Scalable Performance Analysis of Parallel Systems: Concepts and Experiences

Holger Brunst[a b] and Wolfgang E. Nagel[a]

[a]Center for High Performance Computing,
Dresden University of Technology,
01062 Dresden, Germany
{brunst, nagel}@zhr.tu-dresden.de

[b]Department for Computer and Information Science,
University of Oregon,
Eugene, OR, USA
brunst@cs.uoregon.edu

We have developed a distributed service architecture and an integrated parallel analysis engine for scalable trace based performance analysis. Our combined approach permits to handle very large performance data volumes in real-time. Unlike traditional analysis tools that do their job sequentially on an external desktop platform, our approach leaves the data at its origin and seamlessly integrates the time consuming analysis as a parallel job into the high performance production environment.
Keywords: Parallel Computing, Performance Analysis, Tracing, Profiling, Clusters

1. INTRODUCTION

For more than 20 years, parallel systems have been used successfully in many application domains to investigate complex problems. In the early eighties, supercomputing was performed mostly on vector machines with just a few CPUs. The parallelization was based on the shared memory programming paradigm. We should keep in mind that in these times, the performance of applications on such machines was several hundred times higher, compared to even state-of-the-art mainframes. Compared to PCs, the factor was even in the order of a couple of thousands. This means scalability of performance, and its usage was reasonably easy for the end users in those days.

The situation has changed significantly. Today, we still want to improve performance, but we have to use thousands of CPUs to achieve performance improvement factors comparable to those of past times. Most systems are now based on SMP nodes which are clustered – based on improved network technology – to large systems. For very big applications, the only working programming model is message passing (MPI[6]), sometimes combined with OpenMP[3] as a recent flavor of the shared memory programming model. The reality has shown that applications executed on such large systems often have serious performance bottlenecks. The achievement mostly is scalability in parallelism and only sometimes scalability in performance.

This paper describes extended concepts and experiences to support performance analysis on systems with a couple of thousand processors. It presents a new tool architecture which has recently been developed in the ASCI/Earth Simulator scope to support applications on very large machines and to ease the performance optimization process significantly.

2. A DISTRIBUTED PERFORMANCE ANALYSIS SERVICE

The distributed performance analysis service described in this paper has been recently designed and prototyped at Dresden University of Technology in Germany. Based on the experience gained from the development of the performance analysis tool Vampir[1,2], the new architecture uses a distributed approach consisting of a parallel analysis server, which is supposed to be running on a segment of a large parallel production environment, and a visualization client running on a desktop workstation.

Both components interact with each other over the Internet by means of a standard socket based network connection. In the discussion that follows, the parallel analysis server together with the visualization client will be referred to as VNG (**V**ampir **N**ext **G**eneration). The major goals of the distributed parallel approach can be formulated as follows:

1. Keep performance data close to the location where they were created
2. Perform event data analysis in parallel to achieve increased scalability where speedups are on the order of 10 to 100
3. Limit the network bandwidth and latency requirements to a minimum to allow quick performance data browsing and analysis from remote working environments.

VNG consists of two major components, an analysis server (vngd) and a visualization client (vng). Each may be running on a different kind of platform. Figure 1 depicts an overview of the overall service architecture. Boxes represent modules of the two components whereas arrows indicate the interfaces between the different modules. The thickness of the arrows is supposed to give a rough measure of the data volume to be transferred over an interface whereas the length of an arrow represents the expected latency for that particular link.

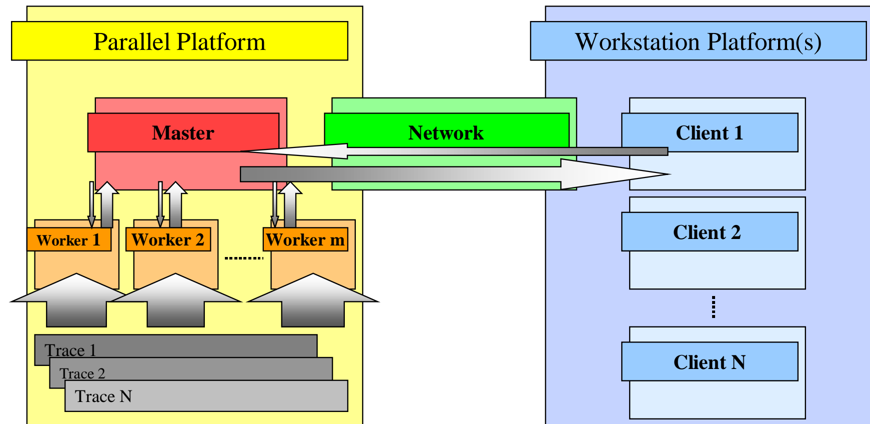

Figure 1. Overview of the distributed performance analysis service

On the left hand side of Figure 1 we can see the analysis server, which is to be executed on a dedicated segment of a parallel computer having access to the trace data as generated by an application. The server can be a heterogeneous program (MPI combined with pthreads), which consists of master and worker processes. The workers are responsible for trace data storage and analysis. Each of them holds a part of the overall trace data to be analyzed. The master is responsible for the communication to the remote clients. He decides on how to distribute analysis requests among the workers and once the analysis requests are completed, the master merges the results into a response to be sent to the client.

The right hand side of Figure 1 depicts the visualization clients running on remote desktop graphics workstations. A client is not supposed to do any time consuming calculations. Therefore, it has a pretty straightforward sequential GUI implementation. The look and feel is very similar to performance analysis tools like Vampir, Jumpshot[8], and many others. Following this distributed approach we comply with the goal of keeping the analysis on a centralized platform and doing the visualization remotely. To support multiple client sessions, the server makes use of multi-threading on the master and worker processes. The next section provides detailed information about the analysis server architecture.

3. A NEW PARALLEL ANALYSIS INFRASTRUCTURE

3.1. Requirements

During the evolution of the Vampir project, we identified three requirements with respect to current parallel computer platforms that typically cannot be fulfilled by classical sequential post mortem software analysis approaches: 1. exploit *distributed memory* for analysis tasks, 2. process both long (regarding time) and wide (regarding number of processes) program traces in *real-time*, 3. *limit the data* transferred to the visualization client to a volume that is independent of the amount of traced event data.

3.2. Architecture

Section 2 has already provided a rough sketch of the analysis server's internal architecture, which will now be described in further detail. Figure 2 can be regarded as a close-up of the left part of the service architecture overview. On the right hand side we can see the MPI master process being responsible for the interaction with the clients and the control over the worker processes. On the left hand side m identical MPI worker processes are depicted in a stacked way so that only the upper most process is actually visible.

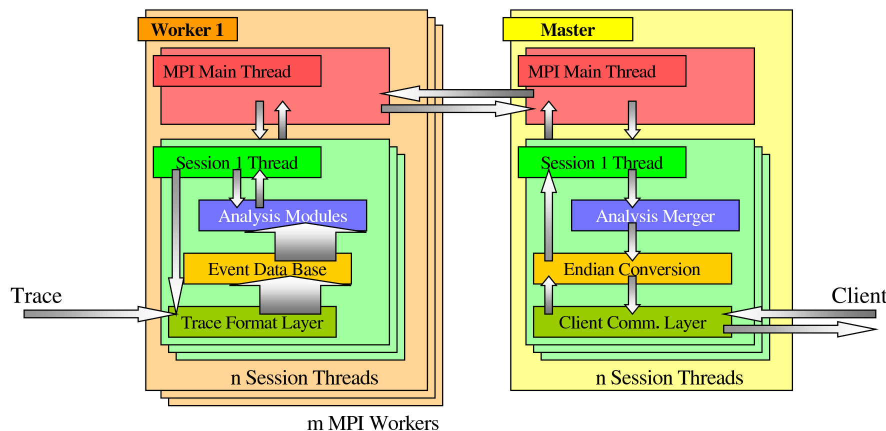

Figure 2. Analysis server in detail

Every single MPI worker process is equipped with one main thread handling MPI communication with the master and if required, with other MPI workers. The main thread is created once at the very beginning and keeps on running until the server is terminated. Depending on the number of clients to be served, every MPI process has a dynamically changing number of n session threads being responsible for the clients' requests. The communication between MPI processes is done with standard MPI constructs whereas the local threads communicate by means of shared buffers, synchronized by mutexes and conditional variables. This permits a low overhead during interactions between the mostly independent components.

Session threads can be subdivided into three different module categories as there are: analysis modules, event data base modules, and trace format modules. Starting from the bottom, trace format modules include parsers for the traditional Vampir trace format (VPT), a newly designed trace format (STF) by Pallas and the TAU[4,7] trace format (TRC). The modular approach allows to easily add other third party formats. The data base modules include storage objects for all supported event categories like functions, messages, performance metrics etc. The final module category provides the analysis capabilities of the server. This type of module performs its work on the data provided by the data base modules.

In contrast to the worker process described above the situation for the vngd master process is slightly different. First of all, the layout with respect to its inherent threads is identical to the one applied on the worker processes. Similar to a worker process, the main thread is also responsible for doing all MPI communication with the workers. The session threads on the other hand have different tasks. They are responsible for merging analysis results received from the workers, converting the results to a platform independent format, and doing the communication with the clients like depicted on the right hand side of Figure 2.

3.3. Loading and Distributing the Trace Data

The main issue in building a parallel analysis infrastructure, is how to distribute the accruing work effectively. One way is to identify analysis tasks on dedicated data sets and distribute them among the worker processes. Unfortunately, this approach is rather inflexible and does not help to speed up single analysis tasks which remain sequential algorithms. Hence, the central idea here is to achieve the work distribution by partitioning the data rather than distributing activity tasks. Naturally, this requires corresponding distributed data structures and analysis algorithms to be developed.

To interactively browse and analyze trace data, the information needs to be in memory to meet the real-time requirements. To quickly load large amounts of data, every worker process (see Figure 1) directly accesses the trace data files via an instance of a trace format support library local to the process, which allows for selective event data input. That is, every worker only has to read the data that is going to be processed by himself.

To support multiple trace file formats (VTF, STF, TAU), we introduced a format unification middleware that provides a standardized interface to internal data structures for third party trace file format drivers. The middleware supports function entry/exit events, point to point communication events, collective operations and hardware counter events. Depending on their type and origin (thread), the data is distributed in chunks among the workers. On each worker process it is attached to a newly created thread that is responsible for all analysis requests concerning this particular program trace data.

3.4. Analyzing the Trace Data in Parallel

One of the major event categories handled by performance analysis tools is the category of block-enter and block-exit events. They can be used to analyze the run time behavior of an application on different levels of abstraction like the function level or the loop level. Finer or coarser levels can be generated depending on the way an application is instrumented. Being a common performance data representative, we will now discuss its parallel processing in detail.

The analysis of the above event type requires a consistent stream of events to maintain its inherent stack structure. The calculation of function profiles, call path profiles, timeline views, etc. highly depends on this structure. To create a degree of parallelism to benefit from, we chose a straight forward data/task distribution. Every worker holds a subset t_w of traces where w is the worker ID. The single trace to worker mapping function can be arbitrary but should not correlate to any thread patterns to avoid load balancing problems. In our implementation, we used an interleaved thread ID to worker mapping, which works best for traces with a large number of threads.

To receive a function profile for an arbitrary time interval the following steps take place. Depending on user input the display client sends a request over the network to the server. On the server side, the session thread (every client is handled by a different session thread) on the MPI master process receives the request and buffers it in a queue. Request by request, the session thread then forwards the request to the main thread on the MPI master process from where it can be broadcasted to all MPI worker processes. The responsible session threads on every worker can now calculate the profile for the traces he owns. The results are handed back (to the MPI master process) by the main thread of every MPI worker process. As soon as the information arrives at the respective session thread on the MPI master process, the results are merged and then handed over to the client. By keeping the event data on the worker processes and exchanging pre-calculated results only between the server and client, network bandwidth and latency is no longer a limiting factor.

4. EVALUATION

To verify this scalable, parallel analysis concept, a test case was chosen and evaluated on our test implementation. The following measurements were taken and compared to sequential results as obtained by the commercial performance analysis tool Vampir 4.0:

- Wall clock time to fully load a selected trace file,
- Wall clock time to calculate a complete function profile display,
- Wall clock time to calculate a timeline representation.

The tests are based on a trace file (in STF format) that was generated by an instrumented run of the sPPM[5] ASCI benchmark application. The trace file has a size of 327 MB and holds approximately 22 million events. Depending on the available main memory VNG successfully handles files that are ten to hundred times bigger but for the reference measurements, Vampir also had to load the full trace into main memory which was the limiting factor. The experiments were carried out on 1, 2, 4, 8, and 16 MPI worker processes and one administrative MPI process, respectively. The test platform was an Origin 3800 equipped with 64 processors (400 MHz MIPS R12000) and 64 GB of main memory. The Client was running on a desktop PC operating under Linux.

4.1. Benchmark Results

Table 1 shows the timing results for the three test cases above. Column 1 of Table 1 gives the wall clock reference times measured for the commercial tool Vampir. The experimental results obtained with VNG will be related to those values. The graph displayed in Figure 3, illustrates the speedups derived from Table 1.

Table 1
Experimental results for the test cases, measured in seconds (wall clock time).

	vampir(1)	vng(1+1)	vng(2+1)	vng(4+1)	vng(8+1)	vng(16+1)
Load Op.:	208.00	91.50	43.45	21.26	10.44	5.20
Timeline:	1.05	0.17	0.18	0.17	0.15	0.16
Profile:	7.86	0.82	0.44	0.25	0.16	0.14

The first row in Table 1 illustrates the loading time for reading the 327 MB test trace file mentioned above. We can see that already the single worker version of VNG outperforms standard Vampir. This is mainly due to linear optimizations we made in the design of VNG which is also the reason for the super scalar speedups we observe in Figure 3 where we compare a multi process run of VNG with a standard Vampir. Upon examination of the speedups for 2, 4, 8, and 16 MPI tasks, we see that the loading time typically is reduced by a factor of two, if the number of MPI tasks doubles. This proves that scalability is granted for a moderate degree of parallelism. Another important aspect is that the performance data is equally distributed among the nodes. This allows to analyze very large trace files in real-time on relatively cheap distributed memory systems whereas with standard Vampir, this is only possible on very large SMP systems.

The second row depicts the update time for a timeline like display. In this case almost no speedup can be observed for an increasing number of MPI processes. This is due to the communication latencies between workers, master and client. Most of the very short time measured is spent in network infrastructure.

Row three shows the performance measurements for the calculation of a full featured profile. The sequential time of approximately eight seconds is significantly slower than the times recorded for the parallel analysis server. The timing measurements show that we succeeded in drastically reducing this amount of time. Absolute values in the order of less than a second even for the single worker version allow smooth navigation in the GUI. The speedups prove scalability of this functionality as well.

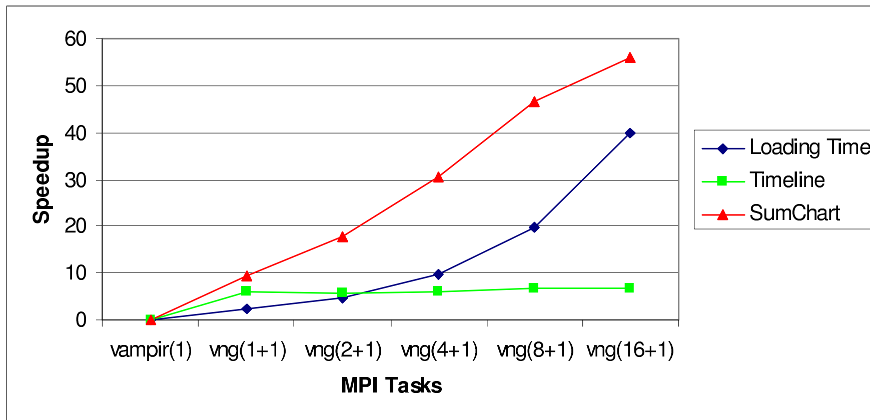

Figure 3. Speedups for the benchmarks

5. CONCLUSION

The numbers above show that we have overcome two major drawbacks of trace based performance analysis. Our client/server approach allows to keep performance data as close as possible to its origin. No costly and time consuming transportation to remote analysis workstations or poor remote display performance needs to be endured anymore. The local visualization clients use a special purpose low latency protocol to communicate with the parallel analysis server which allows for scalable remote data visualization in real-time. Distributed memory programming on the server side furthermore allows the application and the user to benefit from the huge memory and computation capabilities of today's cluster architectures without a loss of generality.

REFERENCES

[1] Brunst, H., Nagel, W. E., Hoppe, H.-C.: Group-Based Peformance Analysis for Multithreaded SMP Cluster Applications. In: Sakellariou, R., Keane, J., Gurd, J., Freeman, L. (eds.): Euro-Par 2001 Parallel Processing, Vol. 2150 in LNCS, Springer, (2001) 148–153

[2] Brunst, H., Winkler, M., Nagel, W. E., Hoppe H.-C.: Performance Optimization for Large Scale Computing: The Scalable VAMPIR Approach. In: Alexandrov, V. N., Dongarra, J. J., Juliano, B. A., Renner, R. S., Kenneth Tan, C. J. (eds.): Computational Science – ICCS 2001, Part II, Vol. 2074 in LNCS, Springer, (2001) 751–760

[3] Chandra, R.: Parallel Programming in OpenMP. Morgan Kaufmann, (2001)

[4] Malony, A., Shende, S.: Performance Technology for Complex Parallel and Distributed Systems. In: Kotsis, G., Kacsuk, P. (eds.): Distributed and Parallel Systems From Instruction Parallelism to Cluster Computing. Proc. 3rd Workshop on Distributed and Parallel Systems, DAPSYS 2000, Kluwer (2000) 37–46

[5] Accelerated Strategic Computing Initiative (ASCI): sPPM Benchmark. http://www.llnl.gov/asci_benchmarks/asci/limited/ppm/asci_sppm.html

[6] Message Passing Interface Forum: MPI: A Message Passing Interface Standard. International Journal of Supercomputer Applications (Special Issue on MPI) 8(3/4) (1994)

[7] Shende, S., Malony, A., Cuny, J., Lindlan, K., Beckman, P., Karmesin, S.: Portable Profiling and Tracing for Parallel Scientific Applications using C++. Proc. SIGMETRICS Symposium on Parallel and Distributed Tools, SPDT'98, ACM, (1998) 134–145

[8] Zaki, O., Lusk, E., Gropp, W., Swider, D.: Toward Scalable Performance Visualization with Jumpshot. The International Journal of High Performance Computing Applications 13(3) (1999) 277–288