John von Neumann Institute for Computing

Developing Scalable Applications with Vampir, VampirServer and VampirTrace

Matthias S. Müller, Andreas Knüpfer, Matthias Jurenz, Matthias Lieber, Holger Brunst, Hartmut Mix, Wolfgang E. Nagel

published in

Parallel Computing: Architectures, Algorithms and Applications ,
C. Bischof, M. Bücker, P. Gibbon, G.R. Joubert, T. Lippert, B. Mohr,
F. Peters (Eds.),
John von Neumann Institute for Computing, Jülich,
NIC Series, Vol. **38**, ISBN 978-3-9810843-4-4, pp. 637-644, 2007.
Reprinted in: Advances in Parallel Computing, Volume **15**,
ISSN 0927-5452, ISBN 978-1-58603-796-3 (IOS Press), 2008.

http://www.fz-juelich.de/nic-series/volume38

Developing Scalable Applications with Vampir, VampirServer and VampirTrace

Matthias S. Müller, Andreas Knüpfer, Matthias Jurenz,
Matthias Lieber, Holger Brunst, Hartmut Mix, and Wolfgang E. Nagel

ZIH (Center for Information Services and HPC)
Technische Universität Dresden, 01162 Dresden
E-mail: {matthias.mueller, andreas.knuepfer, matthias.jurenz}@tu-dresden.de
E-mail: {matthias.lieber, holger.brunst, hartmut.mix, wolfgang.nagel}@tu-dresden.de

Abstract

This paper presents some scalability studies of the performance analysis tools Vampir and VampirTrace. The usability is analyzed with data collected from real applications, i.e. the thirteen applications contained in the SPEC MPI 1.0 benchmark suite. The analysis covers all phases of performance analysis: instrumenting the application, collecting the performance data, and finally viewing and analyzing the data. The aspects examined include instrumenting effort, monitoring overhead, trace file sizes, load time and response time during analysis.

1 Introduction

With (almost) petaflop systems consisting of hundreds of thousands of processors at the high end and multi-core CPUs entering the market at the low end application developers face the challenge to exploit this vast range of parallelism by writing scalable applications. Any performance analysis tool targeting this audience has to provide the same scalability. Even more so, as many performance problems and limitations are only revealed at high processors counts. Furthermore, massively parallel applications tend to be especially performance critical – otherwise they would not employ so many resources.

The remainder of this paper is organized as follows: Section 2 provides background information for the Vampir family of performance analysis tools. The next Sections 3 and 4 look at the overhead involved in the performance analysis process. Firstly, at tracing time to estimate the impact of the measurement infrastructure onto the dynamic behaviour of parallel programs. Secondly, at analysis time when interactive visualization for very huge amounts of data is required to allow a convenient work flow. Section 5 gives a summary and an outlook on future features of scalable performance analysis tools.

2 Description of Vampir and VampirTrace

For the analysis of MPI parallel applications there are a number of performance analysis tools available[1]. Vampir, which is being developed at ZIH, TU Dresden, is well known in the HPC community. Today, there are two versions of Vampir. Firstly, the workstation based classical application with a development history of more than 10 years[2]. Secondly, the more scalable distributed version called VampirServer[3,4].

In addition there is the instrumentation and measurement software VampirTrace. Although not exclusively bundled to Vampir and VampirServer, this is the preferred way to collect the input data for both analysis tools.

2.1 Vampir

Vampir[2,5] is a performance analysis tool that allows the graphical display and analysis of program state changes, point-to-point messages, collective operations and hardware performance counters together with appropriate statistical summaries. It is designed to be an easy to use tool, which enables developers to quickly display program behaviour at any level of detail. Different timeline displays show application activities and communication along a time axis, which can be zoomed and scrolled. Statistical displays provide quantitative results for arbitrary portions of the timelines. Powerful zooming and scrolling allows to pinpoint the real cause of performance problems. Most displays have context-sensitive menus which provide additional information and customization options. Extensive filtering capabilities for processes, functions, messages or collective operations help to reduce the information to the interesting spots. The implementation is based on standard X-Windows and Motif and works on desktop workstations as well as on parallel production systems. It is available for nearly all 32- and 64-bit platforms like Linux-based PCs and Clusters, IBM, SGI, SUN, and Apple.

2.2 VampirServer

VampirServer[3,4] is the next generation, parallel implementation of Vampir with much higher scalability. It implements a client/server architecture. The server is a parallel program which uses standard communication methods such as MPI, pthreads, and sockets. The complex preparation of performance data is carried out by the server component. The server itself consists of one master process and a variable number of worker processes. The visualization of performance results is done by a small client program connecting to the server, more precisely to its master process[3].

VampirServer implements parallelized event analysis algorithms and customizable displays which enable fast and interactive rendering of very complex performance monitoring data. Trace information are kept in distributed memory on the parallel analysis machine. Therefore, ultra large data volumes can be analyzed without copying huge amount of data. Visualization can be carried out from any laptop or desktop PC connected to the Internet.

The implementation is based on standard MPI and X-Windows and works on most parallel production systems (server) and desktop Unix workstations (client). Currently, the list of supported vendor platforms includes: IBM, SGI, SUN, NEC, HP, and Apple. The current version 1.7 was used for all experiments shown below.

2.3 VampirTrace

VampirTrace provides a convenient measurement infrastructure for collecting performance data. VampirTrace supports the developer with instrumentation and tracing facilities tailored towards HPC applications. It covers MPI and OpenMP as well as user code. Instrumentation modifies a target application in order to detect and record run-time events of interest, for example a MPI communication operation or a certain function call. This can be done at source code level, during compilation or at link time with various techniques. The VampirTrace library takes care of data collection within all processes. This includes user events, MPI events, OpenMP events as well as timing information and location (Cluster node, MPI rank, Thread, etc.). Furthermore, it queries selected hardware performance counters available on all platforms supported by PAPI[6].

Automatic instrumentation of the source code using the compiler is available with compilers from GNU, Intel (version 10), IBM, PGI and SUN (Fortran only). Binary instrumentation is performed with DynInst[7]. An analysis of the MPI calls made by the application is made using the so called profile interface of the MPI library.

The collected performance data is written to file using the Open Trace Format (OTF). OTF[9] is a fast and efficient trace format library with special support for parallel I/O. It provides a convenient interface and is designed to achieve good performance on single processor workstations as well as on massive parallel super computers. Transparent blockwise ZLib compression allows to reduce storage size.

VampirTrace is available at http://www.tu-dresden.de/zih/vampirtrace under BSD Open Source license for various platforms, e.g. Linux, IBM, SGI, SUN. The implementation is based on the Kojak tool-set[8]. Development is done at ZIH, TU Dresden, Germany in cooperation with ZAM, Research Center Jülich, Germany, and the Innovative Computing Laboratory at University of Tennessee, US. The current version is 5.3 which was used for the experiments below.

3 Overhead of Instrumentation

The data collection and measurement phase is the first occasion where additional overhead is introduced by the tracing infrastructure. There are two parts involved: instrumentation before execution and measurement during run-time. However, the former has no significant impact at all, because instrumentation does not depend on the number of processes or run time. Yet, the latter imposes notable overhead during run-time. There are four individual contributions:

- initialization at program start-up
- per-event overhead (in event handlers)
- storage of trace data to disk
- finalization

The initialization sets up internal data structures, get symbol information etc. and normally does not add noticeable overhead to the program start-up. Instead, calling event handlers contributes the most part of the critical overhead of tracing. The per-event overhead does not depend on the duration of the recorded event, thus significant overhead is produced for very frequent events, especially frequent short function calls.

Storing trace data on disk produces considerable overhead as well. Therefore, the trace data is first written to memory buffers and afterwards flushed to permanent storage. If

Code	LOC	Language	MPI call sites	MPI calls	Area
104.milc	17987	C	51	18	Lattice QCD
107.leslie3d	10503	F77,F90	43	13	Combustion
113.GemsFDTD	21858	F90	237	16	Electrodynamic simulation
115.fds4	44524	F90,C	239	15	CFD
121.pop2	69203	F90	158	17	Geophysical fluid dynamics
122.tachyon	15512	C	17	16	Ray tracing
126.lammps	6796	C++	625	25	Molecular dynamics
127.wrf2	163462	F90,C	132	23	Weather forecast
128.GAPgeofem	30935	F77,C	58	18	Geophysical FEM
129.tera_tf	6468	F90	42	13	Eulerian hydrodynamics
130.socorro	91585	F90	155	20	density-functional theory
132.zeusmp2	44441	C,F90	639	21	Astrophysical CFD
137.lu	5671	F90	72	13	SSOR

Table 1. Size of Applications (measured in Lines of Code) and programming language

possible, VampirTrace tries to flush only at the finalization phase. In this case there is no interference with the timing of events. Otherwise, the program execution needs to be interrupted eventually. This is marked within the trace but nevertheless it is a severe perturbation. During the finalization phase some post-processing of the trace data is required. This costs additional effort (in computation and I/O) but does not influence the quality of the measurement.

In the following, the overhead will be measured with several experiments on a SGI Altix 4700 machine with 384 Intel Itanium II Montecito cores (1.6 GHz) and 512 GB memory. As a first approach we measured the overhead of instrumenting a function call with a simple test program that calculates π and calls a simple function ten million times. We measured an overhead for each function call of 0.89μs using source code instrumentation and 1.12μs using binary instrumentation. The overhead increases to 4.6μs when hardware performance counters are captured.

For a thorough analysis how this overhead affects the analysis of real applications we instrumented the thirteen applications of the SPEC MPI2007 benchmark[10]. Table 1 contains a short description of the all codes. The analysis consists of two scenarios: first we do full source code instrumentation capturing each function call using a smaller dataset (the so called *train* data set) with 32 CPUs. Second, we do an analysis of the MPI behaviour, capturing only MPI calls made by the application using a large production like data set (called *mref*) with 256 CPUs[10].

In order to get useful performance measurements the overhead introduced by the monitoring must not be too high. In Table 2 the two main contributions to the overhead are distinguished if possible. The overhead during tracing (trace) dominates the total overhead, while the overhead from writing trace data to disk (flush) after tracing is smaller. In some cases, intermediate flusing occurs. Then, the most part of the flushing overhead is brought forward to the tracing phase and is indistinguishable. The missing entries show where no valid trace could be created. For this cases only the sum is given Table 2. Note,

	original	Fully instrumented		original	MPI instrumented	
Code	(train)			(mref)		
	32 CPUs	trace	flush	256 CPUs	trace	flush
104.milc	9.1s	1081s		267s	265.9s	8.2s
107.leslie3d	24.5s	57.0s	43.0s	192s	192.2s	17.7s
113.GemsFDTD	88.9s	-	-	1281s	1259s	32s
115.fds4	18.7s	34.6s	20.7s	605s	597s	21s
121.pop2	57.2s	-	-	444s	5598s	
122.tachyon	12.7s	2595s		264s	271.3s	13.5s
126.lammps	36.1s	(?) 15.6s	-	493s	498s	13s
127.wrf2	24.3s	992s		331s	343s	20s
128.GAPgeofem	4.3s	-	-	106s	-	-
129.tera_tf	89.1s	169.8s	56.1s	290s	287.1s	18s
130.socorro	29.3s	1755s		195s	(?) 177.4s	19s
132.zeusmp2	25.7s	26.0s	3.9s	160s	160.8s	17s
137.lu	12.4s	15.7s	6.4s	92s	96.4s	18.5s

Table 2. Runtime and overhead of fully instrumented and MPI instrumented codes. The overhead is divided in *trace* overhead and overhead from *flush* operations. Entries marked with only on number for both suffer from very large trace sizes that cause intermediate flushing. Entries marked with (?) show reproducible inconsistent results. Missing entries indicate erroneous behaviour where no valid trace could be generated.

that this happens mostly for full traces but only once for MPI-only traces[a].

Figure 1 shows that this effect is directly coupled with total trace volumes. Here, the size occupied in internal buffers is essential. This is not equal to the trace file sizes but proportional. In the same way, different file formats show proportional trace sizes.

As soon as memory buffers are exceeded, flushing becomes inevitable, triggering the unsatisfactory outcome. The memory buffer size is limited by the available memory and by the application's consumption. Only their difference is available for VampirTrace. Here, a quite comfortable buffer size of 2 GB per process was allowed.

In real world performance analysis, intermediate flushing should be avoided by all means! There are two standard solutions which can also be combined: Firstly, limited tracing, i.e. tracing a selected time interval of a subset of all processes only. Secondly, filtering of symbols, i.e. omitting certain functions from tracing.

4 Scalability of Performance Analysis

Once performance data is available from a large scale parallel test-run, one wants to analyze it in order to unveil performance flaws. There are several approaches for automatic, semi-automatic or manual analysis[1] complementing each other. Vampir and VampirServer provide an interactive visualization of dynamic program behaviour. Therefore, the main challenges are firstly, coping with huge amounts of trace data and secondly, accomplishing quick responses to user interactions. Both is vital for a convenient work flow.

Vampir and VampirServer rely on loading the trace data to main memory completely. This is the only way to achieve a quick evaluation on user requests. As a rule of thumb,

[a] In this example there is a tremendous point to point message rate of up to 3 million per second.

VampirServer needs about the memory size of the uncompressed OTF trace. For the VampirServer distributed main memory is good enough, thus, the memory requirements can be satisfied by just using enough distributed peers with local memory capacity each.

In the beginning of a performance analysis session, the trace data needs to be loaded into memory. The distribution across multiple files (i.e. OTF streams) enables parallel reading. This is another important advantage over the sequential counterpart. So, the input speed is only limited by the parallel file system and the storage network. Still, reading rather large amounts of trace data requires some time, compare Fig. 1.

However, this is required only once at the beginning and allows quicker responses during the whole performance analysis session. See first column of Table 3 for the times VampirServer requires to load the SpecMPI traces with 256 streams – corresponding to the number of MPI processes writing their own streams. The first row of Table 4 shows how load time decreases with a growing number of worker nodes for VampirServer with another example. This shows almost perfectly linear scaling.

After loading, the user wants to browse the data with various timeline displays and statistics windows using zooming, unzooming and scrolling. Table 3 shows that this is quite fast for all examples of the SpecMPI MPI-only traces, with the exception of *121.pop2* and *128.GAPgeofem* due to the extremely huge volume. All response times are $\leq 5s$ except for the call tree computation, which needed up to $14s$ for some larger traces. The experiments have been performed with 32+1 VampirServer nodes, i.e. 32 workers plus one master, running on an SGI Altix 4700 with a maximum I/O rate of 2 GB/s.

More detailed experiments are shown in Table 4 for varying numbers of nodes. It reveals that some tasks are always fast regardless of the number of nodes, for example the *timeline* and the *summary timeline*. At the same time, these are the most frequently used ones. The *timeline unzoom* is always fast because it uses internal caching, i.e. it requires no re-computation. For some tasks there is a notable increase in response time as the number of nodes becomes too small (4+1), in particular for the *counter timeline*. However, with enough analysis nodes, there is a sufficient responsiveness for all tasks.

So the evaluation and visualization with VampirServer looks quite promising. Provided enough distributed memory, i.e. distributed worker nodes, it allows a truly interactive work

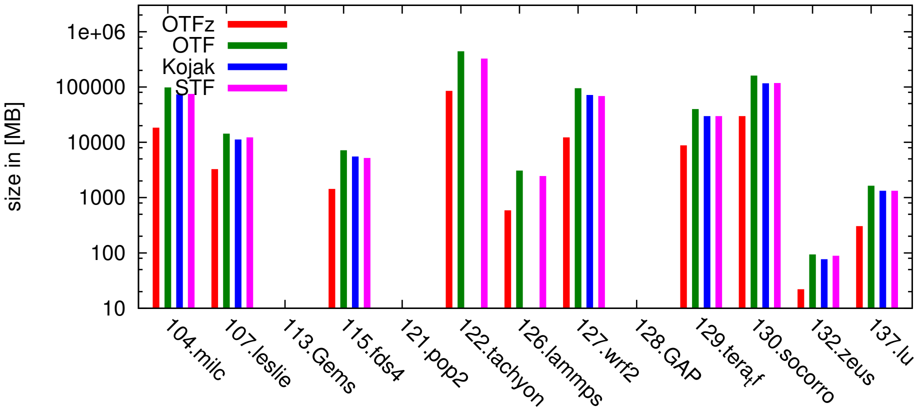

Figure 1. Filesize of the fully instrumented runs with different compressed and uncompressed trace formats.

code	startup & load	Timeline	Process TL	Summary TL	Counter TL	Message Statistics	Call Tree	Trace Size
104	5s	1s	2s	0s	1s	0s	0s	190 MB
107	29s	1s	1s	1s	4s	1s	2s	1.7 GB
113	10s	2s	1s	1s	9s	0s	1s	348 MB
115	7s	1s	1s	1s	3s	0s	0s	67 MB
121								128 GB
122	7s	3s	1s	3s	8s	0s	0s	1.6 MB
126	7s	2s	1s	1s	10s	0s	0s	64 MB
127	68s	4s	1s	4s	14s	1s	6s	4.1 GB
128								
129	14s	5s	2s	4s	9s	2s	2s	710 MB
130	43s	5s	1s	5s	5s	2s	3s	1.4 GB
132	12s	3s	1s	3s	3s	1s	1s	419 MB
137	79s	3s	2s	2s	4s	2s	4s	3.1 GB

Table 3. Vampir Server response times for MPI-only traces of the SpecMPI benchmarks with 32+1 processes.

CPUs for Analysis	Load	Timeline (TL)	TL Zoom	TL Unzoom	Summary TL	Counter TL	Message Statist.	Call Tree
1+1	957	≤ 1	5	≤ 1	≤ 1	29	≤ 1	148
4+1	217	≤ 1	2	≤ 1	≤ 1	8	≤ 1	43
16+1	58	≤ 1	≤ 1	≤ 1	≤ 1	2	≤ 1	12
32+1	29	≤ 1	≤ 1	≤ 1	≤ 1	≤ 1	≤ 1	7

Table 4. Response times for the *107.leslie3d* full trace (3.2 GB compressed / 13.6 GB uncompressed in OTF).

flow even for huge traces. The software architecture is suitable for distributed memory platforms, which are most common today and allows quite easy extensibility.

5 Summary and Conclusion

We examined the scalability for the Vampir tools family with an extensive set of example applications from the SpecMPI benchmark suite. The experiments for tracing overhead showed reasonable overhead for most cases but also quite substantial overhead for very large traces. The solution is twofold: Either to reduce trace size with existing methods. Or to extend the tracing infrastructure for even lower disturbance with huge traces in the future. The analysis of the resulting traces with VampirServer turned out to be very reliable. Most traces can be visualized and analyzed with reasonable hardware resources in an interactive and convenient way. Only very few gigantic traces were critical. Instead of investing more and more resources for them (and even larger ones in the future) alternative methods might be better suited, which do not require main memory in the order of magnitude of the trace size[11]. Another promising project is to add existing and newly developed evaluation procedures into the existing VampirServer framework. Especially semi-automatic and fully-automatic evaluation may support the user in finding the actual spots that need close manual inspection. Hopefully, those can profit from VampirServer's scalable architecture as much as the ones examined in this paper.

References

1. S. Moore, D. Cronk, K. London and J. Dongarra, *Review of performance analysis tools for MPI parallel programs*, in: Y. Cotronis and J. Dongarra, eds., Recent Advances in Parallel Virtual Machine and Message Passing Interface, 8th European PVM/MPI Users' Group Meeting, LNCS **2131**, pp. 241—248, Santorini, Greece, (Springer, 2001).
2. W. E. Nagel, A. Arnold, M. Weber, H.-C. Hoppe and K. Solchenbach. *VAMPIR: Visualization and analysis of MPI resources*, in: Supercomputer, vol. **1**, pp. 69–80, SARA Amsterdam, (1996).
3. H. Brunst and W. E. Nagel, *Scalable performance analysis of parallel systems: Concepts and experiences*, in: G. R. Joubert, W. E. Nagel, F. J. Peters, and W. V. Walter, eds., Parallel Computing: Software, Alghorithms, Architectures Applications, pp. 737–744, (Elsevier, 2003).
4. H. Brunst, D. Kranzlmüller and W. E. Nagel, *Tools for scalable parallel program analysis – VAMPIR NG and DEWIZ.* in: Distributed and Parallel Systems, Cluster and Grid Computing, vol. **777** of *International Series in Engineering and Computer Science*, pp. 93–102, (Kluwer, 2005).
5. H. Brunst, M. Winkler, W. E. Nagel and H.-C. Hoppe, *Performance optimization for large scale computing: The scalable VAMPIR approach*, in: V. N. Alexandrov, J. J. Dongarra, B. A. Juliano, R. S. Renner and C. J. Kenneth, eds., *International Conference on Computational Science – ICCS 2001*, vol. **2074**, pp. 751–760, (Springer, 2001).
6. S. Browne, C. Deane, G. Ho and P. Mucci, *PAPI: A portable interface to hardware performance counters*, in: Proc. Department of Defense HPCMP Users Group Conference, (1999).
7. J. K. Hollingsworth, B. P. Miller and J. Cargille, *Dynamic program instrumentation for scalable performance tools*, in: Proc. Scalable High Performance Compitung Conference, Knoxville, TN, (1994).
8. F. Wolf and B. Mohr, *KOJAK – a tool set for automatic performance analysis of parallel applications*, in: Proc. European Conference on Parallel Computing (EuroPar), vol. **2790** of *LNCS*, pp. 1301–1304, Klagenfurt, Austria, (Springer, 2003).
9. A. Knüpfer, R. Brendel, H. Brunst, H. Mix and W. E. Nagel, *Introducing the open trace format (OTF)*, in: V. N. Alexandrov, G. D. Albada, P. M. A. Slot and J. Dongarra, eds., 6th International Conference for Computational Science (ICCS), vol. **2**, pp. 526–533, Reading, UK, (Springer, 2006).
10. M. S. Müller, M. van Waveren, R. Liebermann, B. Whitney, H. Saito, K. Kalyan, J. Baron, B. Brantley, Ch. Parrott, T. Elken, H. Feng and C. Ponder, *SPEC MPI2007 – an application benchmark for clusters and HPC systems*. in: ISC2007, (2007).
11. A. Knüpfer and W. E. Nagel, Compressible memory data structures for event-based trace analysis, Future Generation Computer Systems, **22**, 359–368, (2006).