FORSCHUNGSZENTRUM JÜLICH GmbH
Zentralinstitut für Angewandte Mathematik
D-52425 Jülich, Tel. (02461) 61-6402

Interner Bericht

Automatic Performance Analysis of SMP Cluster Applications

Felix Wolf, Bernd Mohr

FZJ-ZAM-IB-2001-05

August 2001

(letzte Änderung: 13.08.2001)

Automatic Performance Analysis of SMP Cluster Applications

Felix Wolf
f.wolf@fz-juelich.de

Bernd Mohr
b.mohr@fz-juelich.de

Forschungszentrum Jülich (Germany)
Zentralinstitut für Angewandte Mathematik

August 13, 2001

Abstract

The EXPERT tool environment provides a complete tracing-based solution for automatic performance analysis of MPI, OpenMP, or hybrid applications running on SMP cluster machines.

EXPERT describes performance problems using a high level of abstraction in terms of common situations that result from an inefficient use of the underlying programming model(s). The set of supported problems is extensible and can be custom tailored to application-specific needs.

The analysis is carried out along three interconnected dimensions: class of performance behavior, call tree position, and thread of execution. Each dimension is arranged in a hierarchy, so that the user can view the behavior on varying levels of detail. All three dimensions are interactively accessible using a scalable but still accurate tree display. Colors provide assistance in finding interesting nodes even in case of large trees.

1 Introduction

SMP cluster technology offers the opportunity to produce scalable high performance architectures at a cost that will make such environments attainable for a broader community of users also including small and medium sized enterprises. However, the reduced cost of these systems comes at the price of a complex hierarchical architecture, which often demands the concurrent usage of more than one parallel programming model in the same application.

As a consequence, performance optimization becomes more difficult and creates a need for advanced performance tools that are custom tailored for this class of computing environments. In particular automatic tools are neccessary in the face of the large amount of performance data often produced on such machines. Current state-of-the-art tools such as VGV [6] visualize the collected performance data in a scalable way but still require the user to find out where the performance problems are located and what their reasons are.

In contrast, the EXPERT tool environment[1] is able to automatically locate and explain performance problems. EXPERT is based on event tracing and allows the analysis of MPI

[1] The work on EXPERT is carried out as a part of the KOJAK project [5, 9] and is embedded in the ESPRIT working group APART [18].

[14], OpenMP [17], or hybrid applications running on SMP clusters as well as on more traditional non-SMP and non-cluster systems.

EXPERT investigates the performance behavior along three dimensions: class of performance behavior, position within the dynamic call tree, and location (e.g., node or process). The comprehensive behavioral classification used by EXPERT provides the ability to explain problems intelligibly in terms of common situations that result from a non-optimal usage of the programming model they are related to. In addition, it is possible to integrate application specific classifications by using appropriate extension mechanisms.

Each of the analyzed dimensions is organized in a hierarchy. The user can access these hierarchies interactively using *weighted trees*, which provide a intuitive graphical presentation of performance behavior on different levels of detail along each dimension. The weighted trees indicate the severity of a performance problem using colors, which allows maxima to be easily identified even in a large set of values. The trees are interconnected, so that the user can view one dimension with respect to a selection in another dimension.

The remainder of this article is organized as follows: First, we describe the overall architecture of our tool environment in the next section. Then, in Section 3 we deal with event trace generation. In Section 4 we present the abstraction mechanism that is used to specify complex situations representing inefficient performance behavior. After that, we introduce the actual analysis component and how it can be extended to deal with application specific requirements in Section 5. Section 6 proves our concept by applying it to two real-world examples. Finally, we consider related work and conclude the paper.

2 Overall Architecture

The whole tool environment is made up of three components.

- The EXPERT performance tool (Extensible PERformance Tool)
- The EARL trace analysis language (Event Analysis and Recognition Language)
- The EPILOG tracing library (Event Processing, Investigating and LOGging)

The EPILOG library is used to generate event traces from parallel applications in the EPILOG binary trace data format. The resulting trace files serve as input for the EXPERT performance tool, which carries out the actual performance analysis and presents its results to the user. To make the analysis process simple and easy to extend, EXPERT uses EARL to map the event trace onto a higher level of abstraction. Using these abstractions, EXPERT is able to easily identify complex compound events in the event trace that represent common situations of inefficient behavior.

Early prototypes of EARL [20] and EXPERT [21] provided only limited assistance in the analysis of MPI applications. This article describes a substantial redesign and many enhancements including support for MPI collective communication and OpenMP as well as a graphical presentation of performance behavior.

3 Performance Data

Our analysis process relies on event traces as performance data source, because event traces preserve the temporal and spatial relationships among individual events, which

are necessary to prove many interesting performance properties for the application being investigated.

The event traces used in our approach are compliant with the newly designed portable EPILOG binary trace data format. In contrast to traditional trace data formats, the EPILOG format is suitable to represent the executions of MPI, OpenMP, or hybrid parallel applications being distributed across one or more (possibly large) clusters of SMP nodes. It maps events onto their location within the hierarchical hardware as well as to their process and thread of execution. It supports storage of all necessary source code and call site information, hardware performance counter values, and marking of collectively executed operations for both MPI and OpenMP.

The user can generate event traces for C, C++, and Fortran applications just by linking to the EPILOG trace library. To intercept function calls and returns, the library uses the internal profiling interface of the PGI compiler suite [8] being installed on our Linux SMP cluster testbed ZAMpano [11]. A portable source code instrumenter for Fortran, C, and C++ based on the PDT toolkit [13] will be available soon, too. The implementation of EPILOG is thread safe, a necessary feature not present in most traditional tools.

MPI-specific events are generated by a appropriate wrapper function library utilizing the MPI standard profiling interface. In addition, OPARI [16], a portable tool for automatic instrumentation of OpenMP constructs on the source code level, allows OpenMP-specific events to be traced and linked back to the source code.

4 Abstraction Mechanisms

EARL maps an EPILOG trace file to the EARL event trace model. The EARL event trace model provides abstractions that allow compound events representing inefficient behavior to be easily described.

The model considers an event trace as a chronologically sorted sequence of primitive events. Depending on the event type, each event is characterized by a set of attributes. The event types are organized in a hierarchy. There are programming-model–independent event types representing simple region enters and exits. Types indicating point-to-point and collective communication are used to map the MPI model. OpenMP event types comprise fork and join operations, lock synchronization operations, and - similar to MPI - an event type indicating the collective execution of parallel constructs.

Common to all event types are attributes indicating the position within the event trace, the event location, and the time stamp. The position is used as a unique identifier, by which events can be referenced. The location of an event is a tuple containing the machine, the node within the machine, the process, and the thread. The event type itself can also be accessed as value of an attribute.

Additionally, EARL provides two types of abstractions on top of the basic part of the model:

- System states
- Pointer attributes

System states map individual events onto a set of events that represent one aspect of the parallel system's execution state at the moment when the event happens. System states include the *region stack* containing all events of entering the region instances, in

which the program is currently executing, or the *message queue* containing all events of sending a message that is currently in transfer. Other system states provide all events of a MPI collective or OpenMP parallel operation just being completed.

Pointer attributes connect related events, so that you can define compound events along a path of related events. EARL provides attributes pointing from an arbitrary event to the region-enter event of the current region, from a message-receipt event to the corresponding send event, from a join event to its corresponding fork event, and from a lock event to the preceding lock event that modifies the same lock.

An essential part of the model is the dynamic call tree, which is computed from all region-enter events. Each enter event has exactly one associated node in the tree, and thus the call tree partitions these events into equivalence classes of events that are associated with the same node in the call tree. Without loss of generality, the least recent event in each class is chosen to be its representative. As an additional pointer attribute, EARL provides a link from each enter event to its representative. Thus, we have a simple means to associate a performance relevant compound event with the corresponding execution phase of the parallel program.

EARL is implemented as a C++ class, whose interface is embedded in the Python scripting language. The class provides efficient random access to the events of the trace and allows the utilization of the abstractions mentioned in this section. [22] contains a more detailed description of the underlying model.

5 Analysis Process

The design of the analysis process is based on the specifications and terminology as presented in [4]. The analysis process attempts to prove *performance properties* for one execution of a parallel application and to classify them according to their influence on the performance. A performance property characterizes a class of performance behavior. One can prove it by evaluating an associated condition based on the events in the trace file. A *severity* measure indicates the influence of a performance property on the performance behavior and allows the comparison of different performance properties for one run of an application.

The analysis process is carried out by the EXPERT component. It is implemented in Python using Tk for the graphical user interface and EARL for trace access. Its architecture is based on the idea of separating the analysis process from the specification of the performance properties; that is, the performance properties are not hard-coded into the EXPERT tool but specified separately.

5.1 Specification of Performance Properties

The performance properties are specified in form of *patterns*. Patterns are Python classes, which are responsible for detecting compound events indicating inefficient behavior. They provide a common interface making them exchangeable from the perspective of the tool. The specifications use the abstractions provided by EARL and, for this reason, are very simple.

The analysis process follows an event driven approach. EXPERT walks sequentially through the event trace and invokes for each single event call-back methods to the pattern instances and supplies the event as an argument. A pattern can provide a different call-back method for each event type. The call-back method itself then tries to locate

a compound event representing an inefficiency, thereby following links (i.e., pointer attributes) emanating from the supplied event or investigating system states. This mechanism allows the simple specification of very complex performance relevant situations and an explanation of inefficiency that is very close to the terminology of the programming model.

The common interface also provides a method to launch a configuration dialog for the input of pattern-specific parameters before the analysis process as well as a method to launch a presentation dialog for the display of pattern-specific results afterward, which allows the treatment of pattern-specific performance criteria.

EXPERT organizes the performance properties in a hierarchy. The upper levels of the hierarchy (i.e., those that are closer to the root) correspond to more general behavioral aspects such as time spent in MPI functions. The deeper levels correspond to more specific situations such as time lost due to blocking communication.

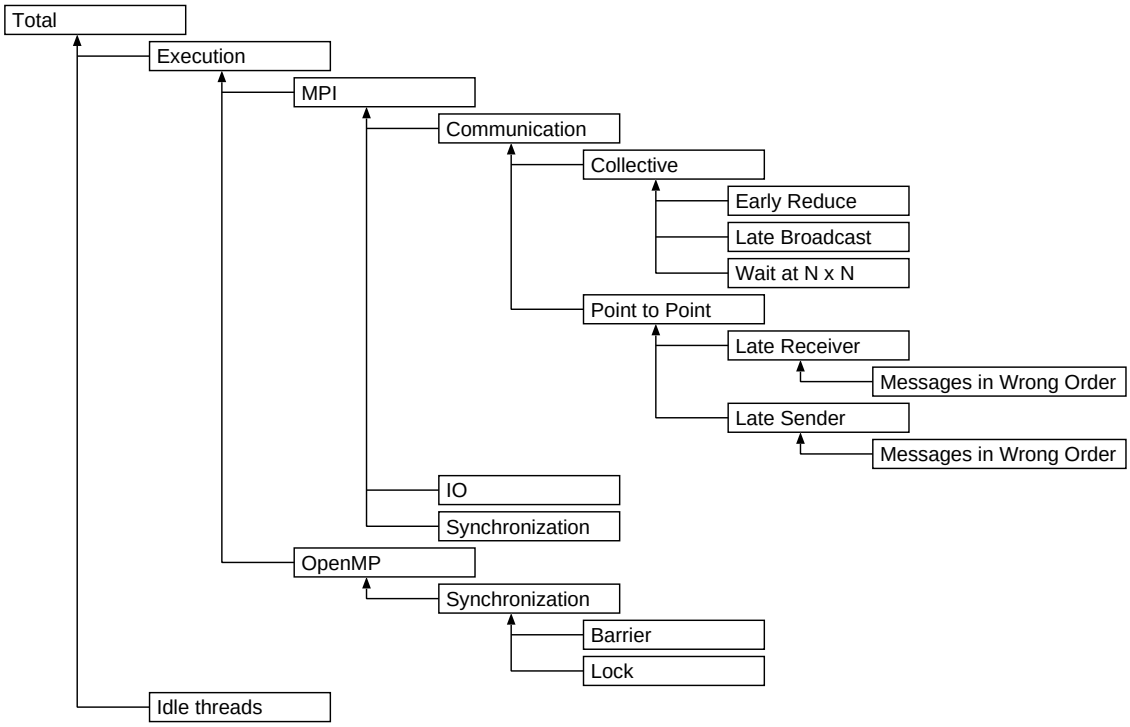

Figure 1: Hierarchy of performance properties.

Figure 1 shows the complete hierarchy of performance properties being currently supported by EXPERT. We shall briefly discuss some of the most interesting ones in Section 5.1.1 and 5.1.2.

5.1.1 Examples of MPI Performance Properties

Late Sender This property refers to the time wasted, when a call to a blocking receive operation (e.g, `MPI_RECV` or `MPI_Wait`) is posted before the corresponding send operation is executed.

Late Receiver This property refers to the inverse case. A send operation blocks until

the corresponding receive operation is called. This can happen for several reasons. Either the MPI implementation is working in synchronous mode by default or the size of the message to be sent exceeds the available MPI-internal buffer space and the operation blocks until the data is transferred to the receiver.

Messages in Wrong Order This property, which has been motivated by [7], deals with the problem of passing messages out of order. The sender is sending messages in a certain order, but the receiver is expecting the arrival in another order. The implementation locates such situations by querying the message queue each time a message is received and by looking for older messages with the same target as the current message. This situation can be a specialization of either *Late Sender* or *Late Receiver*.

Wait at N x N Collective communication operations that send data from all processes to all processes exhibit an inherent synchronization, that is, no process can finish the operation until the last process has started. The time until all processes have entered the operation is measured and used to compute the severity. Note that this property requires to identify all parts of a collective-operation instance in the event stream.

5.1.2 Examples of OpenMP Performance Properties

Barrier Synchronization The time spent on implicit (compiler-generated) or explicit (user-specified) OpenMP barrier synchronization.

Lock Synchronization The time a thread waits for a lock that is owned by another thread.

Idle Threads Idle times on processors caused by sequential execution before or after an OpenMP parallel region.

5.2 Representation of Performance Behavior

Each applied pattern instance computes a two-dimensional severity matrix, which contains the severity as a function of the node in the dynamic call tree and the location. Thus, we can represent the complete performance behavior using a three-dimensional matrix, where each cell contains the severity for a specific performance property, call tree node, and location.

The first dimension describes the kind of inefficient behavior. The second dimension describes both its source code location and the execution phase during which it occurs. Finally, the third dimension gives information on the distribution of performance losses across different processes or threads, which allows to draw additional conclusions (e.g., load imbalance, see also [21]).

In addition, each of the dimensions is arranged in a hierarchy: the performance properties in a hierarchy of general and more specific ones, the call tree nodes in their evident hierarchy, and the locations in a hierarchy consisting of the levels machine, node, process, and thread. Thus, it is possible to analyze the behavior on different levels of granularity.

5.3 Presentation of Performance Behavior

The user can interactively access each of the hierarchies constituting a dimension of performance behavior using *weighted trees*, a display technique developed for EXPERT. A weighted tree is a tree browser that labels each node with a weight. EXPERT uses as weight a percentage of the application's total CPU allocation time (e.g., the percentage of time spent in a subtree of the call tree). The weight that is actually displayed depends on the state of the node, that is, whether it is expanded or collapsed. The weight of a collapsed node represents the whole subtree associated with that node, whereas the weight of an expanded node represents only the fraction that is not covered by its descendants because the weights of its descendants are now displayed separately. This allows the analysis of performance behavior on different levels of granularity.

For example, the call tree may have a node *main* with two children *foo* and *bar* (Fig 2). In the collapsed state, this node is labeled with the weight representing the time spent in the whole program. In the expanded state it displays only the fraction that is spent neither in *foo* nor in *bar*.

Figure 2: Weighted tree in collapsed and expanded state.

The weight is displayed simultaneously using both a numerical value as well as a colored icon. The color is taken from a spectrum representing the whole range of possible weights (i.e., 0 - 100 percent). Colors enable the easy identification of nodes of interest even in a large tree, whereas the numerical values enable the precise comparison of individual weights.

The weighted trees of the different analysis dimensions are interconnected, so that the user can display the call tree with respect to a particular performance property, and the distribution across the locations with respect to a particular node in call tree (Fig. 3).

Weighted trees provide a uniform and very intuitive display for each of the analyzed dimensions. Once the user is familiar with this kind of display, she can investigate the performance behavior in a scalable but still accurate way along all its interconnected dimensions.

5.4 Extension Mechanisms

EXPERT provides a large set of built-in performance properties, which cover the most frequent inefficiency situations. But sometimes the user may wish to consider application-specific metrics such as iterations or updates per second. In this case, he can simply write another pattern class that implements his own application-specific performance property according to the common interface of all pattern classes, and place it into a plug-in module.

At startup time, EXPERT dynamically queries the module's name space and looks for newly inserted patterns from which it is now able to build instances. The new patterns are integrated into the graphical user interface and can be used like the predefined ones.

6 Example

We tested our environment for two real-world applications on our SMP cluster testbed [11] having eight nodes with four CPUs each. The first one, TRACE, is a pure MPI application. The second one, REMO is a hybrid MPI/OpenMP application. CPU allocation was done in a way, such that no CPU was assigned to more than one computational thread or single-threaded process.

6.1 TRACE

TRACE [10] simulates the subsurface water flow in variable saturated porous media. The parallelization is based on a parallelized CG algorithm. We ran the application using eight nodes with two process per node (8 processes x 2 processes).

Using the performance property view (Fig. 3, left), it was easy to see that most of the time used for communication routines was spent on waiting due to the situations *Wait at N x N* and *Late Sender*, which are described in Section 5.1.1.

Using the call tree view (Fig. 3, middle), we quickly located two paths that are major sources of the previously identified performance problems:

(a) Wait at N x N:
`trace` → `cgiteration` → `parallelcg` → `paralleldotproduct`
→ `globalsum_r1` → `MPI_Allreduce`

(b) Late Sender:
`trace` → `cgiteration` → `parallelcg` → `parallelfemultiply`
→ `exchangedata` → `exchangebufferswf` → `mrecv` → `MPI_Recv`

The numerical results of our analysis are listed in Table 1. The values represent percentages of total CPU allocation time. The first column refers to the whole program, whereas the second and third column refers only to the call paths listed above. The first row corresponds to the time spent in MPI communication statements. For the two call paths this is just the time needed for the completion of the specific MPI calls at their end. The second and third row correspond to the waiting times caused by *Wait at N x N* and *Late Sender* situations.

Table 1: Performance problems found in TRACE in percent of total CPU allocation time.

Performance Property	Whole program	(a)	(b)
Communication	16.7	4.5	8.5
Wait at N x N	3.8	3.6	
Late Sender	8.1		7.0

In addition, one can see (Fig. 3, right) that the idle times for *Late Sender* in call path (b) expose an uneven but still symmetric distribution across the different processes. Obviously, there is a correlation to the pattern of communication among different subdomains, which the user should examine in a next step.

6.2 REMO

REMO [2] is a weather forecast application of the DKRZ (Deutsches Klima Rechenzentrum). It implements a hydrostatic limited area model, which itself is based on the *Deutschland/Europa* weather forecast model of the German Weather Service (Deutscher Wetterdienst (DWD)). For this paper we analyzed an early experimental MPI/OpenMP version of the production code. The application was executed on four nodes with one processes per node and four threads per process (4 processes x 4 threads).

Figure 4 shows that one half of the total CPU allocation time is idle time, which has been occurred as a result of OpenMP sequential execution outside of parallel regions. Although during this time period the idle threads actually do not execute any code, the time is mapped onto the call tree according to the corresponding master thread. That is, EXPERT assumes that outside parallel regions the slave threads "execute" the same code as their master thread. This method of call tree mapping helps to identify parts of the call tree that might be optimized in order to reduce the amount sequential execution. In case of REMO, we identified the following paths as major sources of idle times:

(a) `remo` → `remorg` → `ec4org` → `progec4` → `phyec`

(b) `remo` → `remorg` → `ec4org` → `progec4` → `progexp`

The numerical results are listed in Table 2. The values represent percentages of total CPU allocation time lost as a result of performance property *Idle Threads*. The first column refers to the whole program, whereas the second and third column refers only to the call paths listed above.

Table 2: Performance problems found in REMO in percent of total CPU allocation time.

Performance Property	Whole program	(a)	(b)
Idle Threads	50.0	13.4	10.3

7 Related Work

Miller and associates [15] developed automatic on-line performance analysis according to the W^3 Search Model in the well-known Paradyn project. In contrast to our approach, the W^3 model describes performance behavior along the dimensions performance problem, program resource, and time. Performance problems are expressed in terms of a threshold and one or more metrics such as CPU time, blocking time, message rates, I/O rates, or number of active processors. Program resources include both hardware resources such as processor nodes or disks and software resources such as procedures, message channels, or barrier instances. The time dimension tries to divide the program execution into phases with certain performance characteristics.

Espinosa [3] implemented an automatic trace analysis tool KAPPA-PI for evaluating the performance behavior of MPI and PVM message passing programs. Here, behavior classification is carried out in two steps. At first, a list of idle times is generated from the raw trace file using a simple metric. Then, based on this list, a recursive inference process continuously deduces new facts on an increasing level of abstraction. Finally,

recommendations on possible sources of inefficiencies are built from the facts being proved on the one hand and from the results of source code analysis on the other hand.

Vetter [19] performs automatic performance analysis of MPI point-to-point communication based on machine learning techniques. He traces individual message passing operations and then, classifies each individual communication event using a decision tree. The decision tree has been previously trained by microbenchmarks that demonstrate both efficient as well as inefficient performance behavior. The ability to adapt to a special target system's configuration helps to increases the technique's predictive accuracy.

Hoeflinger and associates [6] integrated the VAMPIR [1] event trace browser with the GuideView [12] OpenMP analyzer to a new tool VGV for MPI/OpenMP applications. VGV provides a scalable time-line view of an event traces, which highlights sections of multi-threaded program execution. The user can select individual sections and analyze them using a graphical profile display. However, as far as we know VGV does not support automatic behavioral classification.

8 Conclusion

The EXPERT tool environment provides a complete but still extensible solution for automatic performance analysis of MPI, OpenMP, or hybrid application running on SMP cluster machines.

EXPERT represents performance properties on a very high level of abstraction that goes beyond simple metrics and provides the ability to explain performance problems in terms of the underlying programming model(s). The performance property specifications are embedded in a flexible architecture and can be extended and custom tailored to application-specific needs.

The performance behavior is presented along three interconnected dimensions: class of performance behavior, position within the running program and thread of execution. The last dimensions allows even the effects of different communication patterns among subdomains to be investigated.

Each dimension is arranged in a hierarchy, so that the user can view the behavior on varying levels of detail. In particular the hierarchical structure of hybrid applications and SMP cluster hardware is reflected this way. The user can access all three dimensions interactively using a scalable but still accurate tree display. Colors make it easy to identify interesting nodes even in case of large trees.

EXPERT is well suited to analyze a single trace file. But the development process of parallel applications often demands for comparison of trace files representing different execution configurations or development versions. For the future, we intend to integrate mechanisms for comparative performance analysis. In addition, we plan to improve our result presentation by integrating it with an event trace browser such as VAMPIR [1] to visualize instances of an performance problem as time lines and adding source code displays. Finally, we will work on further improving and completing our performance properties catalog.

9 Acknowledgements

We would like to thank all our partners in the ESPRIT working group APART for their contributions to this topic. We also would like to thank Arpad Kiss for providing the basic

tree browser implementation, DKRZ and ICG-4 for giving us access to their applications, and Wolfgang Frings and Reiner Vogelsang for helping us in conducting our experiments. Finally, we would like to thank Craig Soules for his helpful suggestions on the language.

References

[1] A. Arnold, U. Detert, and W.E. Nagel. Performance Optimization of Parallel Programs: Tracing, Zooming, Understanding. In R. Winget and K. Winget, editors, *Proc. of Cray User Group Meeting*, pages 252–258, Denver, CO, March 1995.

[2] T. Diehl and V. Gülzow. Performance of the Parallelized Regional Climate Model REMO. In *Proc. of the Eighth ECMWF Workshop on the Use of Parallel Processors in Meterology.*

[3] A. Espinosa. *Automatic Performance Analysis of Parallel Programs.* PhD thesis, Universitat Autonoma de Barcelona, September 2000.

[4] T. Fahringer, M. Gerndt, and G. Riley. Knowledge Specification for Automatic Performance Analysis. Technical report, ESPRIT IV Working Group APART, November 1999.

[5] M. Gerndt, B. Mohr, M. Pantano, and F. Wolf. Performance Analysis for CRAY T3E. In IEEE Computer Society, editor, *Proc. of the 7th Euromicro Workshop on Parallel and Disributed Pocessing(PDP'99)*, pages 241–248, 1999.

[6] J. Hoeflinger, B. Kuhn, W. Nagel, P. Petersen, H. Rajic, S. Shah, J. Vetter, M. Voss, and R. Woo. An Integrated Performance Visualizer for MPI/OpenMP Programs. Accepted for the 3rd European Workshop on OpenMP (EWOMP 2001), September 2001.

[7] J. K. Hollingsworth and M. Steele. Grindstone: A Test Suite for Parallel Performance Tools. Computer Science Technical Report CS-TR-3703, University of Maryland, October 1996.

[8] Portland Group Inc. Private communication.

[9] Research Centre Jülich. KOJAK (Kit for Objective Judgement and Knowledge-based Detection of Performance Bottlenecks). http://www.fz-juelich.de/zam/kojak/.

[10] Research Centre Jülich. Solute Transport in Heterogeneous Soil-Aquifer Systems. http://www.kfa-juelich.de/icg/icg4/Groups/Pollutgeosys/trace_e.html.

[11] Research Centre Jülich. ZAMpano (ZAM Parallel Nodes). http://zampano.zam.kfa-juelich.de/.

[12] KAI Software, a division of Intel Americas. *Guide View Performance Analyzer*, 2001. http://www.kai.com/parallel/kapro/guideview.

[13] K. A. Lindlan, J. Cuny, A. D. Malony, B. Mohr, R. Rivenburgh, and C. Rasmussen. A Tool Framework for Static and Dynamic Analysis of Object-Oriented Software with Templates. In *Proc. of the Conference on Supercomputers*, Dallas, Texas, November 2000.

[14] Message Passing Interface Forum. MPI Documents, Juli 1997. http://www.mpi-forum.org/docs/docs.html.

[15] B. P. Miller, M. D. Callaghan, J. M. Cargille, J. K. Hollingsworth, R. B. Irvine, K. L. Karavanic, K. Kunchithapadam, and T. Newhall. The Paradyn Parallel Performance Measurement Tool. *IEEE Computer*, 28(11):37–46, 1995.

[16] B. Mohr, A. Malony, S. Shende, and F. Wolf. Towards a Performance Tool Interface for OpenMP: An Approach Based on Directive Rewriting. Accepted for the 3rd European Workshop on OpenMP (EWOMP 2001), September 2001.

[17] OpenMP Architecture Review Board. OpenMP Specifications, October 1998. http://www.openmp.org/specs/.

[18] ESPRIT Working Group APART (Automatic Performance Analysis: Resources and Tools). Homepage. http://www.fz-juelich.de/apart/.

[19] J. Vetter. Performance Analysis of Distributed Applications using Automatic Classification of Communication Inefficencies. In *Proc. of the 14th International Conference on Supercomputing*, pages 245–254, Santa Fe, New Mexico, May 2000.

[20] F. Wolf and B. Mohr. EARL - A Programmable and Extensible Toolkit for Analyzing Event Traces of Message Passing Programs. In A. Hoekstra and B. Hertzberger, editors, *Proc. of the 7th International Conference on High-Performance Computing and Networking (HPCN'99)*, pages 503–512, Amsterdam (The Netherlands), 1999.

[21] F. Wolf and B. Mohr. Automatic Performance Analysis of MPI Applications Based on Event Traces. In *Proc. of the European Conference on Parallel Computing (Euro-Par)*, pages 123–132, Munich (Germany), August 2000.

[22] F. Wolf and B. Mohr. Specifying Performance Properties of Parallel Applications Using Compund Events. *Parallel and Distributed Computing Practices (Special Issue on Monitoring Systems and Tool Interoperability)*, 2001. Accepted for publication.

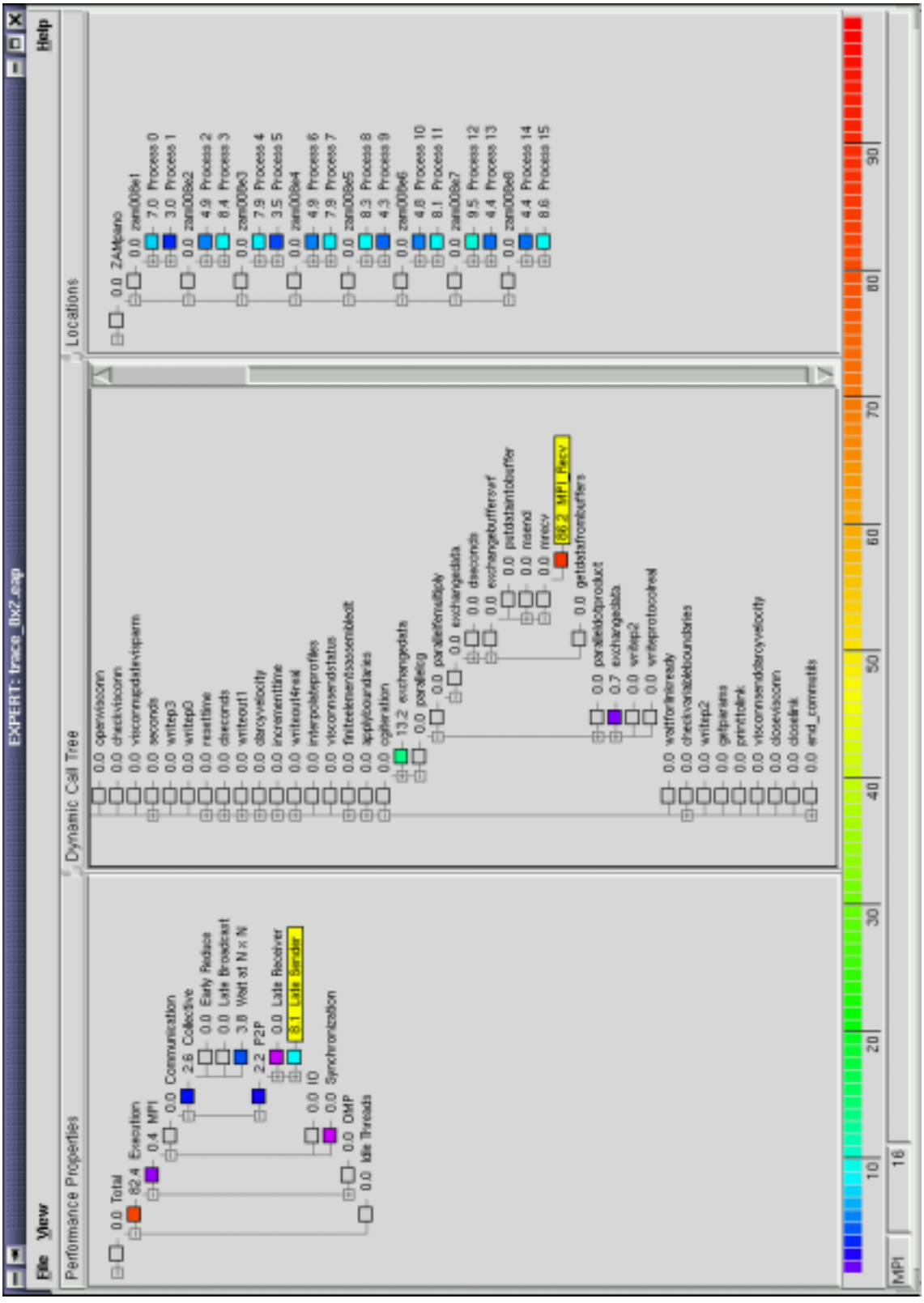

Figure 3: Display of performance behavior in EXPERT for MPI application TRACE. The values and colors on the left are percentages of the total CPU allocation time. The percentages in the middle refer only to the selection (yellow box) on the left. The percentages on the right refer only to the selection in the middle.

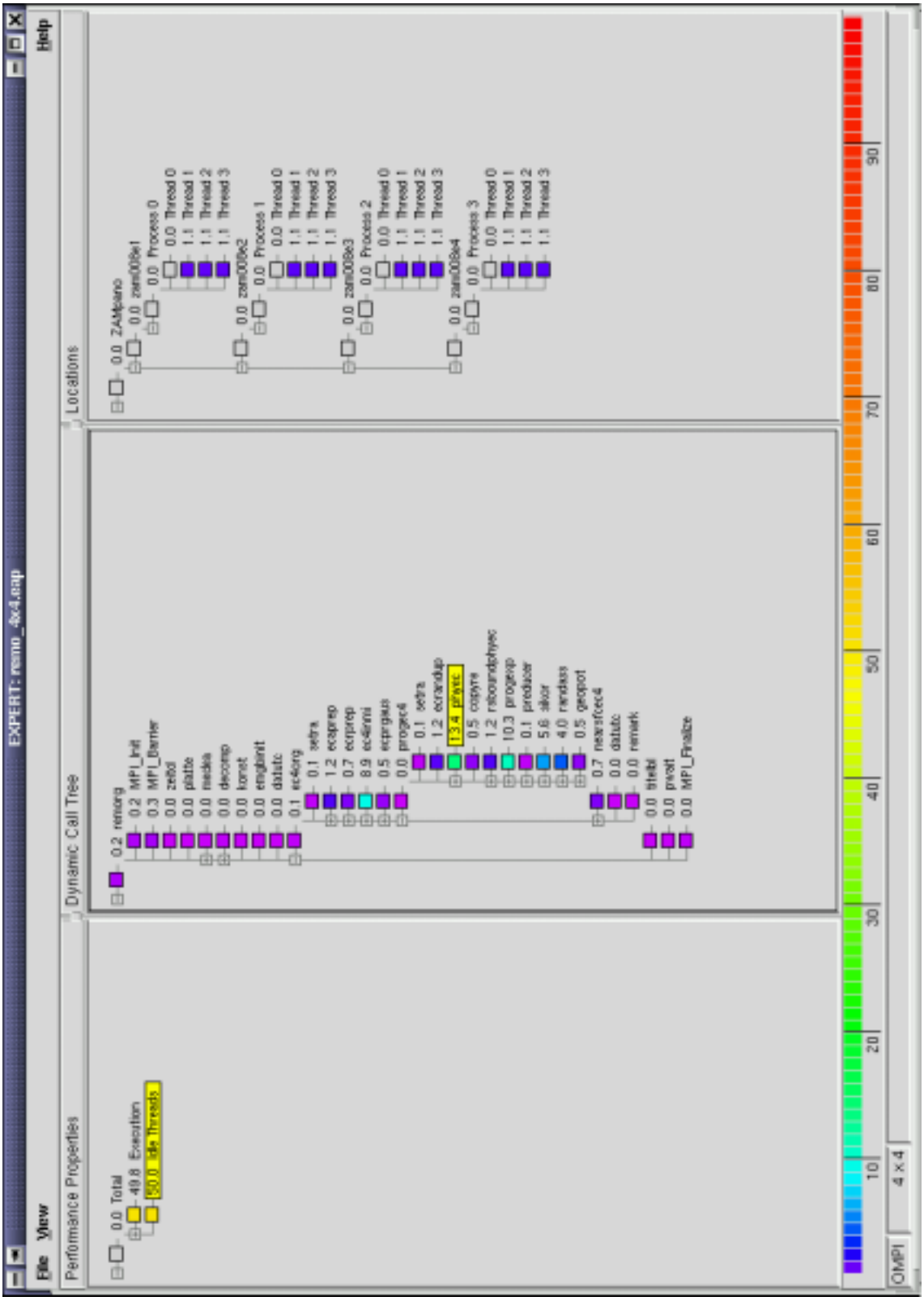

Figure 4: Display of performance behavior in EXPERT for MPI/OpenMP application REMO. In contrast to Fig. 3 the values and colors of all three views are percentages of the total CPU allocation time.