Automatic search for patterns of inefficient behavior in parallel applications

Felix Wolf [1], Bernd Mohr [2], Jack Dongarra [1], Shirley Moore [1]

[1] *University of Tennessee, ICL, 1122 Volunteer Blvd Suite, 413, Knoxville, TN 37996-3450, USA,* {fwolf, dongarra, shirley}@cs.utk.edu
[2] *Forschungszentrum Jülich, ZAM, 52425 Jülich, Germany,* b.mohr@fz-juelich.de

SUMMARY

Event tracing is a powerful method of analyzing the performance behavior of parallel applications. Because event traces record the temporal and spatial relationships between individual runtime events, they allow application developers to analyze dependences of performance phenomena across concurrent control flows. However, in view of the large amounts of data generated on contemporary parallel machines, the depth and coverage of a purely manual analysis is often limited. In this article, we present an approach of automatically searching event traces for execution patterns indicating inefficient behavior, allowing developers to study the performance of their applications on a very high level of abstraction, while consuming significantly less expert time than a manual analysis.

KEY WORDS: performance tools, event tracing, pattern search

1. Introduction

High performance computing is playing an increasingly critical role in advanced scientific research as simulation and computation are becoming widely used to augment and/or

*Correspondence to: Felix Wolf, University of Tennessee, Innovative Computing Laboratory, 1122 Volunteer Blvd Suite 413, Knoxville, TN 37996-3450, USA, fwolf@cs.utk.edu
Contract/grant sponsor: DOE SciDAC Program; contract/grant number: DE-FC02- 01ER25490
Contract/grant sponsor: DOE MICS Program; contract/grant number: DE-FG02-01ER25510
Contract/grant sponsor: DoD HPCMP PET Program; contract/grant number: N62306-01-7110
Contract/grant sponsor: European IST Working Group APART; contract/grant number: IST-2000-28077
Contract/grant sponsor: Helmholtz Research Programme Scientific Computing - Key Technologies

replace physical experiments. However, the gap between peak and achieved performance for scientific applications running on parallel systems has grown considerably in recent years. The complexity architecture of parallel systems and the interdependence between different components and layers in conjunction with communication structures imposed by the algorithm present difficult challenges for performance optimization of scientific applications. Tools are needed that collect and present relevant information on application performance on a high level of abstraction so as to enable developers to easily identify and determine the causes of performance bottlenecks.

Event tracing is a powerful method of analyzing the performance behavior of parallel applications. Time-stamped events, such as entering a function or sending a message, are recorded at runtime and analyzed afterward with the help of software tools. Graphical trace browsers, such as VAMPIR and Intel Trace Analyzer [3, 19], allow the fine-grained investigation of parallel performance behavior using a zoomable time-line display, as well as provide statistical summaries of communication behavior. Because event traces record the temporal and spatial relationships between individual runtime events, they allow application developers to analyze dependences of performance phenomena across concurrent control flows.

However, in view of the large amounts of data generated on contemporary parallel machines, the depth and coverage of the visual analysis offered by a browser is limited as soon as it targets more complex patterns not included in the statistics generated by such tools. In this article, we present an alternative to manually scanning the time-line display. Our approach automatically searches event traces of MPI and OpenMP programs for execution patterns indicating inefficient behavior, allowing developers to study the performance of their applications on a very high level of abstraction, while consuming significantly less expert time than a manual analysis. The abstraction level is achieved through specifying complex patterns that embody higher-level behavior related to the parallel programming model. Moreover, using wavefront algorithms as an example, we show that the semantic level of these patterns can be further increased by correlating their occurrences with performance-critical phases of the parallelization strategy used in an application. Those phases manifest themselves as another class of patterns in the event trace and can be recognized using knowledge of virtual adjacency relationships between individual processes.

Our approach constitutes the core of the KOJAK toolkit [29] and is primarily implemented in the EXPERT trace analyzer component [11]. EXPERT maps the execution-time penalty caused by each pattern onto a three-dimensional space consisting of three hierarchical dimensions: (i) performance property [†], (ii) call path, and (iii) system resource (e.g., a process). A graphical browser allows a convenient in-depth study of this space on varying levels of granularity.

The remainder of this document is organized as follows: After considering related work in Section 2, we give a brief overview of the KOJAK toolkit and explain the basic process of analyzing a trace file in Section 3. Then, in Section 4, we illustrate how patterns of inefficient execution can be specified and study the underlying abstraction mechanisms along with a realistic example. The benefits of using topological knowledge to explain the occurrence of

[†] *Performance property* is a more general term used instead of "performance problem" that also includes non-negative performance aspects such as computation.

inefficiency patterns in terms of the parallelization strategy of a program is covered in Section 5. Finally, we present a conclusion followed by an outlook on future work in Section 6.

2. Related Work

This work is embedded in the ESPRIT/IST working group APART. One of the core results of APART is the APART Specification Language (ASL) [12], which provides a formal notation to describe performance properties of parallel applications. Performance properties represent a not necessarily negative aspect of an application's performance, such as such as synchronization or computation. In ASL, performance properties are specified as conditions referencing performance-related data, such as source-code entities or certain measurements, in a uniform way by means of an object-oriented data model. A severity expression quantifies a property's impact on the overall performance, while a confidence value quantifies the condition's reliability. The notion of a performance property strongly influenced EXPERT's modular architecture, which is based on encapsulating each performance property to look for in a separate C++ class offering methods to control its evaluation. Conversely, the EXPERT approach inspired mechanisms in ASL to define performance properties based on trace data.

Also stimulated by ASL, Fahringer et al. designed a language called JavaPSL [13] to specify performance properties in the Aksum tool based on the Java programming language. 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) using advanced concepts of the Java language, such as polymorphism, abstract classes, and reflection.

Espinosa implemented an automatic trace analysis tool KAPPA-PI [10] 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.

Gerndt and Krumme developed a rule-based approach [14] to automatic performance analysis of programs on shared–virtual-memory environments, such as SVM Fortran [5]. The analysis process is specified as a rule base consisting of refinement rules and proof rules. Similar to EXPERT, their approach advocates a clear separation between the analysis process as represented by refinement rules and knowledge about potential performance problems as represented by the proof rules.

Vetter automatically identified wait states in MPI point-to-point communication based on machine learning techniques [25]. 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.

An alternative approach to describing complex event patterns was devised by Bates [4]. The proposed Event Definition Language (EDL) focuses on specifying incorrect behavior of distributed systems. It allows compound events to be defined in a declarative manner based on extended regular expressions, where primitive events are clustered to higher-level events using certain formation operators. However, EDL it's suitability for compound events that are associated with some kind of state, such as those targeted by EXPERT, is limited.

The multidimensional hierarchical decomposition of the search space for performance problems has a long tradition. Miller et al. developed the W^3 Search Model [20] as the basis of the on-line performance-analysis performed by Paradyn. The W^3 model describes performance behavior along the dimensions performance problem, various 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 [17]. 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.

Topological knowledge has been used earlier to highlight certain aspects of parallel performance. Ahn and Vetter mapped counter data onto the virtual topology of the SWEEP3D benchmark to identify clusters of related behavior by statistical means [2]. Müllender visualized different network topologies including up to three-dimensional grids and mapped certain communication parameters onto their nodes to better observe communication activities in virtual shared memory systems [22].

Also, topological knowledge has been used for semantic debugging of parallel applications. Huband and McDonald describe a trace-based debugger called DEPICT that exploits topological information to identify processes with logically similar behavior in traces of MPI applications and to display semantic differences among these groups [18]. The comparison is based on the order and number of events. Also interesting to our work is DEPICT's ability to automatically identify the virtual topology using graph-distance measures.

3. The KOJAK Toolkit

KOJAK is an automatic performance evaluation system for MPI, OpenMP, and hybrid applications written in C/C++ or Fortran. KOJAK generates event traces from running applications and automatically searches them offline for execution patterns indicating inefficient performance behavior. KOJAK is jointly developed by Forschungszentrum Jülich, Germany, and the University of Tennessee, USA.

Figure 1 gives an overview of KOJAK's architecture, its components, and the overall process of analyzing a trace file. The process involves three major parts: (i) a semi-automatic multi-level instrumentation of the user application, followed by (ii) the execution on the target platform, and finally (iii) the automatic analysis of the generated trace file.

The component mainly responsible for trace generation is the EPILOG runtime system. The event traces generated by this process capture MPI point-to-point and collective communication

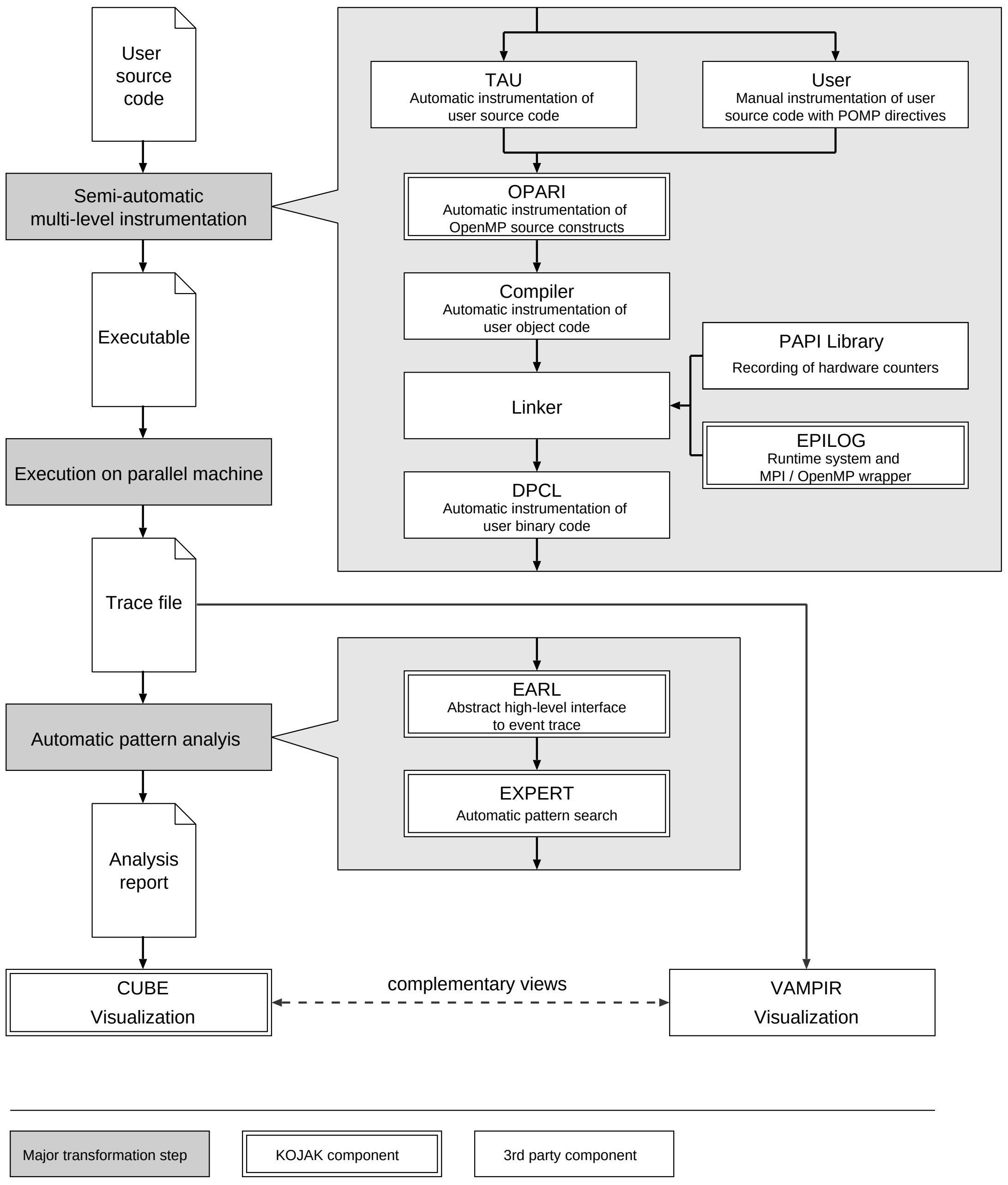

User source code
Semi-automatic multi-level instrumentation
Executable
Execution on parallel machine
Trace file
Automatic pattern analyis
Analysis report
CUBE Visualization
complementary views
VAMPIR Visualization
TAU Automatic instrumentation of user source code
User Manual instrumentation of user source code with POMP directives
OPARI Automatic instrumentation of OpenMP source constructs
Compiler Automatic instrumentation of user object code
PAPI Library Recording of hardware counters
Linker
EPILOG Runtime system and MPI / OpenMP wrapper
DPCL Automatic instrumentation of user binary code
EARL Abstract high-level interface to event trace
EXPERT Automatic pattern search
Major transformation step
KOJAK component
3rd party component

Figure 1. KOJAK architecture.

as well as OpenMP parallelism change, parallel constructs, and synchronization. In addition, data from hardware counters accessed using the PAPI library [7] can be recorded in the event traces. To make measurements with the EPILOG system, the application must be instrumented at specific points to activate EPILOG library calls. These points usually include the entries and exits of various code regions, such as user regions, OpenMP constructs, and MPI calls.

Automatic instrumentation of user regions is supported in three different ways, depending on the availability of certain compilers and third-party tools: using (i) a compiler-supplied profiling interface, using (ii) TAU [23], or using DPCL [9]. In addition, the user is free to instrument user regions manually by placing POMP directives after the entry point and before all exit points. The POMP directives are later processed by OPARI [21], which is also responsible for the automatic instrumentation of OpenMP constructs. MPI functions are instrumented fully automatically by interposing a wrapper library.

During execution, the instrumented code generates several trace files, one for each process or thread. Execution time overheads ranging from 1 to 10 % are reported in [27]. After termination, the local traces are merged into a single global trace file. The global trace is then subject to an off-line analysis performed by KOJAK's EXPERT component, which attempts to identify specific performance properties. Internally, EXPERT represents performance properties in the form of execution patterns that model inefficient behavior. These patterns are used during the analysis process to recognize and quantify inefficient behavior in the application. The performance properties addressed by EXPERT include inefficient use of the parallel programming models MPI and OpenMP as well as low CPU and memory performance. The analysis process automatically transforms the traces into a compact call-path profile that includes the execution time penalties caused by the different patterns broken down by call path and process or thread. Section 4 covers the pattern analysis in more detail.

The call-path profile can be viewed in the CUBE performance browser (Figure 5). CUBE is a generic tool to display a multidimensional performance space consisting of the dimensions (i) performance property, (ii) call path, and (iii) system resource. Each dimension is represented as a tree browser that can be collapsed or expanded to achieve the desired level of granularity or specialization. The tree browsers are coupled such that the penalty caused by a particular performance property can be broken down by call path and process or thread.

4. Pattern Analysis

The central idea behind the KOJAK approach is to identify performance properties by searching event traces recorded during execution for patterns of inefficient behavior. This allows (i) classifying the behavior that leads to a performance degradation and (ii) quantifying its impact on the overall performance. The particular way EXPERT specifies these patterns internally enables us to capture very complex situations not covered by the aforementioned trace-visualization tools or by profiling tools, such as mpiP [26].

EXPERT specifies patterns as *compound events*. A compound event is a set of events appearing in the trace file that satisfy conditions related to the situation the pattern describes. These conditions are expressed in terms of an event model suitable to describe the execution of a parallel program. The execution of a program, as represented by an event trace, is modeled as a chronologically sorted sequence of events representing actions relevant to the purpose of the observation.

Each action is represented by a different event type. Each event type is defined by a set of attributes characterizing the action represented by the type. Actions of interest are sending and receiving point-to-point messages; entering and exiting different types of code regions, such as user functions, OpenMP constructs, and MPI (collective) operations; and synchronization operations, such as acquiring and releasing OpenMP locks. All event types have a location (i.e., the process or thread) as well as a wall-clock time attribute. The basic structure of the event trace along with the applied event-type system is called the *basic event model*.

4.1. Abstraction Mechanisms

As the compound events targeted by our analysis often involve complex inter-event relationships referring to certain aspects of the execution state, such as message queues between different locations or call stacks at the same location, the basic event model is not sufficient to describe the corresponding patterns in a convenient way because the only relationship provided is the global temporal order.

To be better able to express complex relationships among the constituents of a compound event, the basic event model needs to be extended by adding higher-level abstractions defined on top of the basic event model. These abstractions are implemented in the EARL trace access layer [28], which is a class library used by EXPERT during the pattern search to access the trace file (Figure 1, above EXPERT). The main purpose of EARL is to simplify the specification of execution patterns representing performance properties within the EXPERT analyzer and, thus, to allow an easy extension and customization of the pattern base used in the analysis process. As opposed to a raw trace file that allows reading individual event records in a sequential manner, EARL offers random access to individual events and two different categories of abstractions:

- State sequences
- Pointer attributes

State sequences reflect different aspects of the program's overall execution state. The overall execution state consists of a set of (component) states, each of which represents one aspect of the overall state, such as the call stack or the message queue. EARL models each component state as a set of events. These sets are stepwise transformed by the sequence of events making up the trace file. That is, an event causes a state transition altering the event set representing the component state by either removing elements and/or adding itself to the set. Thus, for every component state, an event trace defines a *state sequence*. The initial state is always the empty set. Transition rules define how a state is transformed by an event into its successor state. For example, EARL maintains a message queue for every pair of processes. The initial queue is empty. Whenever a send event occurs, it is added to the queue, and whenever a receive event occurs, the corresponding send event is removed from the queue. Note that the state set derives its queue structure from the implicit ordering of events. Other state sequences describe MPI collective communication, OpenMP parallel operations and lock synchronization, region stacks, and the call tree. Pointer attributes are event attributes that refer to another

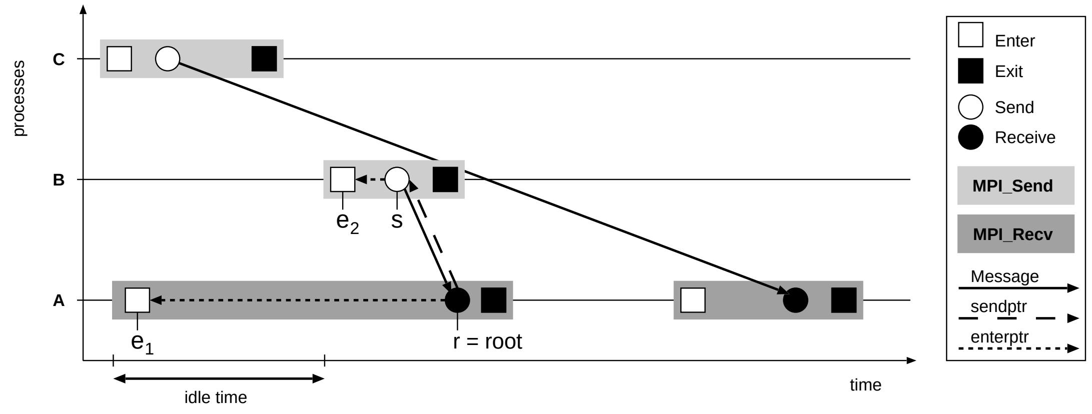


Figure 2. Time-line view of the late-sender compound event.

related event. For example, receive events provide an attribute called *sendptr* that points to the corresponding send event. The implementation of pointer attributes makes use of state sequences. Other pointer attributes connect corresponding enter and exit events, link actions on the same lock, and encode call-path information. The state sequences and pointer attributes provided by EARL are formally defined in [27], a comprehensive documentation of the EARL API can be found in [28].

4.2. Compound Events

To analyze an application's performance behavior during a program run, EXPERT walks sequentially through the trace file and tries to match the patterns that have been previously specified in the form of compound events. To illustrate how compound events are specified and detected, Figure 2 shows the time-line view of a situation called *late sender*. Process A waits for a message from process B that is sent much later than the receive operation has been started. Therefore, most of the time consumed by the receive operation of process A is actually idle time that could be used better. EXPERT recognizes this pattern by waiting for a receive event to appear in the event stream. After capturing one, EXPERT follows pointer attributes computed by EARL (dashed lines in Figure 2) to the enter events of the two communication operations to determine the temporal displacement between these two events (idle time in Figure 2).

Conceptually, a compound event specification consists of three parts: (i) a root event type, (ii) an instantiation part, and (iii) constraints. Whenever EXPERT encounters an instance of the root event type, it stops and tries to match the specified compound event. The matching involves two parts. The first part consists of locating the remaining constituents using state information and pointer attributes. This is specified in the instantiation part. The second part

```
ROOT
                    RECV  $r$;
INSTANTIATION
                    $s_1$  $:= r.sendptr$;
                    $e_1$  $:= r.enterptr$;
                    $e_2$  $:= s_1.enterptr$;
CONSTRAINT
                    $e_1.region$   $=$ MPI_Recv  $\wedge$
                    $e_2.region$   $=$ MPI_Send  $\wedge$
                    $e_1.time$     $< e_2.time$
```

Figure 3. Formal specification of the late-sender compound event.

is optional and includes checking additional constraints that need to be satisfied in order to qualify for a match.

A specification of the late-sender situation is given in Figure 3 in a pseudo notation. The instantiation part starts from the root event (r), which must be a receive event, and follows pointer attributes to identify the remaining constituents (s_1, e_1, e_2). The pointer attributes involved are *sendptr*, which points from a receive event to the corresponding send event, and *enterptr*, which points to the enter event on the top of the call stack (i.e., enter event of the current region instance). The constraint part requires the communication operations to be of synchronous type with the receive operation being posted earlier than its sending counterpart. Internally, compound event specifications are written as C++ or Python classes[‡] that provide callback methods to be called upon occurrence of specific event types (root declaration) in the event stream. A pattern class registers itself for the root event type, whose instances are then provided as argument to the call-back method. When being called, the method body performs the instantiation of the compound event along with an optional constraint check. As a result of the trace-access model provided by EARL, the code of the call-back methods can be kept simple by using expressions very similar to the pseudo notation shown above. The simplicity is derived from the fact that all higher-level abstractions, such as execution states and links between related events, are expressed in terms of event sets or event references, thus never leaving the familiar notion of an event.

Looking at late-sender situation depicted in Figure 2 in the context of the other message sent from process C to A allows the conclusion that the late-sender pattern could have been avoided or at least alleviated by reversing the acceptance order of these two messages. Because the message from C is sent earlier than that from B, it will in all probability have reached process A earlier. So instead of waiting for the message from B, A could have used the time

[‡]At present, we maintain two versions of the analyzer: one in C++ for ease of installation and performance and one in Python for design studies.

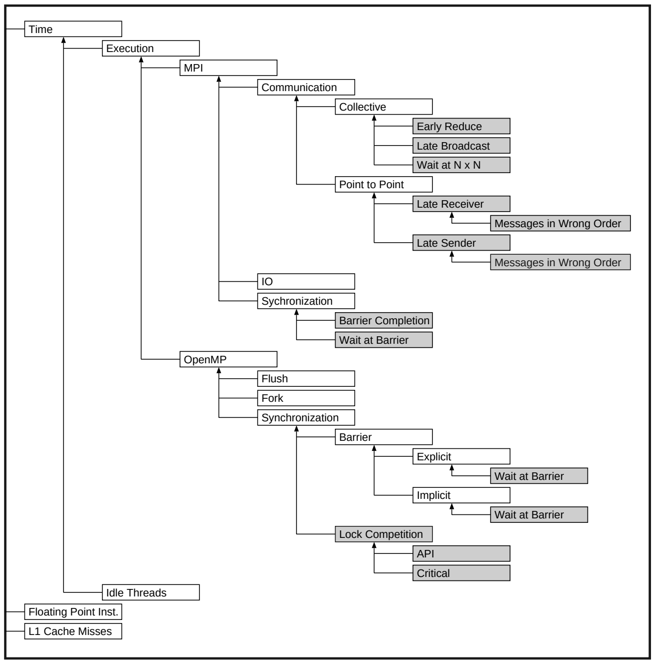
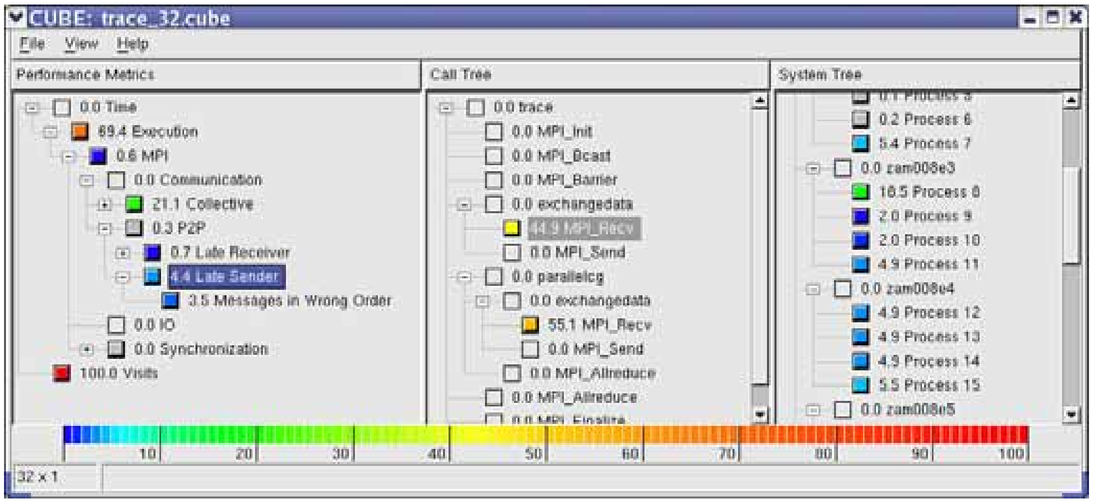


Figure 4. Pattern hierarchy.

better by accepting the message from C first. The late-sender pattern in this context is called *late-sender / wrong-order*. EXPERT recognizes this situation by examining the execution state computed by EARL at the moment when A receives the message from B. It inspects the queue of messages (i.e., their send events) sent to A and checks whether there are older messages than the one just received. In the figure, the queue would contain the event of sending the message from C to A.

The patterns recognized by EXPERT are organized in a specialization hierarchy, as shown in Figure 4, with patterns referring to rather general performance properties at the top and rather specific properties at the bottom. There are two types of patterns: (i) simple profiling patterns (white) based on how much time or some other metric (e.g., cache misses) is spent in certain MPI calls or code regions, and (ii) patterns describing complex inefficiency situations (gray) usually described by more complex compound events (e.g., late sender in point-to-point communication or synchronization delay before all-to-all operations). There are complex patterns defined for MPI-1 as well as OpenMP. A complete description of the patterns supported so far can be found in [27]. Work is also underway to integrate patterns related to MPI-2 RMA operations [15] as well as to loop-level memory bottlenecks.



```
Performance Metrics
0.0 Time
  69.4 Execution
    0.6 MPI
      0.0 Communication
        21.1 Collective
        0.3 P2P
          0.7 Late Receiver
          4.4 Late Sender
            3.5 Messages in Wrong Order
      0.0 IO
      0.0 Synchronization
100.0 Visits

Call Tree
0.0 trace
  0.0 MPI_Init
  0.0 MPI_Bcast
  0.0 MPI_Barrier
  0.0 exchangedata
    44.9 MPI_Recv
    0.0 MPI_Send
  0.0 parallelcg
    0.0 exchangedata
      55.1 MPI_Recv
      0.0 MPI_Send
    0.0 MPI_Allreduce
  0.0 MPI_Allreduce
  0.0 MPI_Finalize

System Tree
    0.1 Process 5
    0.2 Process 6
    5.4 Process 7
  0.0 zam008e3
    18.5 Process 8
    2.0 Process 9
    2.0 Process 10
    4.9 Process 11
  0.0 zam008e4
    4.9 Process 12
    4.9 Process 13
    4.9 Process 14
    5.5 Process 15
  0.0 zam008e5
```

Figure 5. Performance results for MPI application TRACE.

Each pattern calculates a (call path, location) matrix containing the time incurred by the application as a result of a specific property in a particular (call path, location) pair, where a location is a process or thread. Thus, EXPERT maps the (performance property, call path, location) space onto the losses caused by a particular performance property, while the program was executing in a particular call path at a particular location. After finishing the analysis, the mapping is written to a file and can be viewed using the CUBE display tool.

Figure 5 shows the results for an MPI application called TRACE, which simulates the subsurface water flow in variably saturated porous media. The application was executed with 32 processes on a Linux cluster with 8 Pentium III Xeon (550 MHz) 4-way nodes. To reduce the size of the trace file, the instrumentation of user functions was restricted to the main solver routine `parallelcg()` and a function responsible for exchanging buffer contents used inside and outside the main solver (`exchangedata()`).

Almost 8 % of the overall execution time was spent waiting in a late-sender situation caused by `exchangedata()`, a little less than half of it can be attributed to the main solver's operation. However, when executing not on behalf of `parallelcg()`, `exchangedata()` called the receive operation only a few times, pointing to a small number of larger late-sender instances. Also, a significant fraction of the overall late-sender time (3.5 % of execution time) could be classified as the more specialized wrong-order pattern.

Recent work has taken advantage of the specialization relationships to obtain a significant speed improvement for EXPERT and to allow more compact pattern specifications [30]. In the previous version, patterns have only been able to register for primitive events, that is, events as they appear in the event stream, as opposed to compound events consisting of multiple primitive events. The new design allows patterns also to publish compound events detected by themselves as well as to register for compound events detected by others. Transferred to our example, the simple late-sender class publishes all pattern instances it detects. Conversely, the class describing the combined situation late-sender / wrong-order registers for these instances and, then, upon receiving such an instance, only needs to check the message queue, as described earlier. The benefit is twofold: a more compact specification, as the late-sender pattern itself does not need be repeated, and a reduction of work because the matching of the simple late-sender is performed only once.

EXPERT is designed in a modular fashion, separating the pattern specifications from the actual analysis process. Internally, the semantics of individual pattern classes are hidden behind a common class interface, which makes it easy to modify existing patterns or to extend the current pattern base, for example, in order to integrate application-specific patterns, a feature beneficially utilized in Section 5.

5. Virtual Topologies

In many parallel applications, each process (or thread) communicates only with a limited number of other processes. For example, a simulation modeling the spread of pollutants in the environment might decompose the overall simulation domain to smaller pieces and assign each of them to a single process. Given this distribution, a process would then only communicate with processes owning subdomains adjacent to its own. The mapping of data onto processes and the neighborhood relationship resulting from this mapping is called *virtual topology*. In general, a virtual topology is specified as a graph. Many applications use Cartesian topologies, such as two- or three-dimensional grids. Virtual topologies can include processes or threads, depending on the programming model being used. We argue that topological knowledge can help identify performance problems more effectively, especially as many parallel algorithms are parametrized in terms of a virtual topology and this topology often influences the order in which certain computations are performed.

In the Section 4, we showed that automatic pattern analysis in event traces can help generate high-level feedback on an application's performance. We identified wait states recognizable by temporal displacements between individual events across multiple processes or threads but without utilizing any information on logical adjacency between processes or threads. We now show that enriching the information contained in event traces with topological knowledge significantly raises the abstraction level of the feedback returned. In particular, we demonstrate that topological information allows the following: (i) identifying higher-level events related to distinct phases of the parallelization scheme applied in an application in order to refine the present classification of wait states targeted by our pattern analysis and (ii) exposing correlations of these wait states with the topological characteristics of affected processes by visually mapping their severity onto the virtual topology.

As an example, we show correlations between late-sender instances and certain phases in wavefront algorithms, a popular parallelization scheme used to solve particle transport problems. KOJAK has recently been made capable of recording topological information as part of the event trace and of visualizing the severity of the analyzed behaviors in a topological

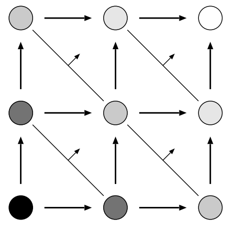


Figure 6. Wavefront propagation of data in SWEEP3D.

display [6]. To keep this extension simple, we restricted ourselves to Cartesian topologies as a common case found in many of today's parallel applications.

The benchmark code SWEEP3D [1] is an MPI program performing the core computation of a real ASCI application. It solves a 1-group time-independent discrete ordinates (Sn) 3D Cartesian geometry neutron transport problem by calculating the flux of neutrons through each cell of a three-dimensional grid (i, j, k) along several possible directions (angles) of travel. The angles are split into eight octants, each corresponding to one of the eight directed diagonals of the grid. To exploit parallelism, SWEEP3D maps the (i, j) planes of the three-dimensional domain onto a two-dimensional grid of processes. The parallel computation follows a pipelined wavefront process that propagates data along diagonal lines through the grid. Figure 6 shows the data-dependence graph for a 3×3 grid. The long arrows symbolize data dependencies, while diagonal lines cut through algorithmically independent processes and represent the computation as it progresses in the form of "wavefronts" from the lower left to the upper right corner (short arrows). The actual direction of the wavefront is determined by the particular angle or octant being processed at a given moment.

The code initiates wavefronts from all four corners of the two-dimensional grid of processes. The wavefronts are pipelined to enable multiple wavefronts to follow each other along the same direction simultaneously. Performance models of wavefront processes, in particular as they appear in SWEEP3D, have been extensively studied [16, 24]. With EXPERT we analyze the characteristics of wavefront communication from an experimental viewpoint with emphasis on wait states resulting from the data dependencies illustrated in Figure 6.

Although parallel operation in SWEEP3D can be very efficient once the pipeline is filled, the opportunity for parallelism is limited whenever the direction of the wavefront changes and the pipeline has to be refilled, although the algorithm allows for some overlap between pipelines in different directions. SWEEP3D uses `MPI_Recv()` calls that are likely to block whenever the pipeline is refilled and the calling process is distant from the pipeline's origin. This phenomenon is a specific instance of the late-sender pattern discussed earlier.

To investigate this type of behavior, we extended the pattern base normally used by our EXPERT analysis tool and added four patterns describing the occurrence of late-sender instances at the moment of a pipeline direction change (i.e., a refill), one pattern for each direction (i.e., SW, NW, NE, SE). The direction change is recognized by maintaining for every process a

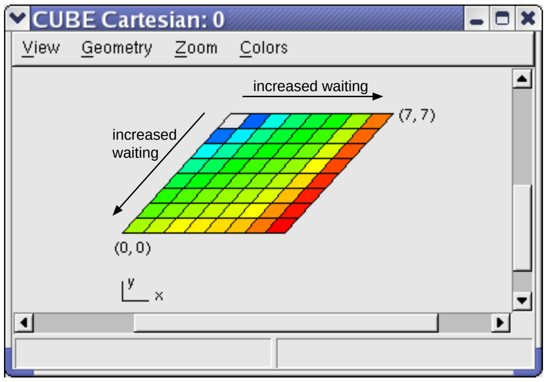


Figure 7. Distribution of late-sender wait states as a result of pipeline refill from North-West.

FIFO queue that records the directions of messages received. For this purpose, the direction of every message is calculated using topological information. Since the wavefronts propagate along diagonal lines, as depicted in Figure 6, each wavefront direction has a horizontal as well as a vertical component, involving messages in two different orthogonal directions. We therefore need to consider two potential wait states at the moment of a direction change, each resulting from one of the two direction components. However, special attention has to be paid to processes located at the border of the grid (Figure 6). Because they have only a limited number of neighbors, their inbound as well as their outbound communication may be restricted to one direction only, depending on their position relative to the wavefront propagation.

To validate our design, we chose a problem size 512 × 512 × 150 grid points and ran the application with 64 processes on a Solaris Cluster equipped with UltraSPARC-III 750 MHz processors. The topology was recorded by inserting EPILOG API calls into the module responsible for the domain decomposition. The total time spent in late-sender wait states was 25.4%. Late sender instances observed simultaneously with a pipeline direction change account for about 60% of the overall late-sender time. The times measured for individual directions vary between 5.6% of total execution time for pipeline refill from North-West and 2.2% for refill from North-East. Figure 7 shows the new topology view rendering the distribution of late-sender times for pipeline refill from North-West (i.e., upper left corner). The colors are assigned relative to the maximum and minimum wait times for this particular pattern. As can be seen, the corner reached by the wavefront last incurs most of the waiting times, whereas processes closer to the origin of the wavefront incur less.

Note that the specifications of our patterns do not make any assumption about the specifics of the computation performed, and should therefore be applicable to a broad range of wavefront applications. Moreover, although the current implementation applies to wavefront processes based on a two-dimensional domain decomposition, we assume that it can be easily adapted to a three-dimensional decomposition by considering wavefronts propagating along three orthogonal direction components instead of two.

6. Conclusion

As many performance problems of parallel applications involve behavioral dependencies between concurrent control flows, trace analysis is an effective way of identifying undesired wait states that are not obvious without considering temporal and spatial relationships between runtime events across different processes and threads. Automating this process can increase programmer productivity and reduce time to solution by reducing both development time and execution time after providing high-level feedback on an application's performance.

We showed that by selecting appropriate abstraction mechanisms, complex performance problems can be specified in the form of patterns to be used as prerequisite for their efficient automatic detection. Using this method, we found evidence of wait states resulting from an inefficient use of the parallel programing model in real applications. Moreover, such wait states can be correlated with distinct phases of the parallelization strategy applied in a program by utilizing knowledge of the virtual process topology. The modular design of our detection tool allowed us to easily extend the base of predefined patterns and to demonstrate this correlation for wavefront algorithms using algorithm-specific patterns.

While our approach gives automatic performance feedback on a significantly higher level than traditional tools, its dependence on the collection of trace files constrains its scalability on present and future architectures consisting of thousands of processors. Therefore, our research in this area will focus on applying parallel and distributed processing approaches to the processing, reduction, and filtering of trace data including smart instrumentation recording only relevant sections of program execution and disregarding repetitive behavior.

REFERENCES

1. Accelerated Strategic Computing Initiative (ASCI). *The ASCI sweep3d Benchmark Code.* `http://www.llnl.gov/asci_benchmarks/`.
2. D. H. Ahn and J. S. Vetter. Scalable Analysis Techniques for Microprocessor Performance Counter Metrics. In *Proc. of the Conference on Supercomputers (SC2002)*, Baltimore, November 2002.
3. 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.
4. P. C. Bates. *Debugging Programs in a Distributed System Environment.* PhD thesis, University of Massachusetts, February 1986.
5. R. Berrendorf, M. Gerndt, and A. Krumme. A Programming Environment for Parallel Computers with Global Address Space. In *Proc. of the Workshop on High-Level Programming Models and Supportive Environments (HIPS), in combination with IPPS.* IEEE, 1996.
6. N. Bhatia, F. Song, F. Wolf, B. Mohr, J. Dongarra, and S. Moore. Automatic Experimental Analysis of Communication Patterns in Virtual Topologies. In *Proc. of the International Conference on Parallel Processing (ICPP)*, Oslo, Norway, June 2005. Submitted for publication.
7. S. Browne, J. Dongarra, N. Garner, G. Ho, and P. Mucci. A Portable Programming Interface for Performance Evaluation on Modern Processors. *The International Journal of High Performance Computing Applications*, 14(3):189–204, 2000.

8. H. W. Cain, B. P. Miller, and B. J. N. Wylie. A Callgraph-Based Search Strategy for Automated Performance Diagnosis. In *Proc. of the 6th International Euro-Par Conference*, volume 1999 of *Lecture Notes in Computer Science*, Munich, Germany, August/September 2000. Springer.
9. L. DeRose, T. Hoover Jr., and J. K. Hollingsworth. The Dynamic Probe Class Library - An Infrastructure for Developing Instrumentation for Performance Tools.
10. A. Espinosa. *Automatic Performance Analysis of Parallel Programs.* PhD thesis, Universitat Autonoma de Barcelona, September 2000.
11. B. Mohr F. Wolf. EPILOG Binary Trace-Data Format. Technical Report FZJ-ZAM-IB-2004-06, Forschungszentrum Jülich, May 2004.
12. T. Fahringer, M. Gerndt, B. Mohr, F. Wolf, G. Riley, and J. L. Träff. Knowledge Specification for Automatic Performance Analysis. Technical Report FZJ-ZAM-IB-2001-08, ESPRIT IV Working Group APART, Forschungszentrum Jülich, August 2001. Revised version.
13. T. Fahringer and C. Seragiotto Júnior. Modelling and Detecting Performance Problems for Distributed and Parallel Programs with JavaPSL. In *Proc. of the Conference on Supercomputers (SC2001)*, Denver, Colorado, November 2001.
14. M. Gerndt, A. Krumme, and S. Özmen. Performance Analysis for SVM-Fortran with OPAL. In *Proc. of the Int. Conf. on Parallel and Distributed Processing Techniques and Applications (PDPTA'95)*, pages 561–570, Athens, Georgia, 1995. IEEE.
15. M.-A. Hermanns, B. Mohr, and F. Wolf. Event-based Measurement and Analysis of One-sided Communication. In *Proc. of the European Conference on Parallel Computing (Euro-Par)*, Lecture Notes in Computer Science, Lisboa, Portugal, August-September 2005. Springer.
16. A. Hoisie, O. Lubeck, and H .Wasserman. Performance Analysis of Wavefront Algorithms on Very-Large Scale Distributed Systems. In *Lectures Notes in Control and Information Sciences*, volume 249, page 171, 1999.
17. J. K. Hollingsworth, B. P. Miller, M. J. R. Gonçalves, O. Naim, Z. Xu, and L. Zheng. MDL: A Language and Compiler for Dynamic Program Instrumentation. In *Proc. of the International Conference on Parallel Architectures and Compilation Techniques*, San Francisco, California, November 1997.
18. S. Huband and C. McDonald. A Preliminary Topological Debugger for MPI Programs. In R. Buyya, G. Mohay, and P. Roe, editors, *Proc. of the First IEEE/ACM International Symposium on Cluster Computing and the Grid*, pages 422–429. IEEE Computer Society, 2001.
19. Intel. *Intel Cluster Tools.* `http://www.intel.com/software/products/cluster/index.htm`.
20. 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.
21. B. Mohr, A. Malony, S. Shende, and F. Wolf. Design and Prototype of a Performance Tool Interface for OpenMP. *The Journal of Supercomputing*, 23:105–128, August 2002.
22. C. Müllender. Visualisierung der Speicheraktivitäten von parallelen Programmen in Systemen mit virtuell gemeinsamen Speicher. Master's thesis, RWTH Aachen, Forschungszentrum Jülich, May 1994.
23. S. S. Shende. *The Role of Instrumentation and Mapping in Performance Measurement.* PhD thesis, University of Oregon, August 2001.
24. D. Sundaram-Stukel and M. K. Vernon. Predictive Analysis of a Wavefront Application Using LogGP. In *Proc. 7th ACM SIGPLAN Symp. on Principles and Practices of Parallel Programming (PPoPP '99)*, pages 141–150, Atlanta, GA, May 1999.
25. 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.
26. J. Vetter and C. Chambreau. *mpiP: Lightweight, Scalable MPI Profiling*, 2004. `http://www.llnl.gov/CASC/mpip/`.
27. F. Wolf. *Automatic Performance Analysis on Parallel Computers with SMP Nodes.* PhD thesis, RWTH Aachen, Forschungszentrum Jülich, February 2003. ISBN 3-00-010003-2.
28. F. Wolf. EARL - API Documentation. Technical Report ICL-UT-04-03, University of Tennessee, Innovative Computing Laboratory, October 2004.
29. F. Wolf and B. Mohr. Automatic performance analysis of hybrid MPI/OpenMP applications. *Journal of Systems Architecture*, 49(10-11):421–439, 2003. Special Issue "Evolutions in parallel distributed and network-based processing".
30. F. Wolf, B. Mohr, J. Dongarra, and S. Moore. Efficient Pattern Search in Large Traces through Successive Refinement. In *Proc. of the European Conference on Parallel Computing (Euro-Par)*, Pisa, Italy, August - September 2004.