# A Distributed Performance Analysis Architecture for Clusters

Holger Brunst[1,2] Wolfgang E. Nagel[1] Allen D. Malony[2]

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

[2] Department of Computer and Information Science
University of Oregon
Eugene, OR, USA
{brunst, malony}@cs.uoregon.edu

## Abstract

*The use of a cluster for distributed performance analysis of parallel trace data is discussed. We propose an analysis architecture that uses multiple cluster nodes as a server to execute analysis operations in parallel and communicate to remote clients where performance visualization and user interactions occur. The client-server system developed, VNG, is highly configurable and is shown to perform well for traces of large size, when compared to leading trace visualization systems.*

**Keywords:** *Parallel Computing, Performance Analysis, Profiling, Tracing, Clusters*

## 1. Introduction

Clusters have proven to be a viable means to achieve large-scale parallelism for computationally demanding scientific problem solving[7]. The relative ease with which more computing, network, and storage resources can be added to a cluster and the available systems software to support them, especially the Linux OS, has made clusters a system of choice for many laboratories. This is not to say that it is also relatively easy to program clusters or to achieve high-performance on cluster systems. While the MPI[6] programming library provides a common programming foundation for clusters, the increasing degree of shared-memory parallelism on cluster nodes encourages mixed-mode styles, as might be obtained from a combination of MPI and OpenMP (or other multi-threading) methods. In either case, it is still necessary to apply performance tools to diagnose a variety of performance problems that can arise in cluster-based parallel execution.

Given the distributed memory system architecture of clusters, many of the performance issues result from the interplay of local computation and remote communication. While measurement techniques based on profiling are useful for highlighting how execution time is spent within (parallel) threads on a single node, process communication behavior between nodes can be only summarized. Profiling may give some insight into communication hotspots or messaging imbalances, but it loses all information about time-dependent communication behavior. In contrast, measurement techniques based on event tracing reveal parallel execution dynamics, allowing users to identify temporal patterns of poor performance behavior at different regions of program execution. The regrettable disadvantage of tracing is the large amount of trace data produced, especially for long running programs, complicating runtime I/O, off-line storage, and post-mortem analysis.

In this paper, we focus our attention on the problem of scalable trace analysis. We propose an integrated performance analysis approach for clusters with the following major goals in mind:

- Keep performance trace data on the parallel cluster platform
- Analyze the parallel trace data in parallel on the cluster
- Provide fast and efficient remote trace visualization and interaction
- Deliver analysis features similar to leading trace analysis tools

Our solution addresses one of the more limiting problems with tracing as a usable method for large-scale performance evaluation, that of dealing with large trace sizes during analysis and visualization.

The paper begins with an overview of the VNG architecture, a prototype for parallel distributed performance analysis we developed at Dresden University of Technology in Germany. Each component is then described in more detail. The runtime system of VNG is particularly important for its scalability. We discuss its operation separately. Experiments were conducted to evaluate the trace processing improvements when using multiple cluster nodes for analysis. These results are shown for two different platforms and compared to baseline performance for the Vampir [3, 4] trace visualization system. The paper concludes with a discussion of the work and directions for the future.

## 2. Architecture Overview

The distributed parallel performance analysis architecture described in this paper has been recently designed and prototyped at Dresden University of Technology in Dresden, Germany. Based on the experience gained from the development of the performance analysis tool Vampir, the new architecture uses a distributed approach consisting of a parallel analysis server, which is supposed to be running on a partition of a parallel clustered environment, and a visualization client running on a remote graphics workstation. Both components interact with each other over a socket based network connection. In the discussion that follows, the parallel analysis server together with the visualization client will be referred to as VNG. The major goals of the distributed parallel approach can be formulated as follows:

1. Keep performance data close to the location where they originally were created
2. Perform event data analysis in parallel to achieve increased scalability where speed-ups are on the order of 10 to 100
3. Allow for a fast and easy to use performance data analysis from remote end-user platforms

VNG consists of two major components, analysis server (vngd) and visualization client (vng). Each can run on a different kind of platform. Figure 1 depicts an overview of the envisioned overall architecture. Boxes represent modules of the 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.

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 machine having access to the trace data generated

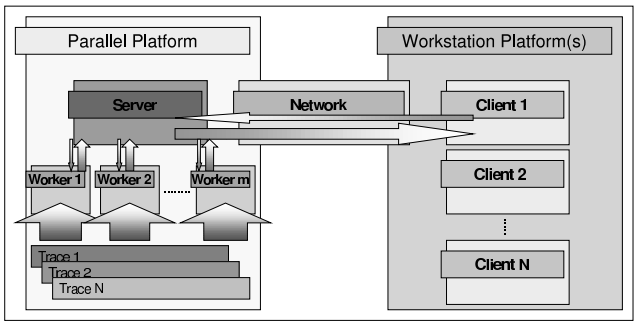


**Figure 1. Architecture Overview**

by an application being traced. The server is a heterogeneous program (MPI combined with pthreads), which consists of worker and boss 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 bosses are responsible for the communication to the remote clients. They decide how to distribute analysis requests among the workers and once the analysis requests are completed, the bosses merge the results into one response that is to be sent to the client.

The right hand side of Figure 1 depicts the visualization client(s) running on a local desktop graphics workstation. The client is freed from time consuming calculations. Therefore, it has a straightforward sequential GUI implementation. The look and feel is very similar to performance analysis tools like Vampir, Jumpshot[9] and others[2]. For visualization purposes, it communicates with the analysis server according to the user's inputs. Multiple clients can connect to the analysis server at the same time.

As mentioned above, the shape of the arrows indicates the quality of the communication links with respect to throughput and latency. Knowing this, we can deduce that the client-to-server communication was designed to not require high bandwidths. In addition, only moderate latencies are required in both directions. Two types of data are transmitted: control information and condensed analysis results. Following this approach, the goal of parallel analysis on the server and remote visualization is achieved. The big arrows connecting the program traces with the worker processes indicate that high bandwidth is a major goal to get fast access to whatever segment of the trace data the user is interested in. This is achieved by reading data in parallel by the worker processes.

To support multiple client sessions, the server makes use of multi-threading on the boss and worker processes. The next section provides detailed information about the analysis server architecture.

## 3. Analysis Server

### 3.1. Requirements

During the evolution of the Vampir project, we identified a set of abstract requirements with respect to current cluster platforms that typically cannot be fulfilled by most classical sequential post mortem software analysis approaches:

- Do *calculations in parallel*.
- Support *distributed memory*.
- Increase *scalability* for both long (regarding time) and wide (regarding number of processes) program traces.
- Allow *preliminary cancellation* of requests. After the user spawned an analysis process, it should be possible to terminate the process without waiting for its completion as it might turn out to be very time consuming. With regards to a parallel infrastructure, this requirement is not easy to fulfill.
- *Limit the data* transferred to the visualization client to a volume that is independent of the amount of traced event data.
- *Portability*. The analysis component should work on as many parallel platforms as possible.
- *Extensibility*. Depending on user demands, new analysis capabilities should be easy to add.

### 3.2. Server Architecture

Section 2 has already provided a rough sketch of the analysis server's internal architecture. We will now go into further detail. Figure 2 can be regarded as a close-up of the left part of the architecture overview. On the right hand side we can see an MPI boss process responsible for the interaction with the client 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.

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

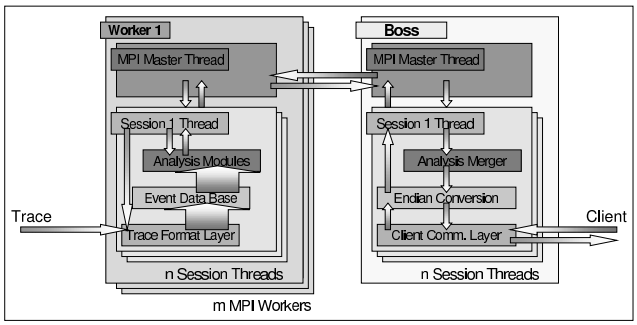


**Figure 2. Analysis Server in Detail**

Session threads can be sub-divided into three different module categories: analysis modules, event database modules, and trace format modules. Starting from the bottom, trace format modules include the parsers for the traditional Vampir trace format (VPT), the newly designed scalable trace format (STF) by Pallas[8] and the TAU[5] trace format (TRC). The modular approach makes it easy to add other third party formats. The database 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 does its work upon the data provided by the database modules.

So far, the worker processes have been discussed. For the boss process(es) the situation is slightly different. The layout of a boss process with respect to its threads is identical to the one applied on the worker processes. Similar to a worker process, the master thread is 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, as depicted on the right hand side of Figure 2.

### 3.3. Communication

**3.3.1. Client/Server.** The design of the communication interface between the display client and the analysis server takes into account the following requirements:

- **Asynchronous/Interleaved communication.** It is possible to trigger multiple requests without actually waiting for the response in order to hide latency whenever multiple instances of the display client make requests for new data.
- **Data transformation/preparation.** To minimize communication overhead, the data transferred between the two components is not raw event data like

what is typically handled by visualization programs. It is important to stress here that the communication volume must not become dependent on the amount of actual event data being processed by the analysis server.

- **Endian conversion.** The display component and the analysis component typically would not reside on the same platform, or even on the same platform architecture. Conversion of data types to identical endianess therefore is supported.
- **Security.** Sensitive data might be transferred between the analysis server and the display client. Our interface allows encryption/authentication via standard mechanisms like SSL, ssh tunneling, or similar, even though this can be in contradiction to latency and throughput optimization.
- **Extensible interface, compatibility.** The interface design takes into account future components transferring data over the same link. Communication between different versions of VNG is possible within the limitation of supported features on both sides.

Based on the above requirements, we identified four different layers of communication to be designed and prototyped:

- Data exchange layer (sockets, shared memory, etc.)
- Endian conversion layer (for inter-architecture communication)
- Serialization packaging layer
- Custom request/response layer

The first layer realizes the pure data transfer by means of simple input and output character buffers. Adapting this layer is sufficient to support different means of communication like sockets, shared memory, etc. Only a very basic API is provided at this layer. On top of this layer type, endian conversion is provided. Data types like float, int, char, string and vectors of them are supported. The third layer is responsible for serialization and packaging of requests and responses. Request and responses are in a format that easily allows adding and removing parameters without loss of compatibility. We therefore used a key word based parameterization for both requests and responses. This final fourth layer is the actual API to the application developer. This interface allows easy request/response generation from and to the analysis server.

We have not tackled the security aspect in this layering approach. We did not want to invent our own security scheme here, but instead we made use of standard ssh tunneling techniques that guarantee proper authentication as well as the encryption of exchanged data. Of course, this layer can be disabled if performance is an issue.

**3.3.2. Boss/Worker.** In the previous section we discussed the "external" client-to-server communication. When it comes to the server's internal communication between worker and boss processes, the situation is different. For the following reasons, we decided to use MPI as the server's internal communication infrastructure:

1. MPI works perfectly on clustered environments.
2. MPI is the standard for message passing.
3. All parallel platforms provide a (presumably efficient) implementation of MPI.
4. MPI has proven to meet high scalability requirements.
5. Many MPI functions (like reductions) provide functionality which we did not have to re-invent.

Unfortunately, the latter cannot be fully exploited because the server must be in the position to cancel a request prematurely: all collective operations are blocking in the sense, that once started, they must be completed on all participating processes. Hence, a separate "control-channel" is required which is based on non-blocking (unfortunately, non-collective) MPI functions. A communication structure uses MPI_Isend, MPI_Irecv and repeated calls to MPI_Test to allow premature cancellation of the current work on a request. Collective operations are then only executed after being told so through the control channel. Unfortunately, this decreases the performance a little. We are looking into better approaches.

**3.3.3. Transaction Protocol.** As stated above, an unsorted key word based transaction protocol seems to have the best potential for future improvements and additions to the communication interface without the loss of compatibility between non-exact matching displays and analysis components. In our case, such a request looks like the following (translated from binary):

```
REQUEST::RequestTimeline,
KEY::Threads, INT(4)::1, 2, 4, 8,
KEY::TimeIntervalStart, FLOAT(1)::StartTime,
KEY::TimeIntervalStop, FLOAT(1)::StopTime,
KEY::HorizontalResolution, INT(1)::800.
```

The ordering of the parameters can be arbitrary. Furthermore, additional unknown or obsolete parameters are allowed, but ignored. Serialization, type translation, and packaging of such requests are generic tasks that work just fine for any future extensions. The above example is written in ASCII text to give an idea of what a generic protocol looks like. The real format of course is a tokenized binary representation, although for debugging purposes, a human readable format is useful as well. This only affects the

packaging layer of our model. Again, this is one of the benefits of a multi-layered client/server interface.

### 3.4. Parallel Analysis

One main issue, is how the work is to be distributed across several workers. It is desirable that all workers are running the same code. Otherwise, software maintenance, flexibility, and extensibility can suffer and possible distribution of files over several file systems cannot be utilized. Instead of diverging into several tasks and assigning them dedicated data sets, the code rather transparently accepts any kind of data and uniformly reacts on it. Hence, the central idea here is to achieve the work distribution by partitioning the data rather than distributing activity tasks.

This, however, implies that it has to be possible to write analysis algorithms following this approach. For our purposes, this is indeed possible, even if reduction requirements make it more difficult than if each analysis task is performed on one process only.

**3.4.1. State/Event Analysis.** One of the major categories of events handled by the analysis server 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 achieved depending on the way an application is instrumented.

The analysis of this 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 depend on this structure. To create a degree of parallelism that the server can benefit from, we chose a two-dimensional task distribution. For relatively short traces, the data is equally distributed among the workers by means of an interleaved process/thread ID to worker mapping, which works best for relatively short traces with a large number of threads. For traces that exceed a certain amount of events per process/thread, the process/thread to worker mapping needs to be adopted every $n$ events. Hence, we achieve an equal distribution of the data among all workers, even in the case of many events but few processes/threads.

In general, the analysis types mentioned above generates results in the form of tables, which can be easily transferred over a network link, as their size does not depend on the number of events being analyzed.

**3.4.2. Communication Analysis.** When it comes to communication events, things get a little bit more difficult. This is basically due to the fact that the nature of communication is at least two-dimensional. Having typically two or more parties involved makes it somewhat difficult to find a proper message to worker mapping so that message profiles and event-oriented timeline representations can be easily calculated. It becomes even more difficult when a certain load balance among the workers is to be fulfilled.

Another problem, which has not been discussed so far, deals with the data aggregation to be done for the information exchange with the client. It arises when messages are to be displayed separately, as it is done by most timeline like tool displays. In a native approach, this type of display requires either sending detailed event information or pre-drawn bitmaps over the network link. Neither solution is really desirable due to the high amount of data or the lack of context. We decided on an approach that even solves another peculiarity. That is the decreasing usefulness of detailed message event displays when dealing with high event densities.

In our solution, the communication events are distributed among the workers in a similar way as the block enter and exit events. Statistics can be easily calculated in a distributed manner. When it comes to detailed event displays with high event densities, events are aggregated in groups which allow both easy transfer to the client and meaningful display of the information in the form of a single graphic (an arrow indicating the actual number of events) and their average properties.

**3.4.3. Performance Metric Analysis.** Some classical performance analysis tools are also capable of acquiring and displaying performance metric data as it can be found on most parallel architectures today. This data is typically generated on a per process basis or for groups of processes, which makes it relatively easy to distribute the data among the workers. Similar to block entry/exit events, profiles and timelines can be calculated independently on the workers. In a subsequent step, the control process merges the condensed data and passes the results over to a client.

**3.4.4. Trace File Input Procedure.** The analysis server is supposed to support multiple trace file formats. Therefore, a format unification layer is needed.

Results from earlier scalability studies for the ASCI PathForward project, revealed that reading event trace data from disk and transforming to appropriate internal data structures required a significant amount of CPU time. We therefore apply a fine-grained parallel input process that does the event translation on top of a much coarser parallel I/O process. To avoid unnecessary inter-worker communication, every worker directly accesses the trace data files via a process-local instance of a trace format support library, which allows for selective event data input. So far, we observed a satisfactory I/O performance on the tested platforms that commonly used NFS. A parallel file system how-

ever, could increase the load performance of our approach even more and fits perfectly well into the architecture.

## 4. Runtime Environment

The server architecture that was explained and discussed in the preceding section was designed for flexibility and portability to today's most common parallel runtime environments. There are, however, certain configuration issues about the underlying hardware platform that should be met in order to allow for the best performance of the distributed performance analysis architecture. The subsequent paragraphs describe an environment that during the development of our parallel analysis server has proven to be well-suited for a seamless and fast distributed performance analysis environment.

### 4.1. Layout

As depicted in Figure 3, the analysis server is expected to run on a small segment of the parallel runtime environment. The bright boxes represent the nodes that are to be executing the parallel application. Execution and tracing of applications takes place on those nodes. The small segment of dark nodes at the right bottom corner is dedicated to interactive analysis of the trace data generated by applications running on the bright nodes. The master analysis node, which is drawn in stripes, does require access from and to the outer world. Clients will have to connect to this particular node in order to place analysis requests.

### 4.2. Common Data Storage System

To avoid costly and time consuming duplication of event trace data, both the calculation and the analysis nodes need a common view on the data. Having any kind of network file system installed, this issue should not pose problems on most target platforms. Keeping the data accessible to both sides of the performance optimization process is crucial for next generation, easy to use analysis tools.

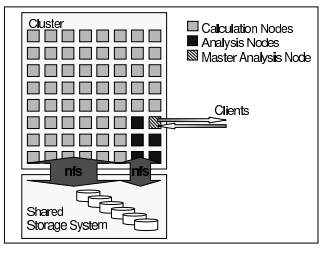


**Figure 3. Runtime Environment Layout**

### 4.3. Security

The distributed approach we propose, requires network access to at least one of the analysis nodes serving as master communication node to the outer world. This brings up security issues. Therefore, the master analysis node should be totally shielded by a firewall allowing just the ssh protocol from a small set of machines known to be trusted clients of the analysis facilities. In addition, the server has been designed to run in user space only, and it is not desired to give potential access to data that is not owned by a client.

### 4.4. Mode of Operation

Experiments with the parallel analysis server have shown that running an interactive analysis daemon requires dedicated resources on the server side to provide reasonable analysis performance. Furthermore, interactive scheduling is required as running the analysis daemon in a batch environment did not turn out to be feasible. This is basically due to the fact that interactive analysis simply requires the analysis server to quickly respond whenever a request comes in. Waiting time generated by a batch system would be a major contradiction to this requirement, and would not be accepted by any tool user.

## 5. System Evaluation

To demonstrate the capabilities of the distributed, parallel analysis system, a set of benchmarks was developed and tested.

### 5.1. The Benchmarks

The following measurements were taken for VNG. They were compared to sequential results obtained for the commercial performance analysis tool Vampir.

- Time needed to fully load a selected trace file.
- Time needed to build the data structures for the summary statistics display.
- Time needed to build the data structures for the timeline display.

In order to obtain the timing results a slightly modified visualization client was used to measure and log the handling of requests in the server.

The tests were carried out for a trace file derived from the sPPM application. The sPPM application is part of the ASCI benchmark suite[1]. The following short description of the code was taken from the benchmark's web page:

> The sPPM benchmark solves a 3D gas dynamics problem on a uniform Cartesian mesh, using a simplified version of the PPM (Piecewise

Parabolic Method) code – hence the ”s” for simplified. The code is written to simultaneously exploit explicit threads for multiprocessing shared memory parallelism and domain decomposition with message passing for distributed parallelism.

The generated trace file has a size of 327 megabytes and holds 22 million events. Due to its size, it is well suited to demonstrate the capabilities of VNG. All experiments were done on 1, 2, 4, 8, and 16 MPI worker tasks plus one administrative MPI task, respectively. These timing experiments where compared with the performance of Vampir which runs sequentially.

## 5.2. The Environment

To provide proof of concept, we used the following local computing environments to perform our tests. Platform A is a Linux cluster that was recently installed at the University of Oregon. It consists of 16 dual-processor nodes interconnected with two independent gigabit Ethernet networks. Platform B is a 128 processor SGI Origin 3800 at Dresden University of Technology. Although the Origin 3800 is an SMP machine, the behavior comes very close to a cluster when running a dedicated MPI Application. It furthermore shows that our architecture is not limited to clustered environments.

### 5.2.1. Platform A:

Linux Cluster
OS: Linux 2.4.20
CPU: Intel Xeon
Clock: 2.80 GHz
Main Memory: 4 GB per Node
Endianness: Little Endian
Display: none
Scheduling: Exclusive

### 5.2.2. Platform B:

SGI multiprocessor architecture (Origin 3800)
OS: IRIX (6.5)
CPU: MIPS R12000
Clock: 400 MHz
Main Memory: 64 GB
Endianness: Big Endian
Display: none
Scheduling: Exclusive

### 5.2.3. Local Desktop Client:

OS: Linux (Kernel 2.4.17)
CPU: AMD Athlon
Clock: 1 GHz
Main Memory: 768 MB
Endianness: Little Endian
Display: X (Local)

| $n$ proc.: | 1* | 1 + 1 | 2 + 1 | 4 + 1 | 8 + 1 | 16 + 1 |
|---|---|---|---|---|---|---|
| Load Op.: | 41.00 | 16.63 | 8.64 | 4.20 | 2.17 | 1.09 |
| Timeline: | 0.35 | 0.07 | 0.07 | 0.06 | 0.07 | 0.07 |
| Profile: | 2.05 | 0.24 | 0.16 | 0.11 | 0.12 | 0.16 |

* sequential Vampir

**Table 1. Timing Results in Seconds for Experiments on Platform A**

| $n$ proc.: | 1* | 1 + 1 | 2 + 1 | 4 + 1 | 8 + 1 | 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 |

* sequential Vampir

**Table 2. Timing Results in Seconds for Experiments on Platform B**

## 5.3. Results

The results for the test cases run on Platform A and B are shown in Table 1 and 2 respectively.

Figures 4 and 5, show the speedups derived from these timing results.

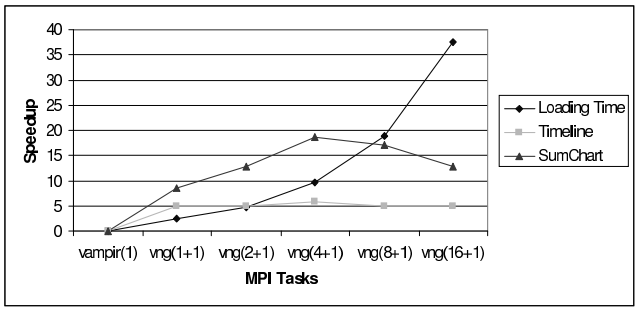


**Figure 4. Speedups for the Parallel Analysis Server on Platform A**

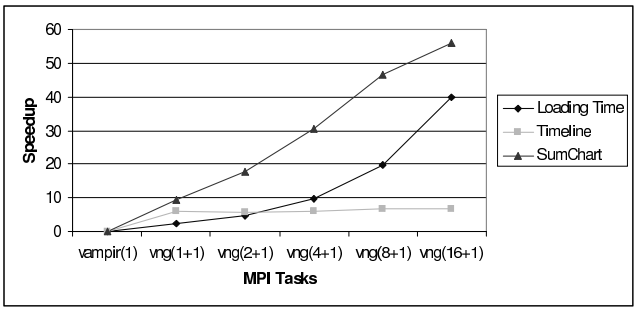


**Figure 5. Speedups for the Parallel Analysis Server on Platform B**

### 5.4. Discussion

The first column of Table 1 and Table 2 gives the timing results obtained with the commercial tool Vampir. The experimental results obtained with VNG will be related to those values.

The first row in Table 1 and Table 2 illustrates the loading time when reading in the 327 megabyte file mentioned in Section 5.1 We can see that even the single worker version of VNG already outperforms standard Vampir. This is mainly due to linear optimizations we did in the design of VNG. This is the principal reason for the super scalar-speedups we observe when we compare a multi-MPI task 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, as the number of MPI tasks doubles. This proves that scalability is achieved. Another important aspect, not mentioned in the tables, is that the amount of memory is reduced per node. This allows one to load very large trace files on a clustered system. With standard Vampir, this was only possible with large SMP systems.

The second row in both tables depicts the update time for the main timeline window. In this case the speedup is not as high as for the loading time. This is mainly due to optimizations that we did in an earlier stage, where we introduced a drawing algorithm that has an $n \log m$ complexity. The parameter of $n$ is equal to the pixel width of a display and $m$ is the number of events to be summarized. From this starting point only a few more optimizations were possible. Notice that the execution time is already quite small.

Row three shows the performance measurements for the calculation of a full featured profile. The sequential time on both platforms is significantly higher than that of the times recorded for the parallel analysis server. The timing measurements show that we succeeded in drastically reducing this amount of time. Absolute values on 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. The speedups for the cluster however indicate that we are dealing with a system that has a weaker processor/network performance ratio compared to the results for the SMP machine although its absolute performance is approximately 4 times higher. This is pretty common for a Linux cluster made of COTS and therefore was not totally unexpected by us. When dealing with bigger trace files, this phenomenon disappears as the time spent for calculations increases while communication stays the same.

What the numbers do not say is that we have basically overcome two of the major drawbacks of trace analysis during the past. The server approach allows us to keep performance data as close to its origin as possible. No costly and annoying transport to a local client is needed anymore. The usage of distributed memory on the server side furthermore allows us to support today's clustered architectures without a loss of generality.

## 6. Conclusion

Parallel trace measurement and analysis has been criticized in recent years for being impractical as a performance evaluation methodology for scalable clustered systems. Primarily, these criticisms point out the issue of large trace volume and the problems it causes for trace storage management, trace analysis, and visualization. One can argue that the amount of trace data is closely tied with the requirements for performance analysis. Hence, the choice of how much trace data to collect and for what events is a decision made during performance problem solving. Certainly, a more refined performance diagnosis strategy could make better, more judicious decisions regarding performance instrumentation to achieve small trace volume. However, from the perspective of the trace analysis system, this is of little consequence, since it must be able to function effectively with traces of large size. Even with careful planning of performance experiments, traces of large-scale parallel machines can quickly grow to the size we tested here.

Consequently, the best we can do to counter the criticism is to improve our trace analysis technology. The VNG system presented in this paper is a validation of tracing as a viable technology for scalable performance analysis. We achieve significant improvements in tracing I/O and analysis functions over the leading commercial system by parallelizing the VNG server using the same cluster technology used for the application's execution. Moreover, greater usability is gained through a separation of analysis and interface, making it possible to support multiple, simultaneous user sessions in a distributed environment.

We are continuing to improve the VNG technology as well as explore its possible applications. For instance, recent

work has attempted to link VNG with a runtime trace generation system, thereby giving users online analysis access to investigate performance problems during long-running programs. Such support could be used to remove uninteresting or redundant sections of the trace from being stored or to inform a computational steering system to guide the application towards better execution performance.

Although scalable cluster systems present VNG with a critical, stressful test case, VNG is not limited to clustered environments. Its usage of standard software technology guarantees a high degrees of portability to other platforms.

## References


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

[2] S. Browne, J. Dongarra, and K. London. Review of performance analysis tools for MPI parallel programs. NHSE Review, 1998.

[3] H. Brunst, W. E. Nagel, and H.-C. Hoppe. Group-based performance analysis for multithreaded SMP cluster applications. In R. Sakellariou, J. Keane, J. Gurd, and L. Freeman, editors, *Euro-Par 2001 Parallel Processing*, number 2150 in LNCS, pages 148–153, Manchester, UK, Aug. 2001. Springer.

[4] 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. K. Tan, editors, *Computational Science – ICCS 2001, Part II*, number 2074 in LNCS, pages 751–760, San Francisco, CA, USA, May 2001. Springer.

[5] A. Malony and S. Shende. Performance technology for complex parallel and distributed systems. In G. Kotsis and P. Kacsuk, editors, *Distributed and Parallel Systems From Instruction Parallelism to Cluster Computing. Proc. 3rd Workshop on Distributed and Parallel Systems, DAPSYS*, pages 37–46. Kluwer, 2000.

[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] H. W. Meuer, E. Strohmaier, J. J. Dongarra, and H. D. Simon. TOP500 supercomputer sites, 21st edition, 2003. http://www.top500.org.

[8] Pallas GmbH, Germany. http://www.pallas.com.

[9] O. Zaki, E. Lusk, W. Gropp, and D. Swider. Toward scalable performance visualization with Jumpshot. *High Performance Computing Applications*, 13(3):277–288, 1999.