Performance Analysis of Large-scale OpenMP and Hybrid MPI/OpenMP Applications with Vampir NG

Holger Brunst[1] and Bernd Mohr[2]

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

[2] Forschungszentrum Jülich, ZAM
Jülich, Germany
b.mohr@fz-juelich.de

Abstract. This paper presents a tool setup for comprehensive event-based performance analysis of large-scale OpenMP and hybrid OpenMP/MPI applications. The KOJAK framework is used for portable code instrumentation and automatic analysis while the new Vampir NG infrastructure serves as generic visualization engine for both OpenMP and MPI performance properties. The tools share the same data base which enables a smooth transition from bottleneck auto-detection to manual in-depth visualization and analysis. With Vampir NG being a distributed data-parallel architecture, large problems on very large scale systems can be addressed.

Keywords: Parallel Computing, OpenMP, Program Analysis, Instrumentation

1 Introduction

OpenMP is probably the most commonly used communication standard for shared-memory based parallel computing. The same applies to MPI when talking about parallel computing on distributed-memory architectures. Both approaches have widely accepted characteristics and qualities. OpenMP stands for an incremental approach to parallel computing which can be easily adapted to existing sequential software. MPI has a very good reputation with respect to performance and scalability on large problem and system sizes. Yet, it typically requires a thorough (re-) design of a parallel application. So far, most parallel applications are either native OpenMP or native MPI applications. With the emergence of large clusters of SMPs, this situation is changing. Clearly, hybrid applications that make use of both programming paradigms are one way to go. OpenMP has proven to work effectively on shared memory systems. MPI on the other hand can be used to bridge the gap between multiple SMP nodes. In a sense, this strategy follows the original idea of OpenMP which is to incrementally parallelize a given code.

In a hybrid scenario only minor changes (i. e. adding OpenMP directives) are required to achieve a moderate performance improvement while going beyond the memory boundaries of an SMP node requires more sophisticated techniques like message passing. Quite natural, a program that combines multiple programming paradigms is not easy to develop, maintain, and optimize. Portable tools for program analysis and debugging are almost essential in this respect. Yet, existing tools [1–3] typically concentrate on either MPI or OpenMP or exist for dedicated platforms only [4, 5]. It is therefore difficult to get on overall picture of a hybrid large-scale application. This paper presents a portable, distributed analysis infrastructure which enables a comprehensive support of hybrid OpenMP applications. The paper is organized as follows. The next section deals with collecting, mapping, and automatic classification of OpenMP/MPI performance data. Based hereon, Section 3 goes a step further and presents an architecture for in-depth analysis of large hybrid OpenMP applications. In Section 4 mixed mode analysis examples are given. Finally, Section 5 concludes the joint tool initiative.

2 The KOJAK Measurement System

The KOJAK performance-analysis tool environment provides a complete tracing-based solution for automatic performance analysis of MPI, OpenMP, or hybrid applications running on parallel computers. KOJAK describes performance problems using a high level of abstraction in terms of execution patterns that result from an inefficient use of the underlying programming model(s). KOJAK's overall architecture is depicted in Figure 1. The different components are represented as rectangles and their inputs and outputs are represented as boxes with rounded corners. The arrows illustrate the whole performance-analysis process from instrumentation to result presentation.

The KOJAK analysis process is composed of two parts: a semi-automatic multi-level instrumentation of the user application followed by an automatic analysis of the generated performance data. The first part is considered semi-automatic because it requires the user to slightly modify the makefile.

To begin the process, the user supplies the application's source code, written in either C, C++, or Fortran, to OPARI, which is a source-to-source translation tool. OPARI performs automatic instrumentation of OpenMP constructs and redirection of OpenMP-library calls to instrumented wrapper functions on the source-code level based on the POMP OpenMP monitoring API [6, 7]. This is done to capture OpenMP events relevant to performance, such as entering a parallel region. Since OpenMP defines only the semantics of directives, not their implementation, there is no equally portable way of capturing those events on a different level.

Instrumentation of user functions is done either during compilation by a compiler-supplied instrumentation interface or on the source-code level using TAU [8]. TAU is able to automatically instrument the source code of C, C++, and Fortran programs using a preprocessor based on the PDT toolkit [9].

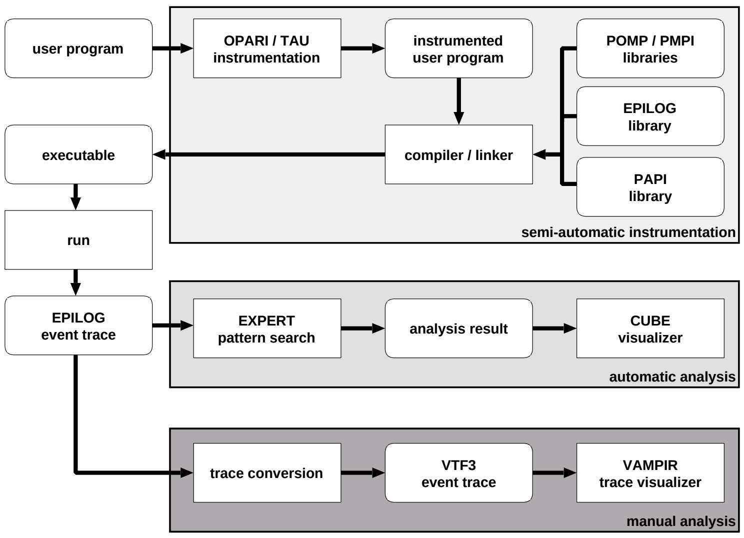


Fig. 1. KOJAK overall architecture.

Instrumentation for MPI events is accomplished with a wrapper library based on the PMPI profiling interface, which generates MPI-specific events by intercepting calls to MPI functions. All MPI, OpenMP, and user-function instrumentation calls the EPILOG run-time library, which provides mechanisms for buffering and trace-file creation. The application can also be linked to the PAPI library [10] for collection of hardware counter metrics as part of the trace file. At the end of the instrumentation process, the user has a fully instrumented executable.

Running this executable generates a trace file in the EPILOG format. After program termination, the trace file is fed into the EXPERT analyzer. (See [11] for details of the automatic analysis, which is outside of the scope of this paper.) In addition, the automatic analysis can be combined with a manual analysis using VAMPIR [12] or VAMPIR NG [13], which allows the user to investigate the patterns identified by EXPERT in a time-line display via a utility that converts the EPILOG trace file into the VAMPIR format.

3 The Distributed VAMPIR NG Program Analysis System

The distributed architecture of the parallel performance analysis tool VAMPIR NG [13] outlined in this section has been newly designed 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 running on a segment of a parallel production environment and a visualization client running on a potentially remote graphics workstation. Both components interact with each other over the Internet through a socket based network connection.

The major goals of the distributed parallel approach are:

1. Keep event trace data close to the location where they were created.
2. Analyze event data in parallel to achieve increased scalability (# of events $\sim 1{,}000{,}000{,}000$ and # of streams (processes) $\sim 10{,}000$).
3. Provide fast and easy to use remote performance analysis on end-user platforms.

Vampir NG consists of two major components: an analysis server (`vngd`) and a visualization client (`vng`). Each is supposed to run on a different machine. Figure 2 shows a high-level view of the overall software architecture. Boxes represent modules of the components whereas arrows indicate the interfaces between the different modules. The thickness of the arrows gives 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.

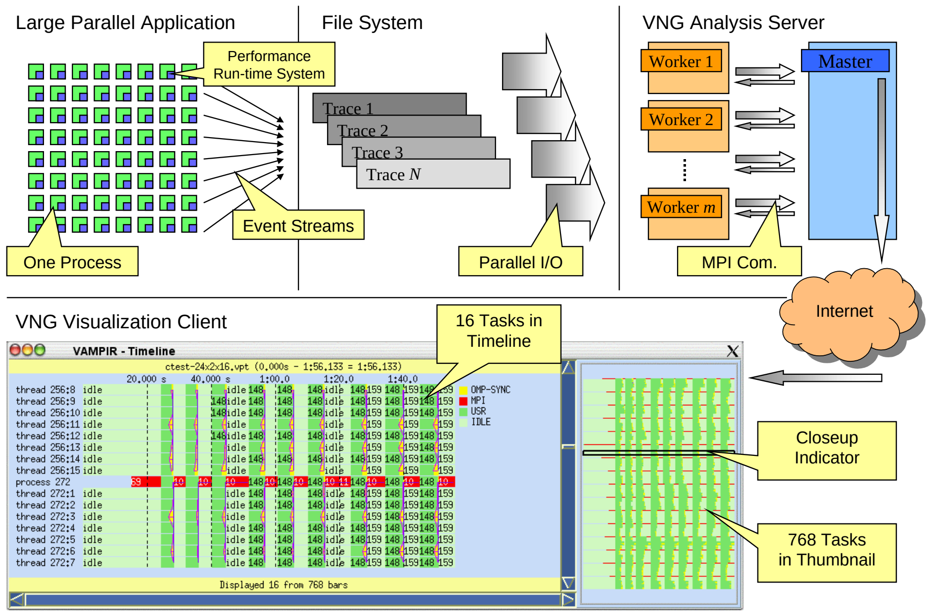


Fig. 2. Vampir NG Architecture Overview

In the top right corner of Figure 2 we can see the analysis server, which runs on a small interactive segment of a parallel machine. The reason for this is two-fold. Firstly, it allows the analysis server to have closer access to the trace data generated by an application being traced. Secondly, it allows the server to execute in parallel. Indeed, the server is a heterogeneous parallel program, implemented using MPI and pthreads, which uses a master/worker approach. The workers are responsible for storage and analysis of trace data. Each of them holds a part of the overall data to be analyzed. The master is responsible for the communication to the remote clients. He decides how to distribute analysis requests among the workers. Once the analysis requests are completed, the master merges the results into a single response package that is subsequently sent to the client.

The bottom half of Figure 2 depicts a snapshot of the VAMPIR NG visualization client which illustrates the timeline of an application run with 768 independent tasks. The idea is that the client is not supposed to do any time consuming calculations. It is a straightforward sequential GUI implementation with a look-and-feel very similar to performance analysis tools like Jumpshot [1], Paraver [4], VAMPIR [12], Paje [3], etc. For visualization purposes, it communicates with the analysis server according to the user's preferences and inputs. Multiple clients can connect to the analysis server at the same time, allowing simultaneous viewing of trace results.

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, the system should operate efficiently with only moderate latencies in both directions. This is basically due to the fact that only control information and condensed analysis results are to be transmitted over this link. Following this approach we comply with the goal of keeping the analysis on a centralized platform and doing the visualization remotely.

The big arrows connecting the program traces with the worker processes indicate high bandwidth. The major goal is to get fast access to whatever segment of the trace data the user is interested in. High bandwidth is basically 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.

4 In-Depth Analysis of Large-Scale OpenMP Programs

The KOJAK analysis infrastructure primarily addresses automatic problem detection. Previously collected trace data is searched for pre-defined problems [14]. The results are displayed in a hierarchical navigator tool which provides links to the respective source code locations. This approach is very effective as it does not require complicated user interactions or expert knowledge. Yet, it is limited to known problems and sometimes the real cause of a phenomenon remains obscure.

With the help of the collected trace data it is even possible to go into further detail. The measurement system in KOJAK supports the generation of VAMPIR NG compatible traces which can be examined according to the hints made by the EXPERT tool.

Having access to the same central data base, VAMPIR NG offers a rich set of scalable remote visualization options for arbitrary program phases. In the following, the sPPM benchmark code [15] will serve as example application demonstrating combined OpenMP and MPI capabilities of VAMPIR NG. The code has been equipped with OpenMP directives and was executed on 128 MPI tasks with eight OpenMP threads each. The test platform was a Power4-based, 30-way SMP cluster system. Altogether, 1024 independent event data streams had to be handled.

4.1 Custom Profiles

Vampir NG supports a grouping concept for flat profile charts *à la* gprof. The summarized information reflects either the entire program run or a time interval specified by the user. The information provided is not limited to functions. Depending on the application, OpenMP and MPI related information like message sizes, counter values etc. can be summarized additionally.

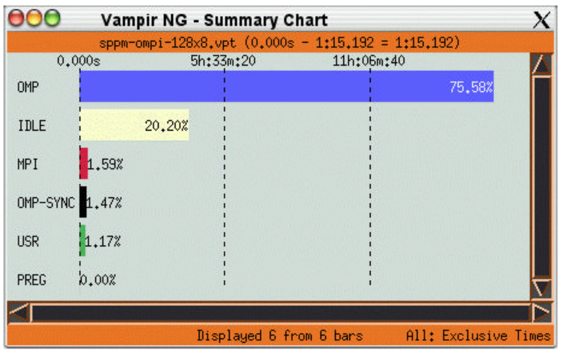


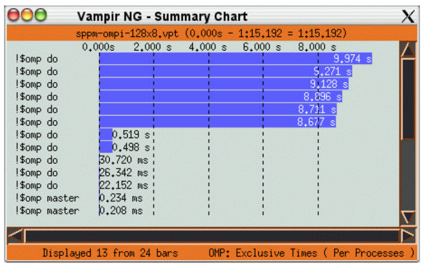
Fig. 3. Summary profile of a sPPM run on 1024 processes/threads.

Figure 3 depicts a summary profile of the full program run which lasted 1:15 minutes. Exclusive timing information is shown as percentages relative to the overall accumulated run-time. The KOJAK OpenMP instrumentation creates the following six default sub-groups of program states:

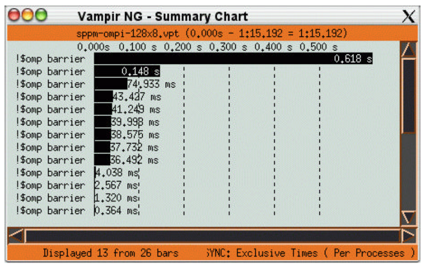
1. *USR:* Code regions which are not parallelized with OpenMP
2. *OMP:* OpenMP parallel execution
3. *OMP-SYNC:* OpenMP implicit and explicit barrier synchronization
4. *PREG:* OpenMP thread startup and termination
5. *MPI:* MPI communication and synchronization
6. *IDLE:* Idle OpenMP threads

Quite obviously, the application spends too much time (20%) doing nothing (*IDLE*). Its cause is unknown. We will come to this phenomenon in the next section. 75% percent of the runtime is spent in OpenMP parallel code. The remaining five percent are spent in MPI and OpenMP synchronization code.

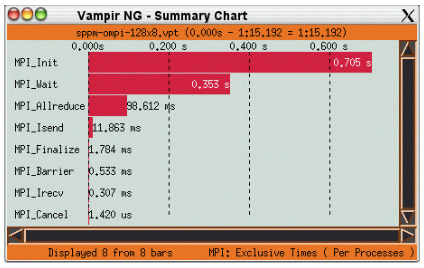
The same display can be used to further analyze the six sub-groups of program states. Figures 4(a) to 4(d) depict summary profiles for the states in *OMP*, *OMP-SYNC*, and *MPI* respectively. From Figure 4(a) we can read that our



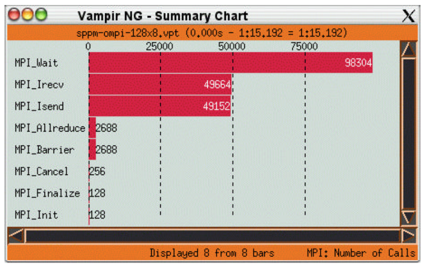
(a) OpenMP Loop Profile



(b) OpenMP Synchronization Profile



(c) MPI Profile



(d) MPI Invocation Profile

Fig. 4. Adaptive VAMPIR NG Profiles

application has twelve major OpenMP do-loops from which six contribute with more than 8.5 seconds each (per process). Only these loops should be considered for further optimization. In Figure 4(b), OpenMP synchronization overhead is depicted. The first two barrier constructs are interesting candidates to be analyzed in further detail. Their active phases during run-time can be located with a navigator display similar to traditional timelines. Depending on user defined queries, the "navigator" (not depicted) highlights selected states only. Figure 4(c) provides information on how MPI is used in this code. Synchronization is the dominant part. Last not least, the number of MPI function calls as depicted in Figure 4(d) tells us that approximately 100,000 messages are exchanged altogether. Considering the 128 MPI processes involved and the short amount of time spent in MPI, this information is more interesting for code debugging than for optimization.

4.2 Hierarchical Timelines

Sometimes, adaptive profiles are not sufficient for understanding an application's inner working. An event timeline as depicted in Figure 5 is very useful to obtain a better understanding. The event timeline visualizes the behavior of individual processes over time. Here, the horizontal axis reflects time, while the vertical axis identifies the process. Colors are used to represent the already mentioned sub-groups of program states. Apparently, navigating on the data of 1024 independent processing entities is a rather complex task. Therefore, an overview

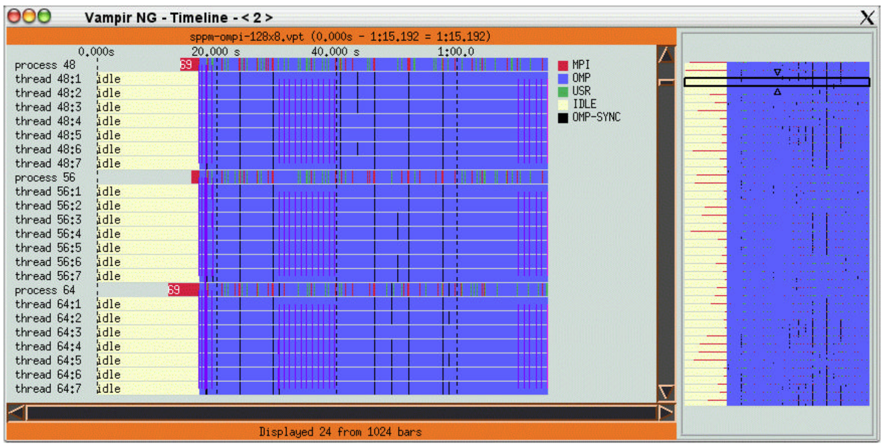


Fig. 5. Event timeline of a sPPM run on 128x8 processes/threads.

of the full set of processes and threads is depicted on the right hand side. The rectangular marking identifies the selection of the trace that is depicted in full detail on the left hand side (process 48 to process 64).

Having access to this kind of application overview, it quickly becomes evident where the 20% idle-time in the profile comes from. Due to the large number of processes and OpenMP threads, the execution platform needs a substantial time (approximately 17 seconds) to spawn the full application. Having a closer look at the startup phase (light/beige section) reveals that the spawning of the MPI processes (MPI_Init) is varying a lot in time. Apparently, MPI_Init has to wait until all processes are up and running before it lets the processes start their individual tasks.

We will now take a closer look at the locations where OpenMP and MPI synchronization takes place. Figure 6 illustrates a section which includes the barrier that has been mentioned earlier in Section 4.1. The program highlights the selected OpenMP barrier with bold dotted lines. From this kind of display we can learn many things, one of which is that MPI and OpenMP synchronization have to work in close cooperation in mixed codes. This particular example shows how MPI collective communication is carried out on the master threads only (which is a common MPI constraint) while OpenMP barriers guarantee that the thread parallelism is not continuing to process inconsistent data. Figure 7 illustrates the differences between an MPI communication process and a respective OpenMP thread by means of a single-task-timeline showing the detailed function call-path.

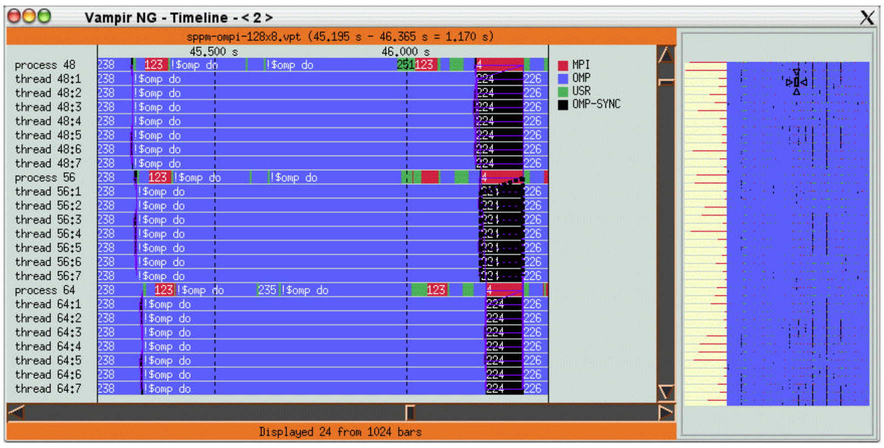
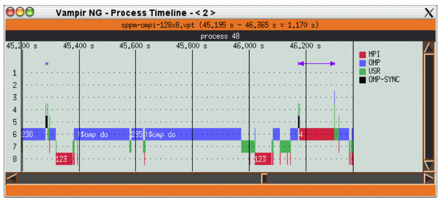
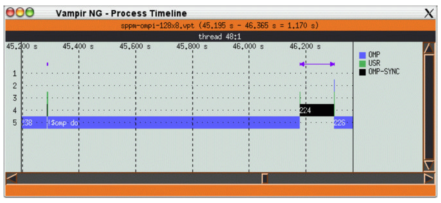


Fig. 6. Synchronization of sPPM run on 128x8 processes/threads.


Vampir NG - Process Timeline - < 2 >
sppm-ompi-128x8.vpt (45.195 s - 46.365 s = 1.170 s)
process 48
MPI
OMP
USR
OMP-SYNC

(a) Single Timeline – MPI Process


Vampir NG - Process Timeline
sppm-ompi-128x8.vpt (45.195 s - 46.365 s = 1.170 s)
thread 48:1
OMP
USR
OMP-SYNC

(b) Single Timeline – OpenMP Thread

Fig. 7. Hybrid MPI/OpenMP Synchronization

5 Conclusion

Data distribution and synchronization in large-scale OpenMP and hybrid MPI/OpenMP applications can lead to critical performance bottlenecks. Profiling alone can hardly help to identify the real cause of problems that fall into this category. Event-based approaches on the other hand are known to generate large volumes of data. In this difficult situation, automatic event-based performance analysis has the potential to quickly detect most known synchronization problems. When dealing with uncommon features or for detailed examination of already detected problems, manual analysis has certain advantages due to human intuition and pattern recognition capabilities. Therefore, an incremental approach with profiling and automatic techniques forming a solid starting point and event-based analysis being used for more detailed questions is advisable. PAPI [10] counter support in both tools completes the detailed performance examination. Finally, our work has shown that both approaches can be effectively combined in a portable way.

References

1. Zaki, O., Lusk, E., Gropp, W., Swider, D.: Toward scalable performance visualization with Jumpshot. High Performance Computing Applications **13** (1999) 277–288
2. Rose, L.D., Zhang, Y., Reed, D.A.: Svpablo: A multi-language performance analysis system. In: 10th International Conference on Computer Performance Evaluation - Modelling Techniques and Tools - Performance Tools '98, Palma de Mallorca, Spain (1998) 352–355
3. de Kergommeaux, J.C., de Oliveira Stein, B., Bernard, P.: Pajè, an interactive visualization tool for tuning multi-threaded parallel applications. Parallel Computing **26** (2000) 1253–1274
4. European Center for Parallelism of Barcelona (CEPBA): Paraver - Parallel Program Visualization and Analysis Tool - Reference Manual. (2000) http://www.cepba.upc.es/paraver.
5. Intel: Intel thread checker (2005) http://www.intel.com/software/products/threading/tcwin.
6. Mohr, B., Mallony, A., Hoppe, H.C., Schlimbach, F., Haab, G., Shah, S.: A Performance Monitoring Interface for OpenMP. In: Proceedings of the fourth European Workshop on OpenMP - EWOMP'02. (September 2002)
7. Mohr, B., Malony, A., Shende, S., Wolf, F.: Design and Prototype of a Performance Tool Interface for OpenMP. The Journal of Supercomputing **23** (2002) 105–128
8. Bell, R., Malony, A.D., Shende, S.: A Portable, Extensible, and Scalable Tool for Parallel Performance Profile Analysis. In: Proceedings of Euro-Par 2003. (2003) 17–26
9. Lindlan, K.A., Cuny, J., Malony, A.D., Shende, S., Mohr, B., Rivenburgh, R., Rasmussen, C.: A Tool Framework for Static and Dynamic Analysis of Object-Oriented Software with Templates. In: Proceedings of Supercomputing 2000. (November 2000)
10. Browne, S., Dongarra, J., Garner, N., Ho, G., Mucci, P.: A Portable Programming Interface for Performance Evaluation on Modern Processors. The International Journal of High Performance Computing Applications **14** (2000) 189–204
11. Wolf, F., Mohr, B.: Automatic Performance Analysis of Hybrid MPI/OpenMP Applications. Journal of Systems Architecture, Special Issue 'Evolutions in parallel distributed and network-based processing' **49** (2003) 421–439
12. Nagel, W., Arnold, A., Weber, M., Hoppe, H.C., Solchenbach, K.: Vampir: Visualization and Analysis of MPI Resources. Supercomputer **12** (1996) 69–80
13. Brunst, H., Nagel, W.E., Malony, A.D.: A distributed performance analysis architecture for clusters. In: IEEE International Conference on Cluster Computing, Cluster 2003, Hong Kong, China, IEEE Computer Society (2003) 73–81
14. Fahringer, T., Gerndt, M., Riley, G., Träff, J.L.: Formalizing OpenMP performance properties with ASL. Lecture Notes in Computer Science **1940** (2000) 428
15. Lawrence Livermode National Laboratory: the sPPM Benchmark Code. (2002) http://www.llnl.gov/asci/purple/benchmarks/limited/sppm/.