

An Integrated Performance Visualizer for MPI/OpenMP Programs*

Jay Hoeflinger¹, Bob Kuhn¹, Wolfgang Nagel², Paul Petersen¹, Hrabri Rajic¹,
Sanjiv Shah¹, Jeff Vetter³, Michael Voss¹, and Renee Woo¹

¹ KAI Software

Intel Americas, Inc.

Champaign, Illinois 61820

{jay.p.hoeflinger, bob.kuhn, paul.m.petersen, hrabri.rajic, sanjiv.shah,
michael.voss, renee.woo}@intel.com

² Center for High Performance Computing

Dresden University of Technology, Germany

nagel@zhr.tu-dresden.de

³ Center for Applied Scientific Computing

Lawrence Livermore National Laboratory

Livermore, California 94551

vetter@llnl.gov

Abstract. As cluster computing has grown, so has its use for large scientific calculations. Recently, many researchers have experimented with using MPI between nodes of a clustered machine and OpenMP within a node, to manage the use of parallel processing. Unfortunately, very few tools are available for doing an integrated analysis of an MPI/OpenMP program. KAI Software, Pallas GmbH and the US Department of Energy have partnered together to build such a tool, VGV. VGV is designed for doing scalable performance analysis - that is, to make the performance analysis process qualitatively the same for small cluster machines as it is for the largest ASCI systems. This paper describes VGV and gives a flavor of how to find performance problems using it.

1 Introduction

Cluster computing has emerged as a defacto standard in parallel computing over the last decade. Now, researchers have begun to use clustered, shared-memory multiprocessors (SMPs) to attack some of the largest and most complex scientific calculations in the world today [8, 2], running them on the world's largest machines including the US DOE ASCI platforms: Red, Blue Mountain, Blue Pacific, and White.

MPI has been the predominant programming model for clusters [12]; however, as users move to “wider” SMPs, the combination of MPI and threads has a

* This work was performed under the auspices of the U.S. Dept. of Energy by University of California LLNL under contract W-7405-Eng-48.

“natural fit” to the underlying system design: use MPI for managing parallelism between SMPs and threads for parallelism within one SMP.

OpenMP is emerging as a leading contender for managing parallelism within an SMP. OpenMP and MPI offer their users very different characteristics. Developed for different memory models, they fill diametrically opposed needs for parallel programming. OpenMP was made for shared memory systems, while MPI was made for distributed memory systems. OpenMP was designed for explicit parallelism and implicit data movement, while MPI was designed for explicit data movement and implicit parallelism. This difference in focus gives the two parallel programming frameworks very different usage characteristics. But these complementary usage characteristics make the two frameworks perfect for handling the two different parallel environments presented by cluster computing: shared memory within a node and distributed memory between the nodes.

Unfortunately, simply writing OpenMP and MPI code does not guarantee efficient use of the underlying cluster hardware. What is more, most existing tools only provide performance information about either MPI or OpenMP, but not both. This lack of integration in our performance tools prevents users from understanding the critical path for performance in their application. To do a good job of performance analysis for such codes, users need detailed information about the expense of operations in their application. Most likely, message passing activity and OpenMP regions are related to the most expensive operations. Viewed in this light, the user needs a performance analyzer to understand the interactions between MPI and OpenMP. For pure message passing codes, several performance analysis tools exist: Vampir [11], TimeScan [3], Paragraph [10], and others. For pure OpenMP codes there is GuideView [7] and a few other proprietary tools from other vendors. For a combination of MPI and OpenMP, we know of only one other tool - Paraver [4].

To address the need for an integrated performance analysis tool, KAI Software and Pallas GmbH have partnered with the Department of Energy through an ASCI Pathforward contract to develop a tool called Vampir/GuideView, or VGV. This tool combines the capabilities of Vampir and GuideView into one tightly-integrated performance analysis tool. From the outset, its design targets performance analysis on systems with thousands of processors.

The purpose of this paper is to describe this tool, how it may be used, and how it can help pin-point the source of performance problems in MPI/OpenMP programs.

2 Related Work

A number of existing tools provide performance analysis of message passing programs. XPVM [5] can be used to analyze the performance of PVM programs. It provides the user an instrumented messaging library and provides a graphical user interface (GUI) for visualizing the performance. The user need not insert instrumentation in their code because the instrumentation exists already in the instrumented library. Vampir [11] and Paragraph [10] are used for MPI programs.

These tools use the MPI profiling interface to capture all MPI calls, then merge the trace information into a single trace file. A visualization program later reads the trace file and draws a graphical representation of the messaging activity between processors.

There are a number of tools for analyzing the performance of HPF codes, which offer a shared memory view to the user, but produce message passing code after having been compiled. The Carnival [13] and Parade [6] systems are examples of these tools. Carnival maintains links to the source code from the instrumentation, so that the user can relate performance to the program, although it was implemented only on IBM systems, to our knowledge. The PARADE system uses by-hand instrumentation and does post-execution “trace animation” through the POLKA animation system.

The Paradyn [1] tool does dynamic instrumentation of a running program by replacing existing instructions with branches to small sections of code called *trampolines* that allow the calling of various instrumentation functions. This provides a very low-overhead and flexible method of instrumentation, but the focus of that project is different from ours since they do not make extensive use of information about the program gathered by a compiler.

The Ovaltine [9] project has developed a tool to analyze the overhead of OpenMP codes, as a way of comparing achieved and achievable performance for a particular code. This type of analysis is already present in the Guideview part of VGV.

The Paraver [4] project, like our own, is focused on building a tool for analyzing the performance of programs that integrate MPI and OpenMP. Paraver is based on a binary instrumenter, that can instrument the MPI functions in a program as well as the OpenMP support functions. Any instrumented region writes trace records to a trace file, which are then displayed through a GUI. The GUI has facilities for user-selected time-scales and zooming in on an arbitrary time range in any display window. The goals of the Paraver project are similar to those of the VGV project, although it is not clear how important scalability is for Paraver. Also, they rely on binary instrumentation, where VGV is based on compiler-inserted instrumentation.

3 Goals of the Project

The main goals of the VGV project are to create an integrated MPI/OpenMP performance analysis tool that is easy to use and that scales well to even the largest systems currently available. This new tool is largely based on the existing Vampir and GuideView tools.

3.1 Scalability

A performance analysis tool faces new problems when it is used for systems with thousands of processors. If the tool is not careful, the amount of information gathered about the performance of a program can become very large, filling

disks or causing large data transfer times. The amount of information displayed on-screen can overwhelm the user if it is not displayed appropriately, and on-screen display space is limited, anyway. The aggressive goal of the VGV project is to quadruple the number of processors that can be analyzed every year for the next two years. This year VGV can handle 1000 processors.

3.2 Integration

To perform effective performance analysis with VGV, there must be an integration of information from Vampir and GuideView. This not only avoids the work of manually coordinating output from the two tools, but also provides a platform for synthesizing an overall performance report. The performance data of both tools should also be integrated with source code information.

3.3 Effective Data Presentation

VGV should present an interface which makes the experience of using it for solving performance problems on large machines not materially different from solving such problems on small systems. The tool should also be able to draw the user's attention to potential performance problems, and help the user locate the source of those problems in the program.

4 Using MPI with OpenMP

Before describing how VGV intends to meet its goals, we will briefly mention some key issues that must be addressed when using MPI with OpenMP. MPI may be used with OpenMP, but the two systems have no knowledge of each other, so a few basic rules must be followed to ensure that they do not interfere with each other.

In general, MPI implementations are not thread-safe, so MPI functions can not be safely used when more than one OpenMP thread is active. Therefore, calls to MPI functions should be done either outside OpenMP parallel regions, as shown in Figure 1, or inside a region in which only one thread is active, such as a `MASTER` region or a `SINGLE` region, as shown in Figure 2.

In addition, if MPI calls are used in a single-threaded section of an OpenMP parallel region, OpenMP barriers on both sides of the single-threaded section are needed to enforce data consistency. This makes sure that the MPI call sees a consistent view of data, and that the following code section sees any modifications to data caused by the MPI call.

Since message passing calls are hard-coded into the program, the messaging structure of the program can not easily adapt to changing patterns of computation. Therefore, the messaging will typically be done to support a fixed structure in the code. OpenMP, on the other hand, can dynamically adjust the number of threads brought to bear on the various parallel loops within the code, so it can adjust to changes in the fine-grained structure of the computation.

```

        CALL MPI_SEND(A(1), N, MPI_REAL, 1, tag, comm, ierr)
        CALL MPI_RECV(A(1), N, MPI_REAL, 0, tag, comm, status, ierr)
!$omp parallel do shared(A, B, N)
        DO I=1,N
            B(I) = F(A(I))
        END DO

```

Fig. 1. Example of using MPI to exchange data outside an OpenMP parallel region.

```

!$omp parallel shared(A, B, N)
!$omp do
        DO I=1,N
            B(I) = F(A(I))
        END DO
!$omp barrier ! to insure consistent memory
!$omp master
        CALL MPI_ALLREDUCE(A, RA, N, MPI_REAL, MPI_SUM, 0, comm)
!$omp end master
!$omp barrier ! to insure consistent memory
!$omp do
        ...
!$omp end parallel

```

Fig. 2. Example of using MPI to do a reduction operation inside an OpenMP parallel region.

Typically, a fixed number of MPI processes are used, corresponding to the number of nodes being used for the computation. The number of processors within each node would represent the maximum number of processors that can be brought to bear on a parallel loop being run within a single MPI process. In adaptive codes, the amount of work in a particular area of a grid can vary widely, so the number of OpenMP processors used in that area might likewise vary. If there is only a small amount of work in a given parallel loop, then only a small number of processors need be used (less processors used means less synchronization overhead).

5 Structure of the Tool

The flow of the integrated tool follows 4 steps:

1. instrumenting the program at compile time,
2. generating an integrated MPI/OpenMP trace file at runtime,
3. post-run performance analysis for MPI with Vampir,
4. analyzing OpenMP performance with GuideView.

This design integrates Vampirtrace and Vampir with the OpenMP components: Guide, the Guide Runtime Library, and GuideView.

Like most MPI performance analysis tools, Vampirtrace uses the MPI library wrapper interface for instrumentation. As each MPI call is performed, an event is written to a trace file. Vampir is the post-run trace file analysis tool.

Guide is a portable OpenMP compiler for Fortran and C++ that restructures source code and inserts calls to the Guide Runtime Library. The Guide Runtime Library layers on top of threads to implement OpenMP functions. The library is instrumented to call clock timers around all the significant OpenMP events. At the end of a run, the information gathered from these timers is written into a statistics file.

The heart of the MPI and OpenMP integration occurs at runtime. The instrumentation of OpenMP and MPI requires coordination. This is achieved by adding OpenMP events to the Vampirtrace API. The Guide Runtime Library is modified to instrument interesting OpenMP events. For each interesting OpenMP event, the execution times are put into a data structure that is time-stamped and sent to the trace file.

In the next phases of the project, dynamic instrumentation will become more important. Then, the user will be able to identify at run time which parts of the program should be instrumented and traced to get a closer and more focused view to performance bottlenecks. The dynamic instrumentation will combine with and complement the compiler-inserted instrumentation.

6 Usage of the Tool

Once an integrated MPI/OpenMP trace file has been created during the application run, it can be viewed by an integrated user interface. Vampir shows the trace file events ordered by time in the timeline display. When an MPI process executes an OpenMP region, a curvy line glyph appears at the top of that process' time line. The user can select that glyph to view that OpenMP region, or can select a set of MPI processes or a time line section for OpenMP analysis.

OpenMP analysis aggregates the OpenMP data structures from all the trace file events in the selection. Then the aggregated data is written to a file where a GuideView server process reads the file.

GuideView displays the OpenMP regions for each MPI process as a separate set of OpenMP data. In this way, the user can use GuideView tools to select a subset of the hundreds of MPI processes that may be running and sort by any OpenMP performance measure. Examples of OpenMP performance measures for sorting are: scheduling imbalance, lock time, time spent in a locked region, and overhead. The user can specify that GuideView show the top or bottom n , where the user specifies n . This mechanism allows a user to compose compound performance queries by sorting on one criteria, filtering the top responders, and then sorting by another criteria.

The user can also view the subroutine profile for one or a selection of MPI processes within GuideView. This can be viewed as inclusive to allow the user to understand the call tree structure, or exclusive to understand which subroutines consume the most time.

One of the important uses of the tool is to locate regions of the program where some processors spend much time waiting while others are doing useful work. This is referred to as a load-imbalance. From the color-coded display, the user can determine how much time each processor spends waiting.

Besides the new analysis features for the OpenMP parts, the usual analysis features of Vampir can be used for the whole program including all parts. As a new feature, hardware performance monitor information is now available for further inspection. Current processor architectures usually offer performance monitor functionality, mostly in the form of special registers that contain various performance metrics like the number of floating-point operations, cache misses etc. The use of these registers is limited because there is no relation to the program structure: an application programmer typically does not know which parts of the application actually cause bad cache behavior. By extending the Vampir trace format, this data is now available inside the Vampir windows to provide identification mechanisms for functions with low performance properties.

As the systems under investigation could have thousands of processors, the scalability requirement has introduced a couple of new hierarchical concepts for the Vampir windows. Especially, a flexible grouping concept has been developed to show data just related to the right level of abstraction: showing information for all processes, showing just accumulated data for the SMP nodes, showing just information for the master thread, etc. This feature enables end users to easily dive into the important program regions that have performance problems.

A further key use of VGV is source code browsing. The source code associated with any part of the performance data may be brought up in a browsing window by clicking the mouse on the data display.

7 Finding Performance Problems with VGV

Figures 3 and 4 show the performance of the MPI/OpenMP version of the program SWEEP3D. In a hypothetical experiment, a user may have run this program on two MPI processes, with four OpenMP threads in each, and discovered that it exhibits very poor speedup. The user could then run VGV and begin with a whole-program view of the performance, such as the frame at the top of Figure 3. This view shows the execution activity for each MPI process as a horizontal bar. The messaging activity between the processes is shown as lines connecting the two process bars. The regions during which OpenMP activity exists is indicated as regions where the wiggle glyph appears at the top of the process bar. As can be seen from the Figure, nearly all of each process bar is covered by OpenMP parallel regions. So where is the problem?

The problem may be investigated by adding OpenMP detail to the MPI activity. The middle frame in Figure 3 adds OpenMP thread activity to the MPI information. We can begin to advance a theory about the cause of the problem from this frame, because there seem to be a large number of small OpenMP execution regions, separated by large gaps.



Fig. 3. Performance display frames for the SWEEP3D program, run on two MPI processes with four OpenMP threads on each. The “curvy line glyph” line at the top of a bar indicates that an OpenMP parallel region is active. The top frame shows the overall timeline of the two MPI processes. The middle frame shows the same information, with the OpenMP threads displayed as well. The bottom frame shows the OpenMP thread activity from a single parallel region.

By zooming in on a single OpenMP region (as in the bottom frame of Figure 3), we see that the parallel execution time of each “helper” thread is separated by a gap at least as large as the execution time itself.

This information seems to point to a large number of relatively small parallel regions, dominated by thread startup and shutdown times. To confirm that theory, we can look at aggregated thread information, as in Figure 4. In the top frame of that Figure, we see the aggregated whole-program information for all threads, and the speedup graph for the code. This information corroborates the view that a large fraction of the time spent in each process is “sequential” (the left-most region in each bar). This corresponds to the thread startup and shutdown times. The parallel execution time is indicated in each right-most bar region, and is a small fraction of the total time. The speedup graph shows that the potential for speedup is very poor in this code.

The aggregated performance information for the whole program, displayed per thread (as in the middle frame of Figure 4), confirms this view. All threads are dominated by sequential execution time. Finally, in the bottom frame of Figure 4 we see the thread information displayed for a single OpenMP parallel region, obtained by clicking on the glyph for a single parallel region.

Each VGV display contains implicit links to the original source code for the program generating the performance data. Any of the timeline regions may be “clicked” to obtain a window positioned at the source code that produced the data. By using this feature, it is possible to display the parallel loops within SWEEP3D that we are judging to be “too small”. From the source code, we could determine how to increase the amount of work done in the parallel regions.

Comparative analysis can also be done by loading trace files from more than one program run. VGV will plot the results together in any of its frames, to make comparison easy, as in Figure 5. This could allow us to experiment with various input data sets, to see how each affected the performance of the program. In the Figure, three program runs are being compared, the serial version of the program, a 2 (MPI) x 2 (OpenMP) version and a 4 (MPI) x 1 (OpenMP) version.

8 Scalability of the Tool

Prior to this project, GuideView already used light-weight summarization techniques to analyze performance statistics for the OpenMP processors. Vampir-trace, on the other hand, wrote trace records to a single trace file for every MPI call. This produces a potentially very large trace file that must not only be stored, but also completely read and analyzed to provide the user with a display.

To be scalable, VGV must adequately address the following issues:

- the disk storage requirements of an event-based tracing tool could become enormous for long runs with large numbers of processors,
- workstation screens have limited space for displaying performance information,
- simply finding a potential performance problem may be very hard in the blizzard of information potentially generated from a massive run.

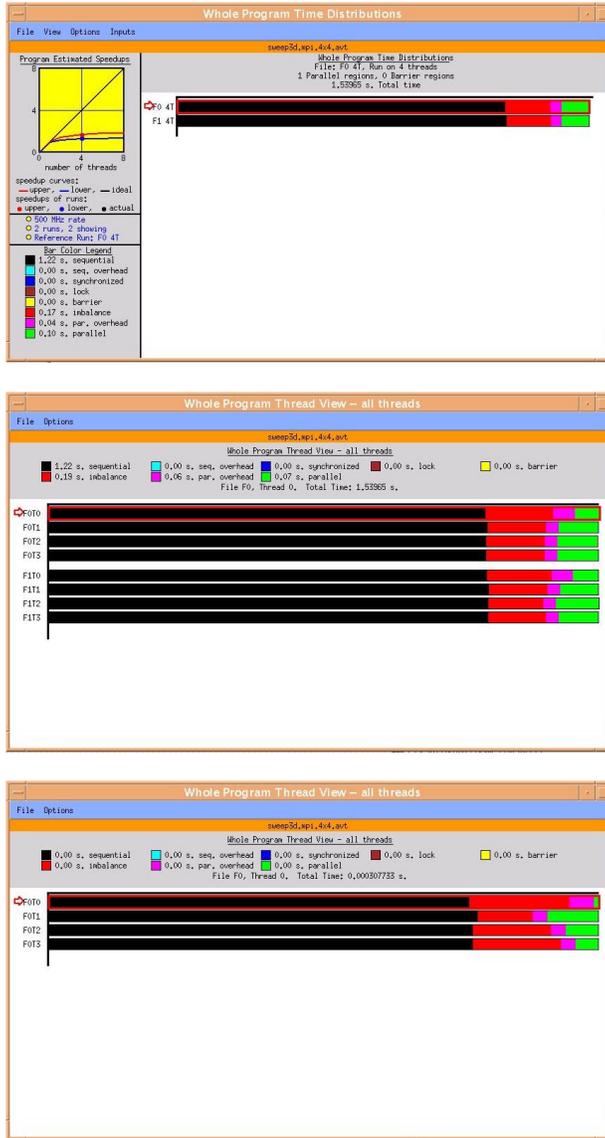


Fig. 4. Performance display frames for the SWEEP3D program, run on two MPI processes with four OpenMP threads on each. The top frame shows the whole program view, displaying aggregated information from all threads and processes. The middle frame shows the threads view of multiple parallel regions. The bottom frame shows the threads view of a single parallel region. In each view, the frames show that the vast majority of time is being spent in sequential execution (the left-most bar region). Parallel execution is the right-most bar region in all cases.

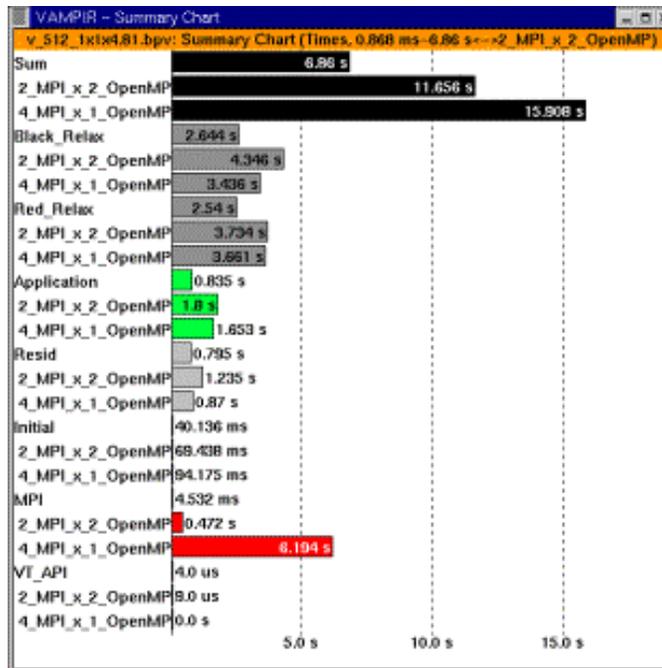


Fig. 5. Comparing three program runs with VGV.

Some of these issues have already been addressed in the current version of VGV. Others will be implemented during the remainder of the project.

VGV will attempt to reduce the size of the trace file through several means:

- event compression** - Specialized trace records can be used for some events and encoded to save space. Collective communication events, which usually require a full trace record for every process can be reduced to a single trace record and a series of small records, one per task. Also, source code line numbers can be encoded to save space in each trace record.
- event combination** - Events occurring commonly together can be replaced by a single event. Very short events which are issued until the MPI state changes (e.g. `MPIProbe` / `MPI_Test`) can be replaced by a single event covering the entire interaction.
- event summarization** - Some events can be summarized by maintaining only min/max/average values and discarding the events.
- structured trace file** - The single trace file can be replaced by multiple, hierarchically structured files. This also saves processing time because a top-level summary file can be processed much more quickly than can the whole original trace file. This allows the user to see a summary then drill down to other levels in the hierarchy for display.
- tracing/instrumentation control** - Tracing can be disabled or enabled according to a variety of criteria. The user could place enable/disable trace calls in the code, or could select specific events to enable/disable, or could trace only certain MPI processes, or a variety of other criteria.

The on-screen presentation of the performance information can be made scalable through vertical scrolling of MPI process time-line information, as well as back-to-front stacking of time-lines.

VGV will use data reduction to attempt to identify potential performance problems for the user. Statistical analysis of the data for a single interval of the user's program can identify processes or processors that require unusual (high or low) amounts of various resources (e.g. cache misses, time, memory access time) and mark them for the user.

We have found that the execution time of the analysis tool can be a major fraction of the time required for the tool. For this reason, the tool will be partitioned into a display component (DC) and a trace processing component (TPC). The TPC can be parallelized and run on a small number of processors. The DC can be potentially multi-threaded.

9 Future Directions for VGV

VGV is still in the design and development stage, since we are only in the second year of the three-year project. We expect significant improvements in the system over the remainder of the project.

Possible future directions for VGV include:

- Two way interaction between Vampir and GuideView. When the user selects something in the GuideView display, VGV should map that back to the Vampir Timeline. For example, when selecting a parallel region, a menu item would zoom in on the first instance of the parallel region in the time line so that you could see the actual event distribution inside the parallel region.
- Integrated resource management. Presently, Vampir and GuideView are run in separate processes. This gives them no ability to co-manage the resources they use. If they were combined in the same process, they could coordinate their use of memory, use of the screen and use of disk space. They could also react as a single unit to signals and user requests.
- Move away from Java to some other, more portable, window manager. Java, contrary to popular belief, has proven to be inconsistently portable when it comes to managing the screen display. This is worse on older platforms and can be attributed to some of the early Java implementations. A more mature system, such as Motif, may be more portable.

10 Conclusion

Vampir/GuideView is intended to be a flexible, easy-to-use tool for finding performance problems in programs written with a combination of MPI and OpenMP, that run for extremely long times and use thousands of processors. We know of no other commercial tool targeted at MPI/OpenMP, and certainly none with the ability to handle the massive runs common on the ASCI machines. During the remaining two years of its development for the ASCI project, we believe that it will become a tool that can be used for the largest ASCI clusters, and will help users quickly pin-point performance anomalies in their codes.

References

1. B. Buck and J.K. Hollingsworth. An API for Runtime Code Patching. *to appear in Journal of Supercomputing Applications and High Performance Computing.*
2. A.C. Calder, B.C. Curtis, and et al. High-Performance Reactive Fluid Flow Simulations Using Adaptive Mesh Refinement on Thousands of Processors. In *Supercomputing 2000: High Performance Networking and Computing Conference*, 2000. electronic pub.
3. Etnus LLC, <http://www.etnus.com/Products/TimeScan/index.html>. *TimeScan Multiprocess Event Analyzer*, 2001.
4. European Center for Parallelism of Barcelona, Technical University of Catalonia, <http://www.cepba.upc.es/paraver/docs/OMPItraceIBM.pdf>. *Paraver Reference Manual*, 2000.
5. J.A. Kohl and G.A. Geist. XPVM 1.0 User's Guide. Technical Report ORNL/TM 12981, Oak Ridge National Laboratory, Oak Ridge, Tennessee, November 1996.
6. J.T. Stasko. The PARADE Environment for Visualizing Parallel Program Executions: A Progress Report. Technical Report Technical Report GIT-GVU-95-03, Graphics, Visualization, and Usability Center, Georgia Institute of Technology, Atlanta, GA, January 1995.

7. KAI Software, a division of Intel Americas, <http://www.kai.com/parallel/kappro/guideview>. *GuideView Performance Analyzer*, 2001.
8. A.A. Mirin, R.H. Cohen, and et al. Very High Resolution Simulation of Compressible Turbulence on the IBM-SP System. In *Supercomputing '99: High Performance Networking and Computing Conference*, 1999. electronic pub.
9. M.K. Bane and G.D. Riley. Automatic Overheads Profiler for OpenMP Codes. In *Proceedings of the European Workshop on OpenMP (EWOMP) 2000, Edinburgh, Scotland, U.K.*, September, 2000.
10. M.T. Heath and J.A. Etheridge. Visualizing the Performance of Parallel Programs. *IEEE Software*, 8(5):29-39, September 1991.
11. Pallas GmbH, <http://www.pallas.de/pages/vampir.htm>. *Vampir 2.5 - Visualization and Analysis of MPI Programs*, 2001.
12. G.F. Pfister. *In Search of Clusters: The Coming Battle in Lowly Parallel Computing*. Prentice Hall, Upper Saddle River, NJ, 1995.
13. W. Meira Jr. and T.J. LeBlanc and A. Poulos. Waiting Time Analysis and Performance Visualization in Carnival. In *ACM SIGMETRICS Symp. on Parallel and Distributed Tools*, May 1996.