Available at
www.ElsevierComputerScience.com
POWERED BY SCIENCE DIRECT®

Journal of Systems Architecture 49 (2003) 421–439

JOURNAL OF SYSTEMS ARCHITECTURE

www.elsevier.com/locate/sysarc

Automatic performance analysis of hybrid MPI/OpenMP applications

Felix Wolf [a,*,1], Bernd Mohr [b]

[a] *Innovative Computing Laboratory, University of Tennessee, 1122 Volunteer Boulevard, Suite 413, Knoxville, TN 37996-3450, USA*
[b] *Zentralinstitut für Angewandte Mathematik, Forschungszentrum Jülich, 52425 Jülich, Germany*

Abstract

The EXPERT performance-analysis environment provides a complete tracing-based solution for automatic performance analysis of MPI, OpenMP, or hybrid applications running on parallel computers with SMP nodes. EXPERT 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). The set of predefined problems can be extended to meet application-specific needs. The analysis is carried out along three interconnected dimensions: class of performance behavior, call tree, and thread of execution. Each dimension is arranged in a hierarchy so that the user can investigate the behavior on varying levels of detail. All three dimensions are interactively accessible using a single integrated view.
© 2003 Elsevier B.V. All rights reserved.

Keywords: Performance analysis; Parallel computing; Event tracing; User interface

1. Introduction

Coupling SMP systems combines the packaging efficiencies of shared-memory multiprocessors with the scaling advantages of distributed-memory architectures. The result is a computer architecture that can scale more cost-effectively in size. Unfortunately, these systems come at the price of a more complex programming environment to deal with the different modes of parallel execution: shared-memory multithreading vs. distributed-memory message passing. As a consequence, performance optimization becomes more difficult and creates a need for advanced performance tools that are custom made for this class of computing environments. While performance tools exist for shared-memory systems and for distributed-memory systems, solving performance problems on parallel computers with SMP nodes is not as simple as combining two tools. When dealing with hybrid (MPI/OpenMP) parallel executions, performance problems arise where an integrated view is required. Current state-of-the-art tools such as VGV [1] can provide such an integrated view including the necessary monitoring capabilities, but suffer from performance-information overload, unable to abstract performance problems from detailed performance data in an integrated hybrid framework.

* Corresponding author. Tel.: +1-865-974-8992; fax: +1-865-974-8296.
E-mail addresses: fwolf@cs.utk.edu (F. Wolf), b.mohr@fz-juelich.de (B. Mohr).
[1] This work was done while Felix Wolf was a Ph.D. student at Forschungszentrum Jülich.

The EXPERT performance-analysis environment [2] is able to automatically detect performance problems in event traces of MPI [5], OpenMP [6], or hybrid applications running on parallel computers with SMP nodes as well as on more traditional non-SMP or single SMP systems. Performance problems are represented as execution patterns that correspond to situations of inefficient behavior. These patterns are specified as compound events which are input for an automatic analysis process that recognizes and quantifies the inefficient behavior in event traces. Mechanisms that hide the complex relationships within compound-event specifications allow a simple description of complex inefficient behavior on a high level of abstraction. In addition, the set of predefined performance problems can bed extended to meet individual (e.g., application-specific) needs.

Like Paradyn [7], which searches for performance problems along different program-resource hierarchies including the call graph [8], EXPERT takes advantage of decomposing the search space into multiple hierarchical dimensions. The analysis process of EXPERT automatically transforms the event traces into a three-dimensional representation of performance behavior. The first dimension is the kind of behavior. The second dimension is the call tree and describes the behavior's 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. The hierarchical organization of each dimension enables the investigation of performance behavior on varying levels of granularity. Each point of the representation is uniformly mapped onto the corresponding fraction of execution time, allowing the convenient correlation of different behavior using only a single view. The user can interactively access all the hierarchies constituting a dimension of performance behavior using standard tree browsers.

[2] The work on EXPERT is carried out as a part of the KOJAK project [2,3] and is embedded in the IST working group APART [4].

The remainder of this article is organized as follows: First, we consider related work in Section 2. Then, we describe the overall architecture of our analysis environment in Section 3. In Section 4, we present the abstraction mechanisms used to simplify the specification of 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. While Section 6 lists limitations of the current implementation, Section 7 proves our concept by applying it to four realistic codes. Finally, we conclude the paper in Section 8.

This work evolved from a Ph.D. thesis project at Forschungszentrum Jülich. A more detailed and comprehensive description of this article's contents can be found in the thesis document [9].

2. Related work

The multidimensional hierarchical decomposition of the search space for performance problems has a long tradition. Miller et al. [7] developed the W^3 search model as the basis of the online performance-analysis performed by Paradyn. The W^3 model describes performance behavior along the dimensions performance problem, program resources including the call graph [8], 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. The different metrics can be specified in a flexible manner using the MDL metric-description language [10]. The main accomplishments of EXPERT in contrast to Paradyn is the description of performance problems in terms of complex event patterns that go beyond counter-based metrics. Also, the uniform mapping of arbitrary performance behavior onto execution time allows the correlation of different behavior in a single view.

Espinosa [11] 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 [12] 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. As opposed to this approach, EXPERT draws conclusions from the temporal relationships of individual events in a platform-independent way, which does not require any training prior to analysis.

JavaPSL [13] has been designed by Fahringer et al. to specify performance properties based on the Java programming language and to be used in the Aksum tool. Whereas EXPERT uses Python to provide a uniform interface to performance properties, JavaPSL exploits similar mechanisms of the Java language, such as polymorphism, abstract classes, and reflection. In contrast to EXPERT, which concentrates on compound-event analysis, JavaPSL puts emphasis on the definition of performance properties based on existing ones (e.g., by defining metaproperties).

Much work has been done on the visualization of performance data. Apart from standard displays of profiles and event traces, such as Apprentice [14] and VAMPIR [15] (Fig. 7), and call-graph-based profile displays, such as Xprofiler [16], very sophisticated performance data displays tried to approach the problem of hiding tool complexity behind simple but still expressive presentation techniques. Solutions range from animated displays, such as those included in ParaGraph [17], to complete virtual reality environments that allow an immersive investigation of the performance space, such as Virtue [18]. However, the emphasis of EXPERT was not the invention of a new display in a technical sense. After all, the use of tree browsers is not revolutionary and even the coloring of nodes in a tree has been previously applied, for example, in the xlcb [19] corefile browser. However, EXPERT shows that an intuitive but still insightful perception of performance behavior can be achieved through uniformity and simplicity both in the logical model of the performance space as well as in its visual representation, which is realized just by coupling standard tree browsers.

3. Overall architecture

The EXPERT performance-analysis environment is depicted in Fig. 1. The different components are represented as boxes with rounded corners and their inputs and outputs are represented as paper sheets with the upper-right corner turned down. The arrows illustrate the whole performance-analysis process from instrumentation to result presentation.

The EXPERT analysis process is composed of two parts: a semi-automatic multilevel instrumentation of the user application followed by an automatic analysis of the generated performance data. The first subprocess is called 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 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 [20]. This is done to capture performance-relevant OpenMP events, 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. OPARI supports all languages for which OpenMP is defined.

Instrumentation of user functions is done either during compilation using a compiler-supplied profiling interface or on the source-code level using TAU [21]. TAU is able to automatically instrument the source code of C, C++, and Fortran

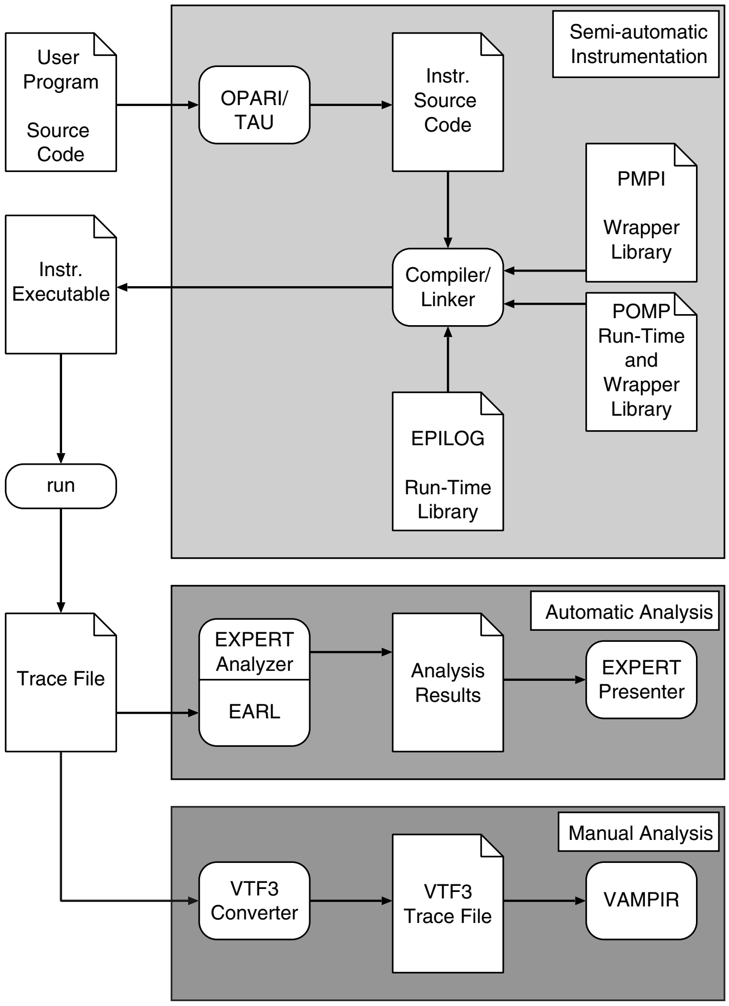


Fig. 1. EXPERT overall architecture.

programs using a preprocessor based on the PDT [22] toolkit.

Instrumentation for MPI events is accomplished with a wrapper library based on the PMPI profiling interface [5], which generates MPI-specific events by intercepting calls to MPI functions. All MPI, OpenMP, and user-function instrumentations call the EPILOG run-time library, which provides mechanisms for buffering and trace-file creation. 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. The analyzer uses EARL [23] to provide a high-level view of the raw trace file. We call this view the *enhanced event model*, and it is where the actual analysis takes place. The analyzer generates an analysis report, which serves as input for the EXPERT presenter.

Using the presenter the user can conveniently navigate through the entire search space along the dimensions bottleneck type, call path, and location. In addition, the automatic analysis can be combined with a manual analysis using VAMPIR [15], which allows the user to investigate the patterns identified by EXPERT in a time-line display. To do this, the user only needs to convert the EPILOG trace file into the VTF3 format. Currently, the EXPERT tool environment supports the following platforms:

- Cray T3E
- Hitachi SR-8000
- IBM Power3 and Power4 cluster
- Linux IA-32 cluster
- SGI MIPS cluster

Compiler-controlled instrumentation can be done either on Linux PC clusters using the unpublished profiling interface of the PGI [24] compiler or on Hitachi using the proprietary compiler [25], respectively. However, currently neither TAU nor the PGI and Hitachi compilers support efficient call-site instrumentation, and hence the traces currently carry no call-site information. For this reason, a call path computed by EXPERT may actually represent a set of call paths differing only in call sites (e.g., line number of a function call).

4. Abstraction mechanisms

A raw EPILOG event trace, as generated by the instrumentation, contains a chronologically sorted sequence of "primitive" events. There are different types of events and, depending on the event type, each event may have different attributes. However, all event types provide at least a time and location attribute characterizing the control flow causing it. This representation of program execution is called the *basic event model*.

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 cover 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.

EARL maps a raw event trace, which is represented on the level of the basic event model, onto an *enhanced event model*. The enhanced event model provides abstractions that allow compound events representing inefficient behavior to be easily described (see [9,26] for details). Whereas the basic event model provides only sequential access to the events, the enhanced event model allows events to be accessed randomly using the event position as a reference (i.e., the first event has position one, etc.). Additionally, the enhanced event model provides two types of abstractions on top of the basic part of the model:

- State sequences
- Pointer attributes

State sequences 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. This allows compound events to be described in the context of the execution state. An example is the *message queue* containing all events of sending messages currently being transferred. Thus, EARL is able to return for a given event the set of all send events of messages that have been sent before or by the given event itself and that have not yet been received at the time of the given event. This set of events is returned as a set of positions, which can then be used to access each event in the set individually. Since each event in the trace is associated with one such set (i.e., one state of the message queue), these sets form themselves a sequence that describes the evolution of one aspect of the system's execution state over time. Other state sequences describe different aspects of the execution state, such as the region call stack or the progress of collective operations.

Pointer attributes connect two corresponding events with one another, so that one can define compound events along a path of corresponding events. Corresponding events are events referring to the same larger unit of activity, such as a whole message exchange. Pointer attributes are added to the attributes defined in the basic model and carry as their value the position of the corresponding event. An example is an attribute pointing from a message-receive event to the corresponding send event. After accessing the receive event, it is easy to access the corresponding send event because its position is provided as an attribute value of the receive event. Other pointer attributes describe different types of correspondence, such as acquiring and releasing a lock object or entering and leaving a region.

An essential part of the enhanced model is the dynamic call tree, which is computed from all enter and exit events. As an additional pointer attribute, EARL provides a link from each enter event to the first enter event visiting the same node in the call tree (i.e., call path). This provides a simple means of associating a performance-relevant compound event with the corresponding execution phase of the parallel program.

5. Analysis component

The design of the analyzer is based on the specifications and terminology presented in [27]. The analyzer attempts to prove *performance properties* for one execution of a parallel application and to quantify them according to their influence on the performance. A performance property characterizes a class of performance behavior and is specified in terms of a *compound event*, which the analyzer tries to detect in an event trace. A compound event is a set of events matching a specific execution pattern, whose constituents are connected by relationships and constraints. For each property, EXPERT calculates a *severity* measure indicating the fraction of execution time spent on that property and, thus, allows the correlation of different properties in a single view.

The run-time events of a parallel application occur on multiple time lines—one for each control flow (i.e., thread). EXPERT regards all control flows as being mapped to different CPUs at any time, that is, processes or threads running on the same SMP node do not share a CPU. EXPERT describes the severity of a particular performance property in terms of wall-clock interval sets that may be distributed across different time lines. All interval sets are subsets of the CPU-*reservation time*, which is the time from the first to the last event multiplied by the number of threads. The severity is defined as the amount of such an interval set and is later presented in percentage of the CPU-reservation time.

The analyzer is implemented in Python using 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 and specified separately.

5.1. Specification of performance properties

Many tools specify performance aspects to be analyzed using a specification language. For example, Paradyn describes performance metrics using the MDL metric-description language [10], while Aksum [13] uses Java to represent different performance properties. Performance properties in EXPERT are specified using a Python [28] class interface to the abstractions discussed in Section 4, which themselves are implemented in C++.

The performance properties in EXPERT 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 of 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 state sequences. This mechanism allows the simple specification of very complex performance-relevant situations and an explanation of inefficiency that is based on 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. Fig. 2 shows the hierarchy of predefined performance properties that are supported by the current prototype of EXPERT. The set should not be regarded as complete, but it is representative in that it shows the usefulness and feasibility of the analysis method and the advantages of the tool architecture used to implement it.

The set of performance properties is split into two parts. The first part, which constitutes the upper layers of the hierarchy and which is indicated by white boxes, is mainly based on summary information involving, for example, the total execution times of special MPI routines, which could also be provided by a profiling tool. However, the second part, which constitutes the lower layers of

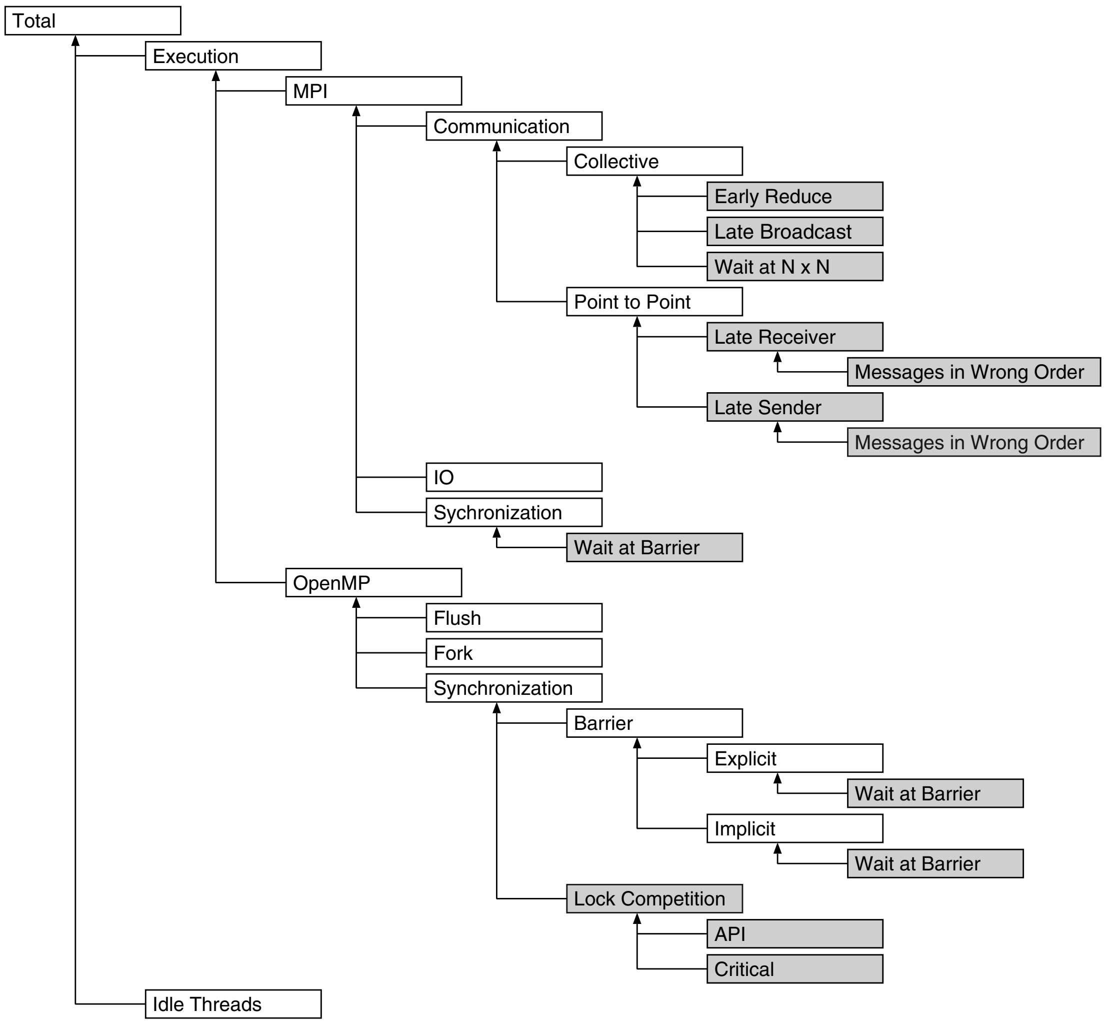


Fig. 2. Hierarchy of performance properties.

the hierarchy and which is indicated by gray boxes, involves idle times that can only be determined by comparing the chronological relation between individual events. This is where the compound-event approach can demonstrate its full power. A major advantage of EXPERT lies in its ability to handle both groups of performance properties in one step. The following briefly explains the performance properties that are currently implemented in EXPERT.

Total. Time spent on program execution including the idle times of slave threads during OpenMP sequential execution.

Execution. Time spent on program execution but without the idle times of slave threads during OpenMP sequential execution.

MPI. Time spent on MPI API calls.

Communication. Time spent on MPI API calls used for communication.

Collective. Time spent on collective communication.

Early Reduce. Collective communication operations that send data from all processes to one destination process (i.e., n-to-1) may suffer from waiting times if the destination process enters the operation earlier than its sending counterparts, that is, before any data could have been sent. The property refers to the time lost as a result of that situation.

Late Broadcast. Collective communication operations that send data from one source process to all processes (i.e., 1-to-n) may suffer from waiting times if destination processes enter the operation earlier than the source process, that is, before any data could have been sent. The property refers to the time lost as a result of that situation.

Wait at $N \times N$. Collective communication operations that send data from all processes to all processes (i.e., n-to-n) exhibit an inherent synchronization among all participants, 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.

Point to Point. Time spent on point-to-point communication.

Late Receiver. A send operation is blocked 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 is blocked until the data is transferred to the receiver.

Messages in Wrong Order (Late Receiver). A *Late Receiver* situation may be the result of messages that are sent in the wrong order, that is, not in the order the receiving processes are expecting them. If a process sends messages to processes that are not ready to receive them, the sender's MPI-internal buffer may overflow so that from then on the process needs to send in synchronous mode causing a *Late Receiver* situation.

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 has been started.

Messages in Wrong Order (Late Sender). A *Late Sender* situation may be the result of messages that are received in the wrong order. If a process expects messages from one or more processes in a certain order while these processes are sending them in a different order, the receiver may need to wait longer for a message because this message may be sent later while messages sent earlier are ready to be received. Both *Messages in Wrong Order* properties have been motivated by [29].

IO (MPI). Time spent on MPI file IO.

Synchronization (MPI). Time spent on MPI barrier synchronization.

Wait at Barrier (MPI). This property is similar to the property *Wait at* $N \times N$. It covers the time spent on waiting in front of an MPI barrier. The time until all processes have entered the barrier is measured and used to compute the severity.

OpenMP. Time spent on the OpenMP run-time system.

Flush (OpenMP). Time spent on flush directives.

Fork (OpenMP). Time spent by the master thread on team creation.

Synchronization (OpenMP). Time spent on OpenMP barrier or lock synchronization. Lock synchronization may be accomplished using either API calls or critical sections.

Barrier (OpenMP). The time spent on implicit (compiler-generated) or explicit (user-specified) OpenMP barrier synchronization. As already mentioned, implicit barriers are treated similar to explicit ones. The instrumentation procedure replaces an implicit barrier with an explicit barrier enclosed by the parallel construct. This is done by adding a nowait clause and a barrier directive as the last statement of the parallel construct. In cases where the implicit barrier cannot be removed (i.e., parallel region), the explicit barrier is executed in front of the implicit barrier, which will be negligible because the team will already be synchronized when reaching it. The synthetic explicit barrier appears in the display as a special implicit barrier construct.

Explicit (OpenMP). Time spent on explicit OpenMP barriers.

Implicit (OpenMP). Time spent on implicit OpenMP barriers.

Wait at Barrier (Explicit). This property corresponds to the property *Wait at* $N \times N$. It covers the time spent on waiting in front of an explicit (user-specified) OpenMP barrier. The time until all processes have entered the barrier is measured and used to compute the severity.

Wait at Barrier (Implicit). Similar to the preceding property, this property covers the time spent on waiting in front of an implicit (compiler-generated) OpenMP barrier. The time until all processes have entered the barrier is measured and used to compute the severity.

Lock Competition (OpenMP). This property refers to the time a thread spent on waiting for a lock that had been previously acquired by another thread.

API (OpenMP). Lock competition caused by OpenMP API calls.

Critical (OpenMP). Lock competition caused by critical sections.

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

Note that all properties involving collectively executed operations, such as MPI collective communication or OpenMP barriers, require to identify all parts of individual collective-operation instances in the event stream.

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 (i.e., call path) and the location (i.e., thread). Thus, the complete performance behavior is represented using a three-dimensional matrix, where each element contains the severity for a specific performance property, call path, 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 [9,30]).

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 a tree browser that labels each node with a weight. EXPERT uses as weight the severity associated with a performance property (i.e., a percentage of the CPU-reservation time). 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. 3). In the

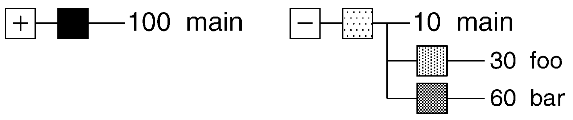


Fig. 3. Node of the call tree in collapsed and expanded state.

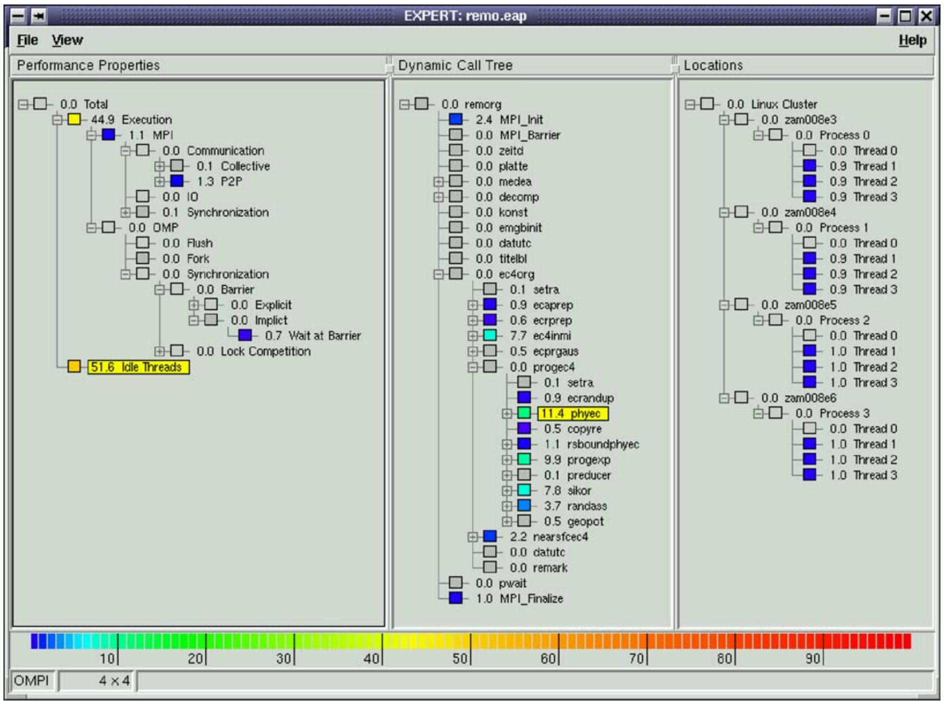
collapsed state, this node is labeled with the weight representing the time spent in the whole program. In the expanded state it displays only the fraction that is spent neither in *foo* nor in *bar*.

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%). To avoid distraction, insignificant values below 0.5% are displayed in gray. 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 trees of the different analysis dimensions are interconnected so that the user can investigate the distribution of a selected performance property across the call tree, and the distribution of a selected performance property present in a selected call-tree node across the locations. In Fig. 4, the selections are indicated by framed node labels. Thus, the user can investigate the performance behavior in a scalable but still accurate way along all its interconnected dimensions using only a single integrated view.

In the *absolute view mode* (i.e., default), the display represents the severity as a percentage of the total CPU-reservation time. However, always referring to the total CPU-reservation time may limit scalability because values may become very small (e.g., in the case of many locations). For this reason, the presenter offers a *relative view mode*. In this relative view mode, a percentage shown in a tree always refers to the selection in the left neighbor tree.

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, it is possible to navigate across the performance space in a scalable but still accurate way



```
0.0 Total
  44.9 Execution
    1.1 MPI
      0.0 Communication
        0.1 Collective
        1.3 P2P
      0.0 IO
      0.1 Synchronization
    0.0 OMP
      0.0 Flush
      0.0 Fork
      0.0 Synchronization
        0.0 Barrier
          0.0 Explicit
          0.0 Implicit
            0.7 Wait at Barrier
        0.0 Lock Competition
  51.6 Idle Threads
```

```
0.0 remorg
  2.4 MPI_Init
  0.0 MPI_Barrier
  0.0 zeitd
  0.0 platte
  0.0 medea
  0.0 decomp
  0.0 konst
  0.0 emgbinit
  0.0 datutc
  0.0 titelbl
  0.0 ec4org
    0.1 setra
    0.9 ecaprep
    0.6 ecrprep
    7.7 ec4inmi
    0.5 ecprgaus
    0.0 progec4
      0.1 setra
      0.9 ecrandup
      11.4 phyec
      0.5 copyre
      1.1 rsboundphyec
      9.9 progexp
      0.1 preducer
      7.8 sikor
      3.7 randass
      0.5 geopot
    2.2 nearsfcec4
    0.0 datutc
    0.0 remark
  0.0 pwait
  1.0 MPI_Finalize
```

```
0.0 Linux Cluster
  0.0 zam008e3
    0.0 Process 0
      0.0 Thread 0
      0.9 Thread 1
      0.9 Thread 2
      0.9 Thread 3
  0.0 zam008e4
    0.0 Process 1
      0.0 Thread 0
      0.9 Thread 1
      0.9 Thread 2
      0.9 Thread 3
  0.0 zam008e5
    0.0 Process 2
      0.0 Thread 0
      1.0 Thread 1
      1.0 Thread 2
      1.0 Thread 3
  0.0 zam008e6
    0.0 Process 3
      0.0 Thread 0
      1.0 Thread 1
      1.0 Thread 2
      1.0 Thread 3
```

Fig. 4. Display of performance behavior in EXPERT for REMO in the absolute view mode.

along all its interconnected dimensions. First, the presenter allows exploration of the full performance space by showing the results of a multidimensional analysis in a multidimensional fashion using three interconnected tree browsers. Second, instead of confusing the user with differently styled views for different metrics, all performance properties are uniformly accommodated in the same display and thus provide the ability to easily compare the effects of different kinds of performance behavior. In addition, since the user only needs to get accustomed to one way of presentation, the necessary learning efforts are small. Third, the ability to investigate the performance behavior of individual nodes in the call tree (i.e., call paths) either including or excluding their descendants allows the analysis of complex source-code hierarchies along the functional dependences of their elements.

5.4. Extension mechanism

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, the user can simply write another pattern class that implements an 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. Limitations

Despite its strengths, the approach taken here exhibits some limitations that result both from general limitations of event tracing on the one hand and from particular properties of the trace-analysis method proposed here on the other hand.

First, the event-trace size, which may easily reach several millions of events or several hundreds of megabytes when dumped to a file, constitutes a severe obstacle to a ubiquitous application of all trace-based performance-analysis techniques. The difficulties of handling large traces result from their local buffer-memory requirements during generation, which may, in addition to competing for the target application's memory, cause significant perturbation when the buffer contents are written to a file as a result of buffer overflow. Also, global trace-file sizes may limit scalability in the case of massively parallel systems with thousands of processors.

Second, as a consequence of the enormous trace-file sizes, the analysis process performed by EXPERT may take several hours to complete. Although a processing time of several hours might be acceptable if it results in substantial performance improvements, to convince the user community a production tool should offer more convenience also with respect to speed. However, the current Python implementation still offers opportunities for optimization.

Third, the way EXPERT computes inclusive and exclusive times associated with a performance property imposes certain constraints on the way performance properties can be arranged in the property hierarchy. To ensure the correctness of computing the inclusive and exclusive severity, the measured severity intervals of siblings in the property hierarchy must be non-overlapping either in time or in location, which may limit the freedom of extending that hierarchy at least to some extent.

We are currently working on several extensions and enhancements to overcome—or at least limit the impact of—these limitations.

7. Examples

The EXPERT performance-analysis environment has been tested for several real-world applications. This section demonstrates that the performance problems addressed by the present approach are of practical relevance and that they can be conveniently localized using the EXPERT presentation component. The test cases comprise two hybrid applications, REMO and SWEEP3D, and two MPI applications, CX3D and TRACE.

The latter two also demonstrate how the automatic analysis done by EXPERT can be combined with VAMPIR's time-line displays to more deeply understand the problems at hand. We consider one event trace per application.

All the experiments were conducted on ZAMpano [31], a parallel computer with eight SMP nodes, each having four Intel Pentium III Xeon (550 MHz) CPUs. CPU reservation was done such that one CPU per thread or single-threaded process was available to each application. Instrumentation of user code was performed using the PGI compiler.

Table 1 summarizes trace-file size and overhead. The head row contains the program name, the first row below shows the number of CPUs used, the second row lists the trace-file size, and the third row gives the execution time. To estimate the runtime overhead introduced by the instrumentation, the minimum execution time of a series of 10 uninstrumented runs was compared to the minimum execution time of a series of 10 instrumented runs. The result is listed in the fourth row. Finally, the last row shows the duration of the analysis process carried out on the test platform.

The large trace-files sizes obtained for only short execution times expose a limitation of the current approach. Restricting the tracing to selected parts of the program and its execution might help to reduce temporal event density while preserving relevant information. The inconveniently long analysis run times are not only a result of large trace-file sizes, but also a consequence of the prototype's early design stage. One opportunity for optimization is, for example, an improved information exchange among different performance properties during analysis. In addition, the re-implementation in a compiled programming language instead of an interpreted scripting language, such as Python, might also contribute to better speed results. The overhead numbers presented in the table are satisfactory, only the instrumentation overhead of REMO reaches nearly 10%. However, since the performance problem identified in REMO is large in relation to the overall execution time, the numbers presented here concerning this problem are still useful (Section 7.1).

Table 1
Trace-file size and overhead

	REMO	SWEEP3D	CX3D	TRACE
CPUs	16	16	8	16
Size (MB)	170	72	34	310
Execution time (s)	37.2	16.5	139.8	58.9
Overhead (%)	9.7	6.0	0.1	4.2
Analysis time (h:min)	9:48	3:22	1:25	12:57

7.1. REMO

REMO [32] is a weather forecast application of the DKRZ (Deutsches Klima Rechenzentrum). It implements a hydrostatic limited area model, which is based on the *Deutschland/Europa* weather forecast model of the German Meteorological Services (Deutscher Wetterdienst (DWD)). We consider an early experimental MPI/OpenMP version of the production code. The application was executed on four nodes with one process per node and four threads per process (4 processes$\times$4 threads).

Fig. 4 shows the result display of REMO in the absolute mode, that is, all values and colors represent percentages of the total CPU-reservation time. The property view indicates that one half (i.e., 51.6%) of the total CPU-reservation time is idle time (i.e., *Idle Threads*) resulting from OpenMP sequential execution outside of parallel regions. Although during this period the idle threads actually do not execute any code, the time is mapped onto the call paths that have been executed by the master thread during this time. That is to say, for analysis and presentation purposes EXPERT assumes that outside parallel regions the slave threads "execute" the same code as their master thread. This method of call-path mapping helps to identify parts of the call tree that might be optimized in order to reduce the amount of sequential execution.

In the case of REMO, the EXPERT display (Fig. 4, middle) allows the easy identification of two call paths as major sources of idle times. The location view (Fig. 4, right) shows the distribution of the idle time across the slave threads.

7.2. SWEEP3D

The benchmark code SWEEP3D [33] represents the core of a real ASCI application. It solves a 1-group time-independent discrete ordinates (Sn) 3D Cartesian (XYZ) geometry neutron transport problem. We consider an early experimental MPI/OpenMP version of the original MPI version. While MPI is responsible for parallelism by domain decomposition, OpenMP is responsible for parallelism by multitasking.

The application was executed on four nodes with one process per node and four threads per process (4 processes × 4 threads). The performance behavior of SWEEP3D demonstrates a potential hazard of hybrid programming, that is, a performance problem resulting from the combination of MPI and OpenMP. MPI calls made outside a parallel region prolong sequential execution and prevent available CPUs from being used by multiple threads.

The results are shown in Table 2. While the top section of the table lists two call paths, the bottom section contains the numerical results obtained for the whole program and these two call paths. The values in the bottom section represent percentages of the total CPU-reservation time. The first column refers to the whole program, whereas the second and third columns refer to the call paths listed above in the table. The call path (a) shown in the table is responsible for most of the losses occurring due to the property *Idle Threads*. However, at the same time this call path exhibits a significant loss due to the property *Late Sender*. Note that *Late Sender* counts the times of the master threads, whereas *Idle Threads* counts the times of the slave threads (three slaves per master). Taking this into account, reducing *Late Sender* by 1% would speed up the application by 4% because speeding up the master also reduces idle times of the slaves. Obviously, one reason for the *Late Sender* problem at call path (a) is receiving messages in the reverse sending order (*Messages in Wrong Order*). Moreover, a significant amount of time is spent on the implicit (i.e., compiler-generated) OpenMP barrier at the end of call path (b). Expanding the node of the property *Implicit Barrier* (Fig. 5, left) reveals that most of that time is lost due to the property *Wait at Barrier*.

7.3. CX3D

CX3D is an MPI application used to simulate *Czochralski* crystal growth [34], a method applied in the silicon-wafer production. The simulation covers the convection processes occurring in a rotating cylindrical crucible filled with liquid melt. The convection, which strongly influences the chemical and physical properties of the growing crystal, is described by a system of partial differential equations. The crucible is modeled as a three-dimensional cubic mesh with its round shape expressed by cyclic border conditions. The mesh is distributed across the available processes using a two-dimensional spatial decomposition. The application was executed on two SMP nodes with four processes per node. MPI communication within SMP nodes was done via shared memory.

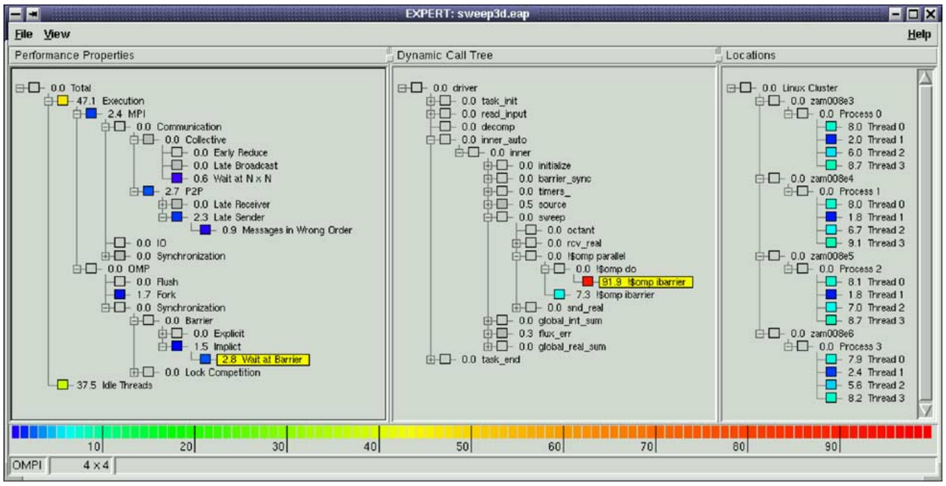
Most of the execution time is spent in a routine called VELO, which is responsible for calculating

Table 2
Performance problems found in SWEEP3D in percentage of the total CPU-reservation time

Call paths

(a) `seep3d` → `inner_auto` → `inner` → `sweep` → `recv_real` → `MPI_Recv`
(b) `driver` → `inner_auto` → `inner` → `sweep` → `!$omp parallel` → `!$omp do` → `!$omp ibarrier`

Performance property	Whole program	(a)	(b)
Idle Threads	37.5	17.5	
Communication	6.5	5.8	
Late Sender	3.2	3.2	
Messages in Wrong Order	0.9	0.9	
Implicit Barrier (OpenMP)	4.3		3.3
Wait at Barrier (OpenMP, implicit)	2.8		2.6



EXPERT: sweep3d.eap

Performance Properties

```
0.0 Total
  47.1 Execution
    2.4 MPI
      0.0 Communication
        0.0 Collective
          0.0 Early Reduce
          0.0 Late Broadcast
          0.6 Wait at N x N
        2.7 P2P
          0.0 Late Receiver
          2.3 Late Sender
            0.9 Messages in Wrong Order
      0.0 IO
      0.0 Synchronization
    0.0 OMP
      0.0 Flush
      1.7 Fork
      0.0 Synchronization
        0.0 Barrier
          0.0 Explicit
          1.5 Implict
            2.8 Wait at Barrier
        0.0 Lock Competition
  37.5 Idle Threads
```

Dynamic Call Tree

```
0.0 driver
  0.0 task_init
  0.0 read_input
  0.0 decomp
  0.0 inner_auto
    0.0 inner
      0.0 initialize
      0.0 barrier_sync
      0.0 timers_
      0.5 source
      0.0 sweep
        0.0 octant
        0.0 rcv_real
        0.0 !$omp parallel
          0.0 !$omp do
            91.9 !$omp ibarrier
          7.3 !$omp ibarrier
        0.0 snd_real
      0.0 global_int_sum
      0.3 flux_err
      0.0 global_real_sum
  0.0 task_end
```

Locations

```
0.0 Linux Cluster
  0.0 zam008e3
    0.0 Process 0
      8.0 Thread 0
      2.0 Thread 1
      6.0 Thread 2
      8.7 Thread 3
  0.0 zam008e4
    0.0 Process 1
      8.0 Thread 0
      1.8 Thread 1
      6.7 Thread 2
      9.1 Thread 3
  0.0 zam008e5
    0.0 Process 2
      8.1 Thread 0
      1.8 Thread 1
      7.0 Thread 2
      8.7 Thread 3
  0.0 zam008e6
    0.0 Process 3
      7.9 Thread 0
      2.4 Thread 1
      5.6 Thread 2
      8.2 Thread 3
```

OMPI 4 x 4

Fig. 5. Display of performance behavior in EXPERT for SWEEP3D in the relative view mode.

the new velocity vectors. Communication is required when the computation involves mesh cells from the border of each processor's subdomain. The execution configuration of CX3D is determined by the number of processes that are assigned to each of the two decomposed dimensions. The experiment presented here was conducted with a decomposition configuration of 8×1 processes.

The results in Table 3 show that a significant amount of the communication time was introduced by *Late Sender* and *Wait at* $N \times N$. Using the call-tree view (Fig. 6, middle), it is easy to identify two call paths mainly responsible for these performance properties. Both call paths are executed as parts of VELO. They are listed in the top section of the table. Using the location view (Fig. 6, right), one can easily investigate the distribution of the property *Late Sender* across the processes. It is obvious that most of the time associated with this property is caused by process 0 and 7.

A VAMPIR time-line diagram of CX3D when executing VELO is shown in Fig. 7. The middle of the time line exhibits a noticeable *Late Sender* instance. Process 7 tries to receive a message from process 6 using MPI_Recv, but the message is sent long after process 7 has entered the receive operation. Some other but smaller instances follow shortly after this one. Finally, on the right part of the time line one can recognize a *Wait at* $N \times N$ instance across all processes. Note that the workload distribution across all processes for the section of the time line shown here seems to be the reason that the MPI operations are entered earlier by process 7 and, thus, the reason for the inefficient behavior. The workload distribution within a function can be easily obtained in EXPERT by selecting the property *Execution* in expanded state, which then reflects the execution time without communication.

Table 3
Performance problems found in CX3D in percentage of the total CPU-reservation time

Call paths

(a) `velo`$\rightarrow$`crecvxs`$\rightarrow$`MPI_Recv`
(b) `velo`$\rightarrow$`MPI_Allreduce`

Performance property	Whole program	(a)	(b)
Communication	18.4	7.1	6.9
Late Sender	5.8	4.6	
Wait at $N \times N$	7.5		6.6

7.4. TRACE

TRACE [35] simulates the subsurface water flow in variably saturated porous media. The parallelization is based on spatial decomposition and a parallelized CG algorithm. The application was executed using four SMP nodes with four processes per node (4 processes $\times$ 4 processes). MPI

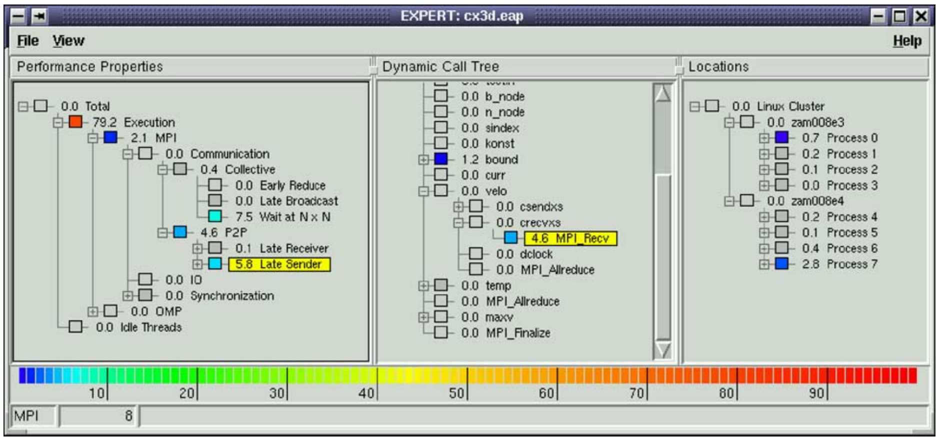


Fig. 6. Display of performance behavior in EXPERT for CX3D in the absolute view mode.

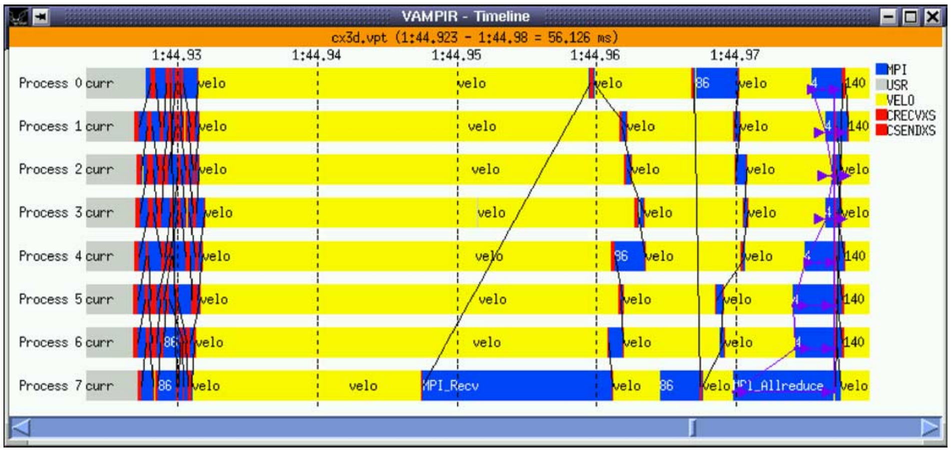


Fig. 7. VAMPIR time-line diagram of CX3D.

communication within SMP nodes was done via shared memory.

Using the performance-property view (Fig. 8, left), it is easy to see that most of the time used for communication routines was spent on waiting due to the situations *Late Sender* and *Wait at* $N \times N$. Using the call-tree view (Fig. 8, middle), one can quickly locate two call paths that are major sources of the previously identified performance problems. The call path mainly responsible for the property *Wait at* $N \times N$ is shown in the vertical middle of the call tree. The presenter display was switched to the relative view mode, that is, whereas the values and colors on the left are percentages of the total CPU-reservation time, the percentages in the middle are fractions of the selection (node with framed label) on the left, and the percentages on the right are fractions of the selection in the middle. For example, the 9.8% shown for the selected call path in the middle is 9.8% of 2.4% of the total CPU-reservation time.

The results of the analysis are listed in Table 4. The first row of the bottom section corresponds to the time spent in MPI communication statements. For the two call paths this is just the time needed for completion of the specific MPI calls at their end. The second and third row correspond to the

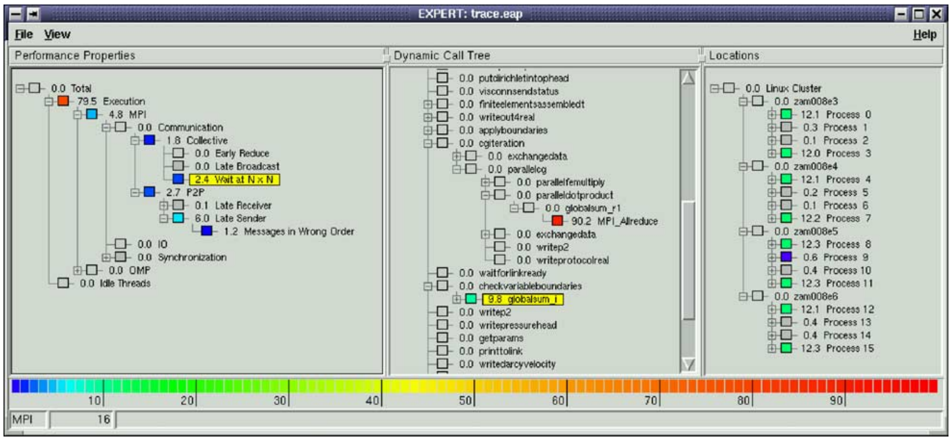


Fig. 8. Display of performance behavior in EXPERT for TRACE in the relative view mode.

Table 4
Performance problems found in TRACE in percentage of the total CPU-reservation time

Call paths

(a) `trace` → `cgiteration` → `parallelcg` → `parallelfemultiply` → `exchangedata` → `exchangebufferswf` → `mrecv` → `MPI_ Recv`
(b) `trace` → `cgiteration` → `parallelcg` → `paralleldotproduct` → `globalsum_rl` → `MPI_Allreduce`

Performance property	Whole program	(a)	(b)
Communication	14.3	7.8	3.0
Late Sender	7.3	5.8	
Wait at $N \times N$	2.4		2.2

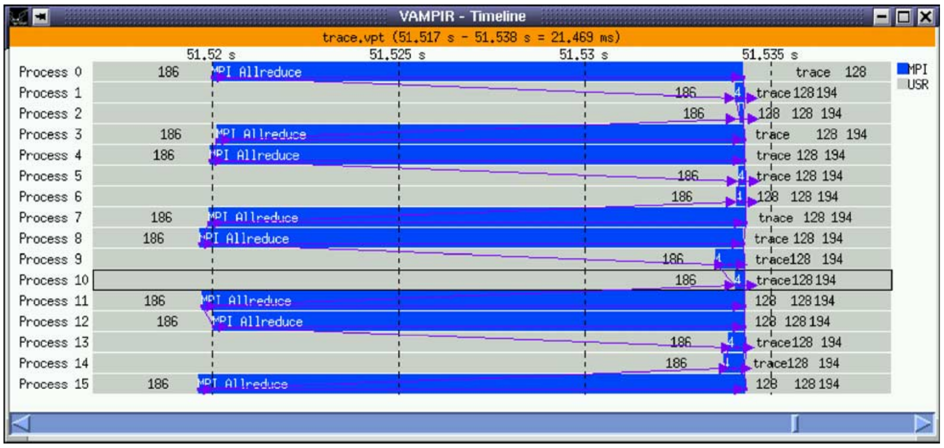


Fig. 9. VAMPIR time-line diagram of TRACE.

waiting times caused by the *Wait at* $N \times N$ and *Late Sender* situations.

The location view (Fig. 8, right) shows the distribution of idle times introduced by *Wait at* $N \times N$ during execution of the call path selected in the middle tree of Fig. 8, which is another call path responsible for that property. Obviously, the idle times expose an uneven but still symmetric distribution across the different processes. The "inner" processes of each SMP node exhibit significantly less waiting time than the "outer" ones. Fig. 9 shows a VAMPIR [15] time-line diagram of TRACE when executing this call path. The time line presents a noticeable *Wait at* $N \times N$ instance. The distribution of the waiting times in MPI_Allreduce shown in the time line bears a clear resemblance to the distribution shown in the EXPERT result display (Fig. 8, right) in that every second pair of processes suffers from significant waiting times.

8. Conclusion

The EXPERT tool environment provides a complete but still extensible solution for automatic performance analysis of MPI, OpenMP, or hybrid applications running on parallel computers with SMP nodes. 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 set of performance-property specifications is embedded in a flexible architecture and can be extended to meet application-specific needs.

The performance behavior is presented along three interconnected dimensions: class of performance behavior, position within the call tree 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. Each point of the representation is uniformly mapped onto the corresponding fraction of CPU-reservation time, allowing the convenient correlation of different behavior in a single integrated view. 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. In the future, we intend to integrate mechanisms for comparative performance analysis. In addition, we plan to improve our result presentation by integrating it more closely with an event-trace browser such as VAMPIR [15] to automatically visualize instances of compound events using time-line diagrams and by adding source-code displays to display their source-code location. Finally, we will work on further improving and completing our performance-property catalog including the integration of hardware-counter based performance properties.

Acknowledgements

We would like to thank all our partners in the IST working group APART for their contributions to this topic. We also would like to thank Arpad Kiss for providing the basic tree browser implementation. And we are grateful to Wolfgang Frings and Reiner Vogelsang for helping us conduct the experiments and to Deutsches Klima Rechenzentrum, the Institut für Chemie und Dynamik der Geosphäre IV, and the Institut für Festkörperforschung for giving access to their applications. Finally, we would like to thank Allen Malony, Janet Carter-Sigglow, and Craig Soules for their helpful comments and suggestions on the language.

References

[1] J. Hoeflinger, B. Kuhn, W. Nagel, P. Petersen, H. Rajic, S. Shah, J. Vetter, M. Voss, R. Woo, An integrated performance visualizer for MPI/OpenMP programs, in: Proceedings of the 3rd European Workshop on OpenMP (EWOMP 2001), Barcelona, Spain, 2001.

[2] F. Wolf, B. Mohr, Automatic performance analysis of hybrid MPI/OpenMP applications, in: Proceedings of the

11th Euromicro Conference on Parallel Distributed and Network based Processing (PDP 2003), Genua, Italy, 2003.

[3] Forschungszentrum Jülich, KOJAK, Kit for objective judgement and knowledge-based detection of performance bottlenecks. Available from <http://www.fz-juelich.de/zam/kojak/>.

[4] IST Working Group APART, Automatic performance analysis: real tools. Available from <http://www.fz-juelich.de/apart/>.

[5] Message Passing Interface Forum, MPI: a message passing interface standard, June 1995. Available from <http://www.mpi-forum.org>.

[6] OpenMP Architecture Review Board, OpenMP Fortran Application Program Interface—Version 2.0, November 2000. Available from <http://www.openmp.org>.

[7] B.P. Miller, M.D. Callaghan, J.M. Cargille, J.K. Hollingsworth, R.B. Irvine, K.L. Karavanic, K. Kunchithapadam, T. Newhall, The Paradyn parallel performance measurement tool, IEEE Computer 28 (11) (1995) 37–46.

[8] H.W. Cain, B.P. Miller, B.J.N. Wylie, A callgraph-based search strategy for automated performance diagnosis, in: Proceedings of the 6th International Euro-Par Conference, Lecture Notes in Computer Science, vol. 1999, Springer, Munich, Germany, 2000.

[9] F. Wolf, Automatic performance analysis on parallel computers with SMP nodes, Ph.D. Thesis, RWTH Aachen, Forschungszentrum Jülich, ISBN 3-00-010003-2, February 2003.

[10] J.K. Hollingsworth, B.P. Miller, M.J.R. Gonçalves, O. Naim, Z. Xu, L. Zheng, MDL: a language and compiler for dynamic program instrumentation, in: Proceedings of the International Conference on Parallel Architectures and Compilation Techniques, San Francisco, California, 1997.

[11] A. Espinosa. Automatic performance analysis of parallel programs, Ph.D. Thesis, Universitat Autonoma de Barcelona, September 2000.

[12] J. Vetter. Performance analysis of distributed applications using automatic classification of communication inefficencies, in: Proceedings of the 14th International Conference on Supercomputing, Santa Fe, New Mexico, 2000, pp. 245–254.

[13] T. Fahringer, C. Seragiotto Júnior, Modeling and detecting performance problems for distributed and parallel programs with JavaPSL, in: Proceedings of the Conference on Supercomputers (SC2001), Denver, Colorado, 2001.

[14] Cray Research, Inc, Introducing the MPP Apprentice Tool, Cray Manual IN-2511, 1994.

[15] A. Arnold, U. Detert, W.E. Nagel, Performance optimization of parallel programs: tracing, zooming, understanding, in: R. Winget, K. Winget (Eds.), Proceedings of Cray User Group Meeting, Denver, CO, 1995, pp. 252–258.

[16] IBM Corporation, IBM AIX Parallel Environment: Operation and Use, IBM Corporation publication SH26-7231, 1996.

[17] M.T. Heath, Performance visualization with paragraph, in: Proceedings of the Second Workshop on Environments and Tools for Parallel Scientific Computing, Philadelphia, 1994, pp. 221–230.

[18] E. Shaffer, S. Whitmore, B. Schaeffer, D.A. Reed, Virtue: immersive performance visualization of parallel and distributed applications, IEEE Computer (1999) 44–51.

[19] Parallel Tools Consortium, xlcb Graphical Browser: User Manual. Available from <http://cs.oregonstate.edu/~pancake/ptools/lcb/xlcb.html>.

[20] B. Mohr, A. Malony, S. Shende, F. Wolf, Design and prototype of a performance tool interface for OpenMP, The Journal of Supercomputing 23 (2002) 105–128.

[21] S. Shende, A.D. Malony, J. Cuny, K. Lindlan. P. Beckman, S. Karmesin, Portable profiling and tracing for parallel scientific applications using C++, in: Proceedings of the SIGMETRICS Symposium on Parallel and Distributed Tools, ACM, 1998, pp. 134–145.

[22] K.A. Lindlan, J. Cuny. A.D. Malony, B. Mohr, R. Rivenburgh, C. Rasmussen, A tool framework for static and dynamic analysis of object-oriented software with templates, in: Proceedings of the Conference on Supercomputers (SC2000), Dallas, Texas, 2000.

[23] F. Wolf, EARL—Eine programmierbare Umgebung zur Bewertung paralleler Prozesse auf Message-Passing-Systemen, Diplomarbeit, RWTH Aachen, Forschugszentrum Jülich, Jül-Bericht 3551, June 1998.

[24] The Portland Group Compiler Technology, Product Documentation. Available from <http://www.pgroup.com/doc/index.htm>.

[25] Hitachi, Hitachi SR 8000 Compiler, Technical Manual.

[26] F. Wolf, B. Mohr, Specifying performance properties of parallel applications using compound events, Parallel and Distributed Computing Practices, Monitoring Systems and Tool Interoperability 4 (3) (2001) (special issue).

[27] T. Fahringer. M. Gerndt, B. Mohr, G. Riley, J.L. Träff, F. Wolf, Knowledge specification for automatic performance analysis, Tech. Rep. FZJ-ZAM-IB-2001-08, ESPRIT IV Working Group APART, Forschungszentrurn Jülich, revised version, August 2001.

[28] D.M. Beazley, Python Essential Reference, second ed., New Riders, 2001.

[29] J.K. Hollingsworth, M. Steele, Grindstone: a test suite for parallel performance tools, Computer Science Technical Report CS-TR-3703, University of Maryland, October 1996.

[30] F. Wolf, B. Mohr, Automatic performance analysis of MPI applications based on event traces, in: Proceedings of the European Conference on Parallel Computing (Euro-Par), Munich, Germany, 2000, pp. 123–132.

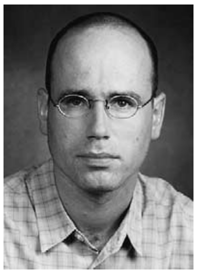
[31] Forschungszentrum Jülich, ZAMpano (ZAM Parallel Nodes). Available from <http://zampano.zam.kfa-juelich.de/>.

[32] T. Diehl, V. Gülzow, Performance of the parallelized regional climate model REMO, in: Proceedings of the Eighth ECMWF Workshop on the Use of Parallel Processors in Meterology, European Centre for Medium-Range Weather Forecasts, Reading, UK, 1998, pp. 181–191.

[33] Accelerated Strategic Computing Initiative (ASCI), The ASCI sweep3d Benchmark Code. Available from <http://www.llnl.gov/asci_benchmarks/>.

[34] M. Mihelcic, H. Wenzl, H. Wingerath, Flow in Czochralski crystal growth melts, Tech. Rep. Jül-2697, Forschungszentrum Jülich, December 1992.

[35] Forschungszentrum Jülich, Solute Transport in Heterogeneous Soil-Aquifer Systems. Available from <http://www.kfa-juelich.de/icg/icg4/Groups/Pollutgeosys/trace_e.html>.



Felix Wolf was born in Munich, Germany, in 1972. He studied computer science at Aachen University of Technology (RWTH Aachen). After that, he worked on parallel performance tools at the Research Centre Juelich until he obtained his Ph.D. from RWTH Aachen in 2003. In 2001, Felix Wolf visited the IBM T.J. Watson Research Center in New York for an internship. He recently joined the Innovative Computing Laboratory at the University of Tennessee. The current topic of his research is performance analysis based on hardware counters.

Bernd Mohr is a senior scientist at the Central Institute for Applied Mathematics of the Research Centre Juelich in Germany. His work focuses on tool support for parallel scientific computing. He is investigating performance instrumentation, measurement, and analysis tools since 1987. He received his Ph.D. degree from the University of Erlangen-Nuernberg, Germany, in 1992.