 gmx_localtop_t* dd_init_local_top(gmx_mtop_t*)
0.00 MPI_Bcast
0.00 init_state
0.00 init_IMD
0.00 inputrec2nboundeddim
0.00 MPI_Scatter
0.00 over_alloc_dd
0.00 save_realloc
0.00 MPI_Scatterv
0.00 copy_df_history
0.00 get_nsgrid_boundaries
0.00 nbnxn_get_ncells
0.00 nbnxn_put_on_grid
0.00 nbnxn_get_atomorder
0.00 nbnxn_set_atomorder
180.26 MPI_Sendrecv
0.00 const t_blocka* atom2constraints_moltype(gmx_constr_t)
0.00 const int** atom2settle_moltype(gmx_constr_t)
0.00 gmx_mtop_atomlookup_init
0.00 gmx_mtop_atomlookup_destroy
0.00 gmx_mtop_atomlookup_settle_init
0.00 MPI_Isend
98.65 MPI_Waitall
0.00 gmx_settledata* settle_init(real, real, real, real, real, real)
0.00 dd_make_local_IMD_atoms
0.00 update_energyhistory
0.00 wallcycle_have_counter
0.00 gmx::ArrayRef<float> prepareSignalBuffer(gmx_signalling_t*)
0.00 reset_bin
0.00 add_bind
0.00 sum_bin
0.00 sum_ekin
0.00 gmx_step_str
0.00 walltime_accounting_start
0.00 walltime_accounting_get_start_time_stamp
0.00 wallcycle_get
0.00 print_ebin_header
0.00 nbnxn_put_on_grid_nonlocal
0.00 nbnxn_atomdata_set
0.00 nbnxn_make_pairlist
0.00 nbnxn_kernel_ElecEw_VdwLJFSw_VF_2xnn
0.00 reduce_energies_over_lists
0.00 nbnxn_atomdata_add_nbat_f_to_f_part
0.00 bool {anonymous}::isPairInteraction(int)
1325.80 MPI_Recv
0.00 mdoutf_get_wcycle
0.00 mdoutf_write_to_trajectory_files
0.00 real (* get_xprime(const t_state*, gmx_update_t))[3]
0.00 MPI_Irecv
0.00 extract_bind
0.00 upd_mdebin
0.00          1325.80 (100.00%)          1325.80

Peer percent
System tree | BoxPlot
- machine Linux
  - switch swibl1
    - node n1
      0.20 MPI Rank 0
      0.20 MPI Rank 1
      0.20 MPI Rank 2
      99.72 MPI Rank 3
      0.20 MPI Rank 4
      0.20 MPI Rank 5
      0.20 MPI Rank 6
      99.65 MPI Rank 7
      0.20 MPI Rank 8
      0.20 MPI Rank 9
      0.20 MPI Rank 10
      99.94 MPI Rank 11
      0.20 MPI Rank 12
      0.20 MPI Rank 13
      0.20 MPI Rank 14
      99.85 MPI Rank 15
      0.21 MPI Rank 16
      0.21 MPI Rank 17
      0.21 MPI Rank 18
      99.56 MPI Rank 19
      0.21 MPI Rank 20
      0.21 MPI Rank 21
      0.21 MPI Rank 22
      99.64 MPI Rank 23
      0.19 MPI Rank 24
      0.19 MPI Rank 25
      0.19 MPI Rank 26
      99.95 MPI Rank 27
      0.19 MPI Rank 28
      0.19 MPI Rank 29
      0.19 MPI Rank 30
      100.00 MPI Rank 31
All (32 elements)
0.00                                   100.00

Selected "Late Sender"
```

Figure 18: SCALASCA analysis of a GROMACS execution with 32 MPI ranks (24 PP and 8 PME ranks).



```
File  Display  Plugins  Help
Restore Setting ▾  Save Settings

Absolute
Metric tree
0.00 Time (sec)
  0.00 Execution
    4577.06 Computation
    0.00 MPI
      31.76 Management
      1.19 Synchronization
      0.00 Communication
        338.98 Point-to-point
          2345.28 Late Sender
          0.15 Late Receiver
        382.63 Collective
        0.00 One-sided
      0.00 File I/O
    0.01 POSIX threads
  28.06 Overhead
1.48e8 Visits (occ)
344 MPI synchronizations (occ)
0 MPI pair-wise one-sided synchronizations (occ)
1.12e8 MPI communications (occ)
0 MPI file operations (occ)
6.17e11 MPI bytes transferred (bytes)
2473.03 Delay costs (sec)
2345.43 MPI point-to-point wait states (propagating vs. tern
2345.43 MPI point-to-point wait states (direct vs. indirect) (
121.08 Critical path (sec)
7748.82 Performance impact (sec)
0.00 Computational imbalance (sec)
0.00          2345.28 (30.44%)          7705.11

Absolute
Call tree | Flat view
0.00 init_ekindata
0.00 gmx_global_stat* global_stat_init(t_inputrec*)
0.00 gmx_shellfc* init_shell_flexcon(FILE*, gmx_mtop_t*, int
0.00 compute_io
0.00 gmx_localtop_t* dd_init_local_top(gmx_mtop_t*)
0.00 MPI_Bcast
0.00 init_state
0.00 init_IMD
0.00 inputrec2nboundeddim
0.00 MPI_Scatter
0.00 over_alloc_dd
0.00 save_realloc
0.00 MPI_Scatterv
0.00 copy_df_history
0.00 get_nsgrid_boundaries
0.00 nbnxn_get_ncells
0.00 nbnxn_put_on_grid
0.00 nbnxn_get_atomorder
0.00 nbnxn_set_atomorder
1084.96 MPI_Sendrecv
0.00 const t_blocka* atom2constraints_moltype(gmx_const
0.00 const int** atom2settle_moltype(gmx_constr_t)
0.00 gmx_mtop_atomlookup_init
0.00 gmx_mtop_atomlookup_destroy
0.00 gmx_mtop_atomlookup_settle_init
0.00 MPI_Isend
477.65 MPI_Waitall
0.00 gmx_settledata* settle_init(real, real, real, real, real, r
0.00 dd_make_local_IMD_atoms
0.00 update_energyhistory
0.00 wallcycle_have_counter
0.00 gmx::ArrayRef<float> prepareSignalBuffer(gmx_signal
0.00 reset_bin
0.00 add_bind
0.00 sum_bin
0.00 sum_ekin
0.00 gmx_step_str
0.00 walltime_accounting_start
0.00 walltime_accounting_get_start_time_stamp
0.00 wallcycle_get
0.00 print_ebin_header
0.00 nbnxn_put_on_grid_nonlocal
0.00 nbnxn_atomdata_set
0.00 nbnxn_make_pairlist
0.00 nbnxn_kernel_ElecEw_VdwLJFSw_VF_2xnn
0.00 reduce_energies_over_lists
0.00 nbnxn_atomdata_add_nbat_f_to_f_part
782.67 MPI_Recv
0.00 mdoutf_get_wcycle
0.00 mdoutf_write_to_trajectory_files
0.00 real (* get_xprime(const t_state*, gmx_update_t))[3]
0.00 MPI_Irecv
0.00 extract_bind
0.00          782.67 (72.14%)          1084.96

Peer percent
System tree | BoxPlot
- machine Linux
  - switch swibl8
    - node n159
      10.82 MPI Rank 0
      10.82 MPI Rank 1
      10.69 MPI Rank 2
      10.69 MPI Rank 3
      10.63 MPI Rank 4
      10.63 MPI Rank 5
      10.33 MPI Rank 6
      95.71 MPI Rank 7
      10.33 MPI Rank 8
      10.06 MPI Rank 9
      10.06 MPI Rank 10
      10.13 MPI Rank 11
      10.13 MPI Rank 12
      10.65 MPI Rank 13
      10.75 MPI Rank 14
      95.40 MPI Rank 15
      7.16 MPI Rank 16
      7.45 MPI Rank 17
      7.93 MPI Rank 18
      7.94 MPI Rank 19
      7.68 MPI Rank 20
      7.68 MPI Rank 21
      7.51 MPI Rank 22
      96.29 MPI Rank 23
      7.05 MPI Rank 24
      6.10 MPI Rank 25
      6.10 MPI Rank 26
      6.51 MPI Rank 27
      6.73 MPI Rank 28
      6.69 MPI Rank 29
      6.69 MPI Rank 30
      97.90 MPI Rank 31
      9.28 MPI Rank 32
      9.28 MPI Rank 33
      9.54 MPI Rank 34
      9.74 MPI Rank 35
      9.61 MPI Rank 36
      9.61 MPI Rank 37
      9.55 MPI Rank 38
      97.52 MPI Rank 39
      9.35 MPI Rank 40
      8.84 MPI Rank 41
      9.01 MPI Rank 42
      9.29 MPI Rank 43
      9.48 MPI Rank 44
      9.65 MPI Rank 45
      9.65 MPI Rank 46
All (64 elements)
0.00          95.40                 100.00
0.00          58.30 (7.45%)         782.67

Selected "MPI Rank 15"
```

Figure 19: SCALASCA analysis of GROMACS execution with 64 MPI ranks (56 PP and 8 PME ranks)

By increasing the number of processors to 64, and keeping the same number of PME processors (8), the cost of LS increases slightly to 30.44% as can be seen in Figure 19. The relative computation time is also reduced to 59.40%.

Apart from the *Late Sender* metric, also the *Critical path* metric is helpful for detecting candidates for optimisation: The critical path of an MPI application is given by following the most time-critical data dependencies of the last active MPI process back to the program start. The *Critical path* metric accounts for the execution times of all routines (including their call paths and executing MPI processes) along the application's critical path. Optimising routines with high contributions to the *Critical path* metric are thus likely to reduce the application's total execution time. Looking at this metric for the GROMACS execution with 32 MPI processes, we observed that the kernel for electrostatics and Van der Waals switching force computation is such a major contributor.

5.4. Concluding Comments on SCALASCA and Score-P

Positive

- SCALASCA offers both a qualitative (through graphs) and quantitative (through statistics) analysis of software performance.
- Precise analysis of MPI, OpenMP and user-defined functions.
- Excellent support from the developer community.
- CUBE4 has support for TAU/Vampir plugins.

Tool limitations

- CUDA is not supported.
- Timeline for the execution of routines, i.e. trace analysis, is not available in SCALASCA.

Opportunities for improvements

- Instrumentation is sometimes cumbersome. The build-procedure of Score-P is difficult for non-experienced users (building Score-P may be required if it is not pre-installed or not compatible to the user's compiler tool suite).
- Even when filter files are used, the size of the resulting trace file can be large. Depending on the filtered routines, the overhead is reduced considerably but still is significant. Care should be exercised to avoid filtering relevant routines.
- Even though Score-P records hardware and software threads, these are ignored by the SCALASCA analyser. Metrics from PAPI, rusage, and perf are ignored by SCALASCA.

6. Tool Comparison

6.1. Feature Comparison

	HPCToolkit	**ITAC**	**Extrae and Paraver**	**SCALASCA**
Performance Measurement	Statistical sampling of Timers and hardware counters	Statistical sampling of MPI Calls and code instrumentation	Code instrumentation	Code instrumentation
Profile view	Metric attribution to source code and full call stack (even aggregates on level of loop constructs); Support for derived metrics	Not provided. (Intel VTune can be used for this)	It is possible to define your own views and metrics.	Trace files can be analyzed in text mode or in graphical mode (CUBE4). Topologies can be visualised in 3D plots.
Trace view	Functions with full call stack, However no MPI message visualisation	Timeline with MPI functions, message visualisation and message details	Almost everything can be viewed. Real 3D-views is not available, but would be a nice future feature.	Timeline is not available in SCALASCA.
Supported Platforms	Linux-x86_64, Linux-x86, Linux-Power	Intel processors (MPICH-2 based MPI system required)	x86 CPU, nVidia GPU, Intel Xeon Phi, ARM	CPU, Intel Xeon Phi (native mode)

6.2. Evaluation Overview

	HPCToolkit	**ITAC**	**Extrae and Paraver**	**SCALASCA**
Runtime Overhead	Low (typically around 5%)	Low (typically around 2%)	Low (typically around 5%)	Strongly depends on filtered routines, it can range from 5% to 40% (data from our test cases).
Usability	Overall good with room for improvements (e.g. definition of derived metrics is a bit clumsy, are not stored and need to be redefined every time)	Good and mostly self-explanatory.	Very easy for a first level analysis. More difficult for advanced studies.	Excellent for initial and advanced profiling levels.
Expertise level	Moderate - Novices usually should be able to use the basic tool features within days.	Easy to use	Novices usually should be able to use the basic tool features within days. Due the flexibility of Extrae/Paraver, very advanced analysis is possible, but those require expert knowledge.	Basic features can be mastered in a few days. However, more detailed analysis of all SCALASCA metrics would require longer time investment.
Robustness	Generally good. Problem reports are answered quickly on mailing list.	Very good.	Generally good. Windows versions of Paraver might break down for very large or too many loaded traces.	Very stable and excellent support from developer community.
Documentation	Excellent (User manual also describes general strategies for performance analysis)	Excellent, official Tutorials available	Tutorials are good, but manuals are of limited help for unexperienced users. Paraver's manual is outdated.	Official documentation is very good. Additional material, such as tutorials and videos can be found on the web.
Installation procedure	Binary releases for hpcviewer and hpctraceviewer (Java-based GUI tools) are available. Build procedure of HPCToolkit core tools is well documented and painless (including required external libraries)	Frequently available on Intel-based MPI systems.	For Paraver binary releases are available. For Extrae, the build procedure is straight-forward.	Installation procedure is difficult. Cube4 (GUI tool), Score-P and SCALASCA require installation of several external libraries.
License	Open Source (BSD)	Commercial	Open Source (LGPL)	Open Source (BSD-Style)
Additional Comments	Well ministered mailing list; Queries and problem reports are answered quickly.		Very helpful support by BSC.	One needs to distinguish between SCALASCA and Score-P which are developed by two independent communities but working in close collaboration

7. Conclusion

In this work, we compared the capabilities of four different performance analysis tools, which are 1) HPCToolkit, 2) Extrae and Paraver, 3) SCALASCA and 4) Intel Tools. The aim of the work is to raise the general awareness for the benefits of using specialised performance analysis tools and to lower the threshold for potential users to getting started with such tools. Again we emphasise that the selection of these four tools is not meant as special recommendation. Instead, we encourage potential users to also consider other tools – see also [2] and appendix, section 8.3.

For demonstrating the practical tool usage, the application code GROMACS has been selected as benchmark code due to its high relevance for the molecular dynamics community and in particular for the three CoEs NOMAD, BioExcel and E-CAM. Of course most results (like tool descriptions, hints, best practices, etc.) are not specific to applications with GROMACS and are therefore also useful for a broad audience. Our findings on late sender or late receiver situations are not necessarily actual problems, but could be a starting point for further investigations by the GROMACS developer community. For example, we found that late sender times still remain significant, even when tuning the number of PME ranks for better load balance – compare Table 6.

The main result of the work is a characterization and comparison of the four selected tools and their primary application areas: HPCToolkit is well suited for node-level performance optimization using profiling or tracing. In addition, HPToolkit's profile viewer can also be used for analysing scalability losses by using user-defined metrics and a method called “differential profiling”.

SCALASCA supports graphical and statistical performance analysis and is especially useful to locate events which lead to unbalanced workloads across processes. Such events are for instance Late Sender or Late Receiver where sending or receiving of messages is not properly synchronised, causing a decrease in overall software performance.

Extrae and Paraver supports a wide range of analysis types. Analyses are based on highly customizable trace and histogram visualizations of different metrics such as hardware counters. Also MPI messages can be investigated in the trace view.

Intel Trace Analyzer and Collector (ITAC) is a tool for MPI communication analysis and is primarily helpful for optimizing inter-node performance and scalability. In addition, ITAC can also be used for MPI correctness checking. Complementary to ITAC, Intel VTune can be used for node-level performance optimization.

Acknowledgements

We would like to acknowledge the kind support and valuable inputs of Mark Abraham (GROMACS) and Rossen Apostolov (BioExcel). Moreover, we like to thank the support teams of BSC, HLRS and LRZ for their professional administrative and technical support. Last but not least, we would also like to thank the developers of HPCToolkit, SCALASCA as well as Extrae and Paraver for their quick and very helpful support by e-mail.

This work was financially supported by the PRACE project funded in part by the EU's Horizon 2020 research and innovation programme (2014-2020) under grant agreement 653838. We acknowledge that the results in this paper have been achieved using the Triolith system at the National Supercomputing Centre at Linköping University, Sweden, as well as the PRACE Research Infrastructure resources *MareNostrum* at the Barcelona Supercomputing Center (BSC) in Spain, *HAZELHEN* at the High-Performance Computing Center Stuttgart (HLRS) in Germany and *SuperMUC* at the Leibniz Supercomputing Centre (LRZ) in Garching near Munich, Germany.

References

[1] N. Mc Donnel, “Inventory of Exascale Tools and Techniques”, PRACE-4IP Deliverable D7.3, 2016, http://www.prace-ri.eu/IMG/pdf/D7.3_4ip.pdf

[2] M. Lysaght et al. “A Report on the Survey of HPC Tools and Techniques”, PRACE-3IP Deliverable D7.2.1, 2013, http://www.prace-ri.eu/IMG/pdf/d7.2.1.pdf

[3] M. J. Abraham et al., “GROMACS: High performance molecular simulations through multi-level parallelism from laptops to supercomputers”, SoftwareX 1-2 19-25 (2015), http://dx.doi.org/10.1016/j.softx.2015.06.001

[4] T. Darden, D. York, and L. Pedersen, “Particle mesh Ewald: An N·log(N) method for Ewald sums in large systems”, J. Chem. Phys., 1993, 98, 10089, http://dx.doi.org/10.1063/1.464397

[5] M. Bull, “Unified European Applications Benchmark Suite”, PRACE-3IP Deliverable D7.4, 2013, http://www.prace-ri.eu/IMG/pdf/d7.3.2_3ip.pdf

[6] V. Pavlov, P. Petkov, “Data I/O Optimization in GROMACS Using the Global Arrays Toolkit”, PRACE white paper, 2011, http://www.prace-ri.eu/IMG/pdf/Data_IO_Optimization_in_GROMACS_Using_the_Global_Arrays_Toolkit-2.pdf

[7] F. Affinito, A. Emerson, L. Litov, P. Petkov, R. Apostolov, L. Axner, B. Hess, E. Lindahl, M.F. Iozzi, “Performance Analysis and Petascaling Enabling of GROMACS”, PRACE white paper, http://www.prace-ri.eu/IMG/pdf/Performance_Analysis_and_Petascaling_Enabling_of_GROMACS.pdf

[8] D. Grancharov, E. Lilkova, N. Ilieva1, P. Petkov, S. Markov and L. Litov, “Analysis of symplectic integration algorithms with variable step size for petascale biomolecular simulations”, PRACE Whitepaper, 2012, http://www.prace-ri.eu/IMG/pdf/analysis_of_symplectic_integration_algorithms_with_variable_step_size_for_petascale_biomolecular_simulations.pdf

[9] S. Alam, U. Varetto, “GROMACS on Hybrid CPU-GPU and CPU-MIC Clusters: Preliminary Porting Experiences, Results and Next Steps”, http://www.prace-ri.eu/IMG/pdf/wp120.pdf

[10] M. Krentel, J. Mellor-Crummey et al. “HPCToolkit: Tools for performance analysis of optimized parallel programs.” Concurrency and Computation: Practice and Experience, 22(6):685–701, 2010, doi:10.1002/cpe.1553, http://www.hpctoolkit.org/pubs/cpe-2010-hpctoolkit.pdf

[11] R. Asbury, M. Wrinn. "MPI tuning with Intel/spl copy/Trace Analyzer and Intel/spl copy/Trace Collector." *Cluster Computing, 2004 IEEE International Conference on*. IEEE, 2004.

[12] Gelabert, H., and G. Sánchez. "Extrae user guide manual for version 2.2. 0." *Barcelona Supercomputing Center (B. Sc.)* (2011).

[13] Pillet, Vincent, et al. "Paraver: A tool to visualize and analyze parallel code." *Proceedings of WoTUG-18: transputer and occam developments*. Vol. 44. No. 1. 1995.

[14] M. Geimer et al. "The Scalasca performance toolset architecture." *Concurrency and Computation: Practice and Experience* 22.6 (2010): 702-719.

8. Appendix

8.1. HPCToolkit

8.1.1. Basic Workflow in the case of static linkage

Here we shortly provide alternative instructions for the usage of HPCToolkit when static linkage is used for building the application. Note that steps 3 (post-processing measurements) and 4 (measurement presentation and analysis) of the workflow are still the same, as described in sections 2.2.3 and 2.2.4.

Step 1) Building the application

In addition to adding debug symbols (see 2.2.1), also the link line has to be adapted. A linker wrapper called `hpclink` is provided, which links the application with HPCToolkit's performance measurement framework. The typical command line for the final linking stage of the application has the following form:

```
hpclink [linker-command] [linker-args]
```

For instance, when compiling an MPI application `myapp` with `mpicc`, the link line will be something like:

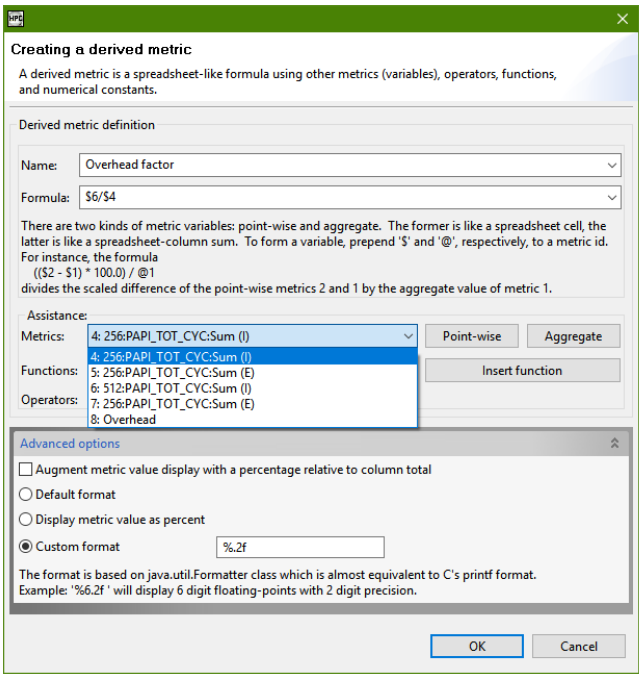
```
hpclink mpicc -o myapp main.o module1.o module2.o …
```

Step 2) Measuring Application Performance

After the application has been linked with `hpclink` as described above, it can be launched as usual (`hpcrun` is not required) and performance measurement is configured via environment variables. Supported environment variables (which correspond more or less 1:1 to `hpcrun` options) are documented in the `hpclink` help (`hpclink -h`), the `hpclink` man page (`man hpclink`) and the HPCToolkit user manual.

8.1.2. Defining a Derived Metric

In `hpcviewer`, derived metrics can be defined by clicking the toolbar button *Add a new derived metric*, f(x). As an example, Figure 20 shows the definition of a derived metric *Overhead factor*. The *Formula* text field shows the underlying expression for this new metric, *$6/$4*, which is the quotient of the metrics *512:PAPI_TOT_CYC:Sum (I)* (= *$6*) and *256:PAPI_TOT_CYC:Sum (I)* (= *$4*). As display format, *%.2f* has been chosen, i.e. metrics will be rounded to exactly 2 decimal places for easier readability. For more information on the definition of derived metrics, we refer to section 6.5 of the HPCToolkit's User's Manual.



Creating a derived metric

A derived metric is a spreadsheet-like formula using other metrics (variables), operators, functions, and numerical constants.

Derived metric definition

Name: Overhead factor

Formula: $6/$4

There are two kinds of metric variables: point-wise and aggregate. The former is like a spreadsheet cell, the latter is like a spreadsheet-column sum. To form a variable, prepend '$' and '@', respectively, to a metric id.
For instance, the formula
  (($2 - $1) * 100.0) / @1
divides the scaled difference of the point-wise metrics 2 and 1 by the aggregate value of metric 1.

Assistance:

Metrics: 4: 256:PAPI_TOT_CYC:Sum (I)

- 4: 256:PAPI_TOT_CYC:Sum (I)
- 5: 256:PAPI_TOT_CYC:Sum (E)
- 6: 512:PAPI_TOT_CYC:Sum (I)
- 7: 256:PAPI_TOT_CYC:Sum (E)
- 8: Overhead

Functions:

Operators:

Advanced options

☐ Augment metric value display with a percentage relative to column total

○ Default format

○ Display metric value as percent

● Custom format: %.2f

The format is based on java.util.Formatter class which is almost equivalent to C's printf format.
Example: '%6.2f ' will display 6 digit floating-points with 2 digit precision.

Figure 20: `hpcviewer`'s user interface for creating a derived metric.

8.1.3. Using HPCToolkit's measurement API

The following changes have been added to GROMACS' source code in order to starting HPCToolkit sampling manually halfway during the simulation (requires the usage of the `mdrun` command line switch `-resethway` and `hpcrun` switch `--delay-sampling`):

```
diff --git a/src/gromacs/CMakeLists.txt b/src/gromacs/CMakeLists.txt
index b7d9bf4..23eca2f 100644
--- a/src/gromacs/CMakeLists.txt
+++ b/src/gromacs/CMakeLists.txt
@@ -187,7 +187,9 @@ target_link_libraries(libgromacs
                       ${XML_LIBRARIES}
                       ${LMFIT_LIBRARIES_TO_LINK}
                       ${THREAD_LIB} ${GMX_SHARED_LINKER_FLAGS} ${OPENCL_LIBRARIES}
-                      ${GMX_STDLIB_LIBRARIES})
+                      ${GMX_STDLIB_LIBRARIES}
+                      $ENV{HOME}/hpctoolkit/5.5.1/lib/hpctoolkit/libhpctoolkit.a)
+target_include_directories(libgromacs SYSTEM PRIVATE
"$ENV{HOME}/hpctoolkit/5.5.1/include")
 set_target_properties(libgromacs PROPERTIES
                       OUTPUT_NAME "gromacs${GMX_LIBS_SUFFIX}"
                       SOVERSION ${LIBRARY_SOVERSION_MAJOR}
diff --git a/src/gromacs/timing/wallcycle.cpp b/src/gromacs/timing/wallcycle.cpp
index 548827d..f60cc0a 100644
--- a/src/gromacs/timing/wallcycle.cpp
+++ b/src/gromacs/timing/wallcycle.cpp
@@ -44,6 +44,8 @@

 #include <array>

+#include <hpctoolkit.h>
+
 #include "gromacs/gmxlib/md_logging.h"
 #include "gromacs/mdtypes/commrec.h"
 #include "gromacs/timing/cyclecounter.h"
@@ -364,6 +366,8 @@ void wallcycle_reset_all(gmx_wallcycle_t wc)
 {
     int i;

+    hpctoolkit_sampling_start();
+
     if (wc == NULL)
     {
         return;
```

8.2. SCALASCA

Instrumentation of GROMACS was done on HAZELHEN cluster of the High Performance Computing Center in Stuttgart. The required modules are:

```
 1) modules/3.2.10.5                    2) eswrap/1.1.0-1.020200.1130.0
 3) switch/1.0-1.0502.60522.1.61.ari    4) craype-network-aries
 5) craype-hugepages16M                 6) craype-haswell
 7) craype/2.5.7                        8) cray-mpich/7.4.3
 9) torque/4.2.10                      10) moab/7.2.9-2014091214-96b8ec34-b9-SUSE11
11) system/ws_tools                    12) system/hlrs-defaults
13) performance/scalasca/2.3-gnu-5.2.0 14) performance/scorep/2.0-beta1-gnu-5.2.0
15) numlib/intel/mkl/11.3              16) tools/python/2.7.8
17) gcc/5.3.0                          18) tools/cmake/3.4.2(default)
19) intel/16.0.3.210                   20) cray-libsci/16.09.1
21) udreg/2.3.2-1.0502.10518.2.17.ari  22) ugni/6.0-1.0502.10863.8.29.ari
23) pmi/5.0.10-1.0000.11050.0.0.ari    24) dmapp/7.0.1-1.0502.11080.8.76.ari
25) gni-headers/4.0-1.0502.10859.7.8.ari 26) xpmem/0.1-2.0502.64982.5.3.ari
27) dvs/2.5_0.9.0-1.0502.2188.1.116.ari  28) alps/5.2.4-2.0502.9774.31.11.ari
29) rca/1.0.0-2.0502.60530.1.62.ari    30) atp/2.0.2
31) PrgEnv-intel/5.2.82
```

The precise installation instructions are given below:

- Within the main directory of your downloaded GROMACS package create the following folders:

```
mkdir build_gmx install_gmx
cd build_gmx
```

- For compiling GROMACS *without instrumentation*, use the following CMake command:

```
CC=cc CXX=CC CXXFLAGS=--std=c++11  cmake .. -DGMX_BUILD_OWN_FFTW=ON -DGMX_MPI=on
-DCMAKE_INSTALL_PREFIX=$HOME/install_gmx -DGMX_FFT_moLIBRARY=fftw3
-DCMAKE_BUILD_TYPE=RelWithDebInfo -DGMX_BUILD_SHARED_EXE=OFF
```

- For compiling GROMACS *with Score-P instrumentation*, use the following CMake command:

```
CC=scorep-cc CXX=scorep-CC CXXFLAGS=--std=c++11  cmake .. -DGMX_BUILD_OWN_FFTW=ON
-DGMX_MPI=on -DCMAKE_INSTALL_PREFIX=$HOME/install_gmx -DGMX_FFT_moLIBRARY=fftw3
-DCMAKE_BUILD_TYPE=RelWithDebInfo -DGMX_BUILD_SHARED_EXE=OFF
```

- In order to work around firewall problems when CMake tries to fetch the FFTW library, download the library manually and move it to this directory:

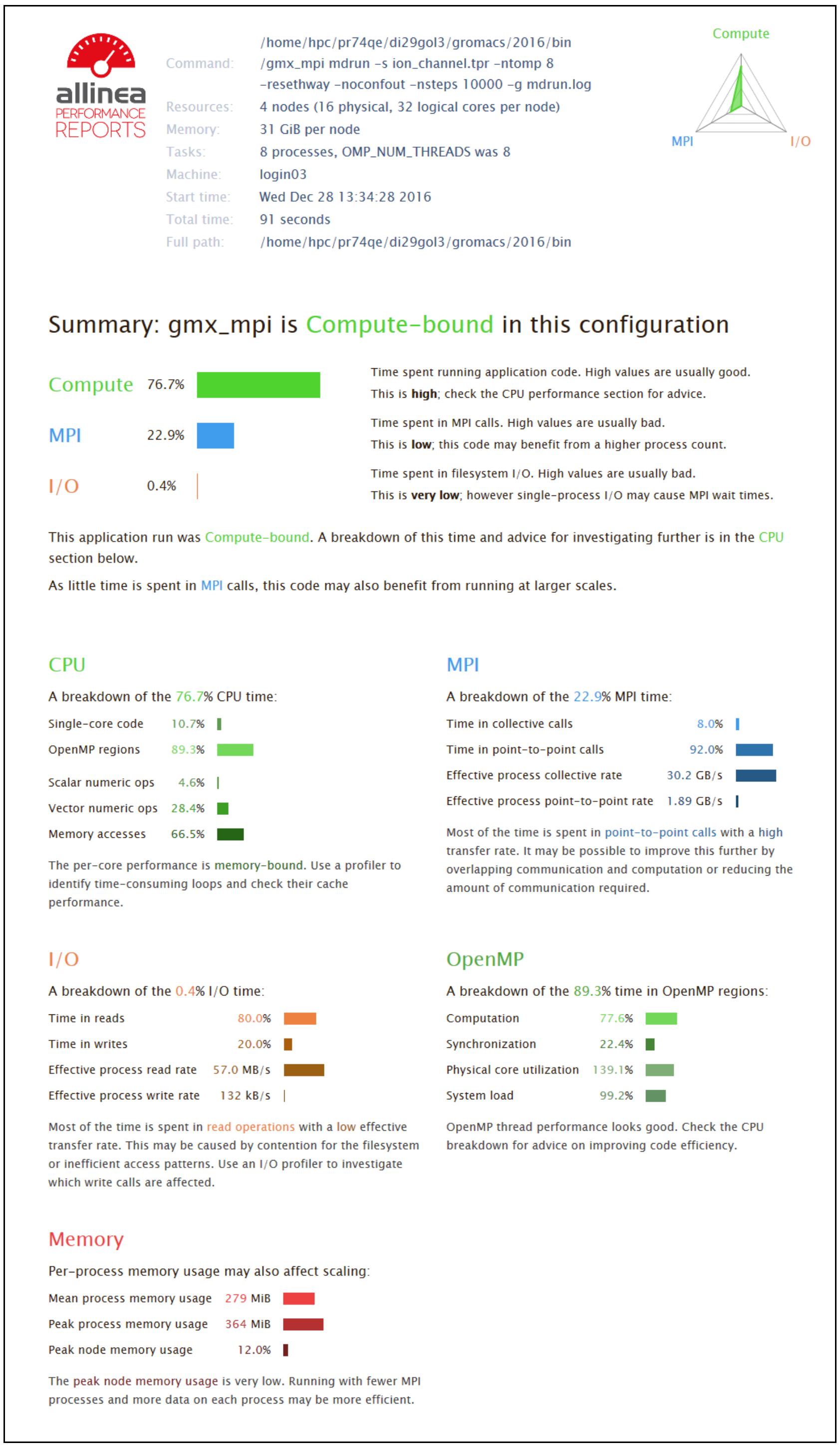
```
cp ../fftw-3.3.4.tar.gz src/contrib/fftw/fftw.tar.gz
```

- Comment out the lines which attempt to download the FFTW package in the CMake-generated file `src/contrib/fftw/fftw-download.cmake`.
- Invoke the build procedure as usual via `make`
- For using the CUBE4 visualiser, load the following module:

```
module load performance/cube/4.4-TP3
```

8.3. Allinea Tools (DDT / MAP / Performance Reports)

As an example of a tool suite which has not been considered in the scope of this work due to limited effort, we would like to mention the Allinea tool suite for HPC applications, i.e. *DDT* for debugging as well as *MAP* and *Performance Reports* for performance analysis. As a commercial tool suite, it fulfils the general expectation of good usability. It has very good integration for most common batch systems which allows to directly start jobs from the *DDT* or *MAP* GUI. Figure 21 shows an example report generated by Allinea Performance Reports, which gives a very quick and good overview on an application's performance.



Command:	/home/hpc/pr74qe/di29gol3/gromacs/2016/bin /gmx_mpi mdrun -s ion_channel.tpr -ntomp 8 -resethway -noconfout -nsteps 10000 -g mdrun.log
Resources:	4 nodes (16 physical, 32 logical cores per node)
Memory:	31 GiB per node
Tasks:	8 processes, OMP_NUM_THREADS was 8
Machine:	login03
Start time:	Wed Dec 28 13:34:28 2016
Total time:	91 seconds
Full path:	/home/hpc/pr74qe/di29gol3/gromacs/2016/bin

Summary: gmx_mpi is Compute-bound in this configuration

Compute	76.7%	Time spent running application code. High values are usually good. This is **high**; check the CPU performance section for advice.
MPI	22.9%	Time spent in MPI calls. High values are usually bad. This is **low**; this code may benefit from a higher process count.
I/O	0.4%	Time spent in filesystem I/O. High values are usually bad. This is **very low**; however single-process I/O may cause MPI wait times.

This application run was Compute-bound. A breakdown of this time and advice for investigating further is in the CPU section below.

As little time is spent in MPI calls, this code may also benefit from running at larger scales.

CPU

A breakdown of the 76.7% CPU time:

Single-core code	10.7%
OpenMP regions	89.3%
Scalar numeric ops	4.6%
Vector numeric ops	28.4%
Memory accesses	66.5%

The per-core performance is memory-bound. Use a profiler to identify time-consuming loops and check their cache performance.

MPI

A breakdown of the 22.9% MPI time:

Time in collective calls	8.0%
Time in point-to-point calls	92.0%
Effective process collective rate	30.2 GB/s
Effective process point-to-point rate	1.89 GB/s

Most of the time is spent in point-to-point calls with a high transfer rate. It may be possible to improve this further by overlapping communication and computation or reducing the amount of communication required.

I/O

A breakdown of the 0.4% I/O time:

Time in reads	80.0%
Time in writes	20.0%
Effective process read rate	57.0 MB/s
Effective process write rate	132 kB/s

Most of the time is spent in read operations with a low effective transfer rate. This may be caused by contention for the filesystem or inefficient access patterns. Use an I/O profiler to investigate which write calls are affected.

OpenMP

A breakdown of the 89.3% time in OpenMP regions:

Computation	77.6%
Synchronization	22.4%
Physical core utilization	139.1%
System load	99.2%

OpenMP thread performance looks good. Check the CPU breakdown for advice on improving code efficiency.

Memory

Per-process memory usage may also affect scaling:

Mean process memory usage	279 MiB
Peak process memory usage	364 MiB
Peak node memory usage	12.0%

The peak node memory usage is very low. Running with fewer MPI processes and more data on each process may be more efficient.

Figure 21: A report generated by Allinea Performance Reports. Important performance metrics of GROMACS are presented in a very clear and appealing way (ion_channel test case, 8 processes running on 4 nodes using 8 threads per process).