A Pattern-Based Approach to

Automated Application Performance Analysis

Nikhil Bhatia, Shirley Moore, Felix Wolf, and Jack Dongarra
Innovative Computing Laboratory
University of Tennessee
(bhatia, shirley, fwolf, dongarra}@cs.utk.edu

Bernd Mohr
Zentralinstitut für Angewandte Mathematik
Forschungszentrum Jülich
b.mohr@fz-juelich.de

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 replace physical experiments. However, the gap between peak and achieved performance for scientific applications running on high-end computing (HEC) systems has grown considerably in recent years. The complex architectures and deep memory hierarchies of HEC systems present difficult challenges for performance optimization of scientific applications. Developers of scientific applications for HEC systems are not necessarily experts in high performance computing architectures and performance analysis. For this reason, performance data at the level of un-interpreted hardware counter data or communication statistics or traces may not be useful to these developers. Higher level abstractions that identify various types of performance problems, such as inefficient use of the memory hierarchy or excessive synchronization delay for example, and that map these problems to the relevant application source code, will be much more useful and allow performance tuning to be done with much less time and effort.

Event tracing is a well-accepted technique for post-mortem performance analysis of parallel applications. Time-stamped events, such as entering a function or sending a message, are recorded at runtime and analyzed afterwards with the help of software tools.

Visualization tools such as Vampir and Intel Trace Analyzer [1], Jumpshot [2], and Paraver [3] can provide a graphical view of the state changes and message passing activity represented in the trace file, as well as provide statistical summaries of communication behavior. However, it is difficult and time-consuming for even expert users to identify performance problems from such a view or from large amounts of statistical data. Automated analysis of event traces can provide the user with the desired information more quickly by transforming the data into a more compact representation at a higher level of abstraction. The KOJAK toolkit [4] supports performance analysis of MPI and/or OpenMP application by automatically searching traces for execution patterns that indicate inefficient behavior. The performance problems addressed include inefficient use of the parallel programming model and low CPU and memory

performance. This presentation will summarize KOJAK's pattern matching approach and give two examples of our recent work on defining new patterns -- one for detecting communication inefficiencies in wavefront algorithms and another for detecting memory bound nested loops.

Figure 1 gives an overview of KOJAK's architecture and its components. The KOJAK analysis process is composed of two parts: a semi-automatic multi-level instrumentation of the user application followed by an automatic analysis of the generated performance data.

Figure 1. KOJAK architecture

The event traces generated by KOJAK's tracing library EPILOG capture MPI point-to-point and collective communication as well as OpenMP parallelism change, parallel constructs, and synchronization. In addition, data from hardware counters accessed using the PAPI library [5,6] can be recorded in the event traces. KOJAK's EXPERT tool is an automatic trace analyzer that attempts to identify specific performance problems. EXPERT represents performance problems 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. Internally patterns are specified as C++ classes that provide callback methods to be called upon occurrence of specific event types in the event stream. The pattern classes are organized in a specialization hierarchy, as shown in Figure 2. There are two types of patterns: 1) simple profiling patterns based on how much time or some other metric (e.g., cache misses) is spent in certain MPI calls or code regions, and 2) patterns describing complex inefficiency situations usually described by multiple events – e.g., late sender in point-to-point communication or synchronization delay before all-to-all operations. Recent work has taken advantage of the specialization relationships to obtain a significant speed improvement for EXPERT and to allow more compact pattern specifications [7]. Each pattern calculates a (call path, location) matrix containing the time spent on a specific behavior in a particular (call path, location) pair, where a location is a process or thread. Thus, EXPERT maps the (performance problem, call path, location) space onto the time spent on a particular performance problem while the program was executing in a bparticular call path at a particular location. After the analysis has been finished, the mapping is written to a file and can be viewed using the CUBE display tool, shown in Figure 3.

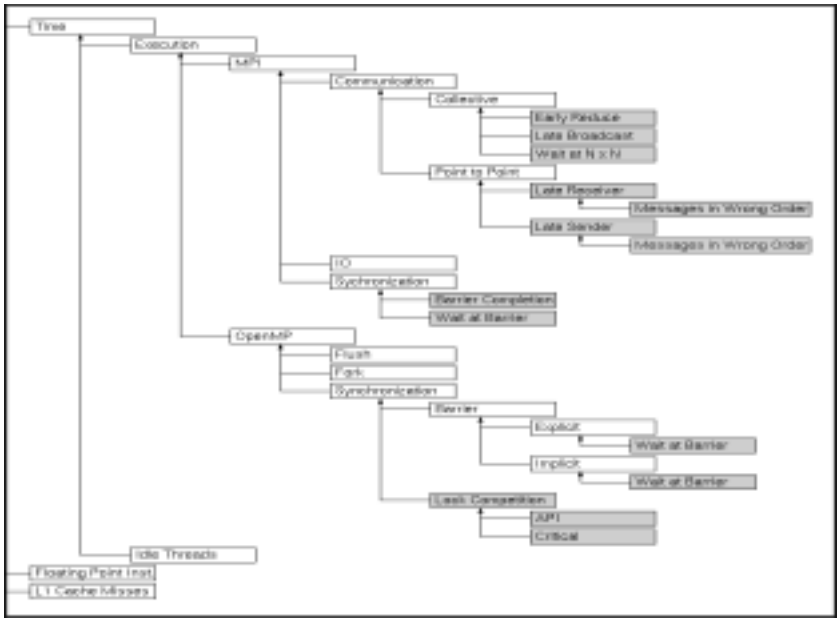


Figure 2. KOJAK pattern specialization hierarchy

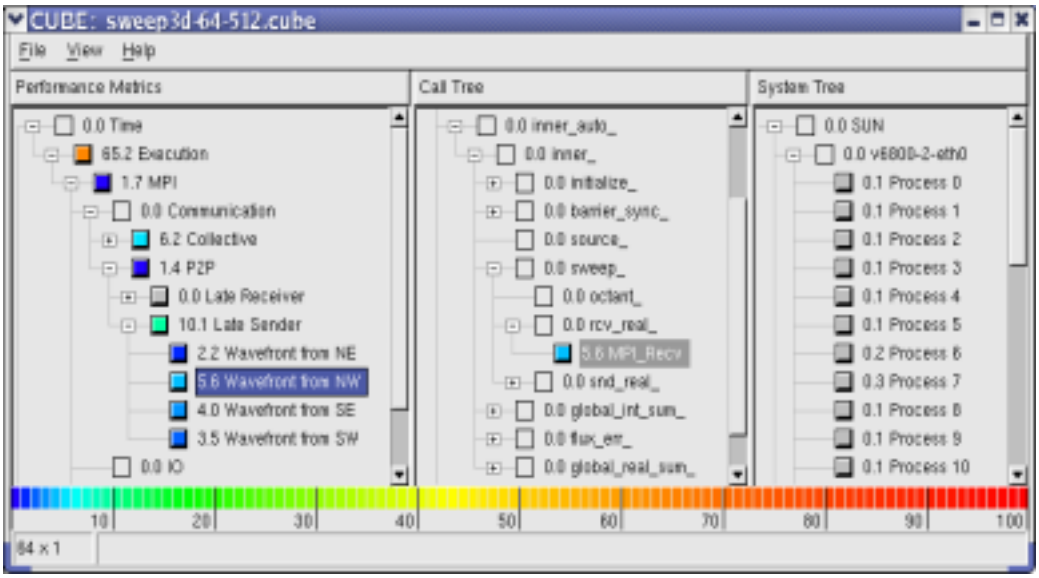
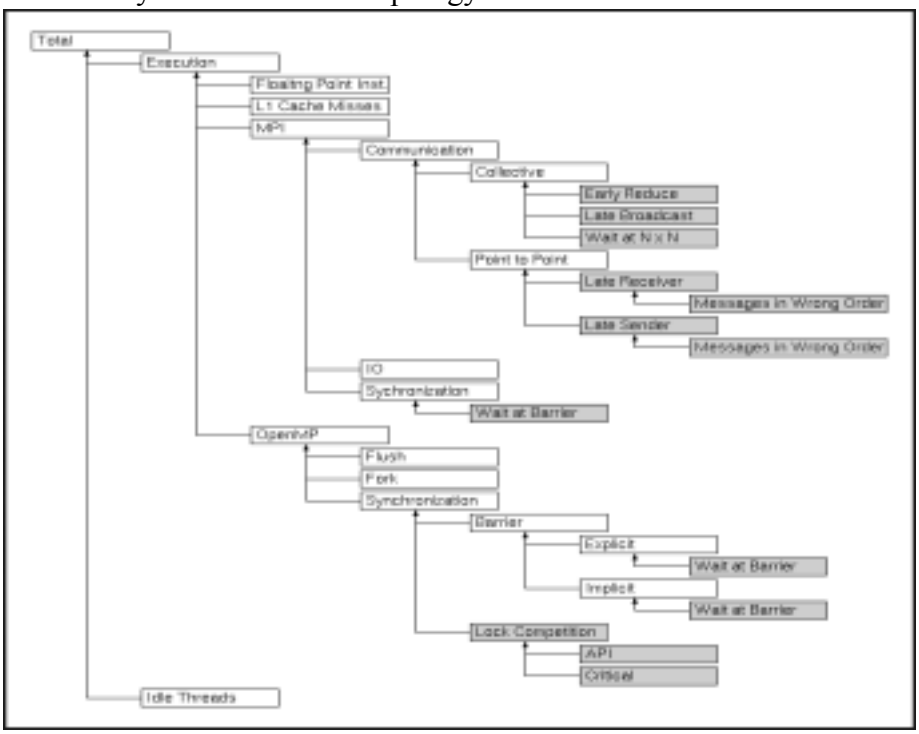


Figure 3. CUBE coupled tree browser

In previous work [8], we have demonstrated that searching event traces of parallel applications for patterns of inefficient behavior is a successful method of automatically generating high-level feedback on an application's performance. This was accomplished by identifying 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. In the following example, we show that enriching the information contained in event traces with topological knowledge allows the occurrence of certain patterns to be explained in the context of the parallelization

strategy applied and, thus, significantly raises the abstraction level of the feedback returned. In particular, we demonstrate that topological information allows the following:

- Detecting higher-level events related to the parallel algorithm, such as the change of the propagation direction in a wavefront scheme.
- Linking the occurrence of patterns that represent undesired wait states to such algorithmic higher-level events and, thus, distinguishing wait states by the circumstances causing them.
- Exposing the correlation of wait states identified by our pattern analysis with the topological characteristics of affected processes by visually mapping their severity onto the virtual topology.



For this purpose, we have developed extensions of KOKAK that provide means of recording topological information as part of the event trace and of visualizing the severity of the analyzed behaviors mapped on to the topology. Moreover, we have enhanced the analysis by specifying additional patterns that exploit topological information to find performance problems related to wavefront algorithms.

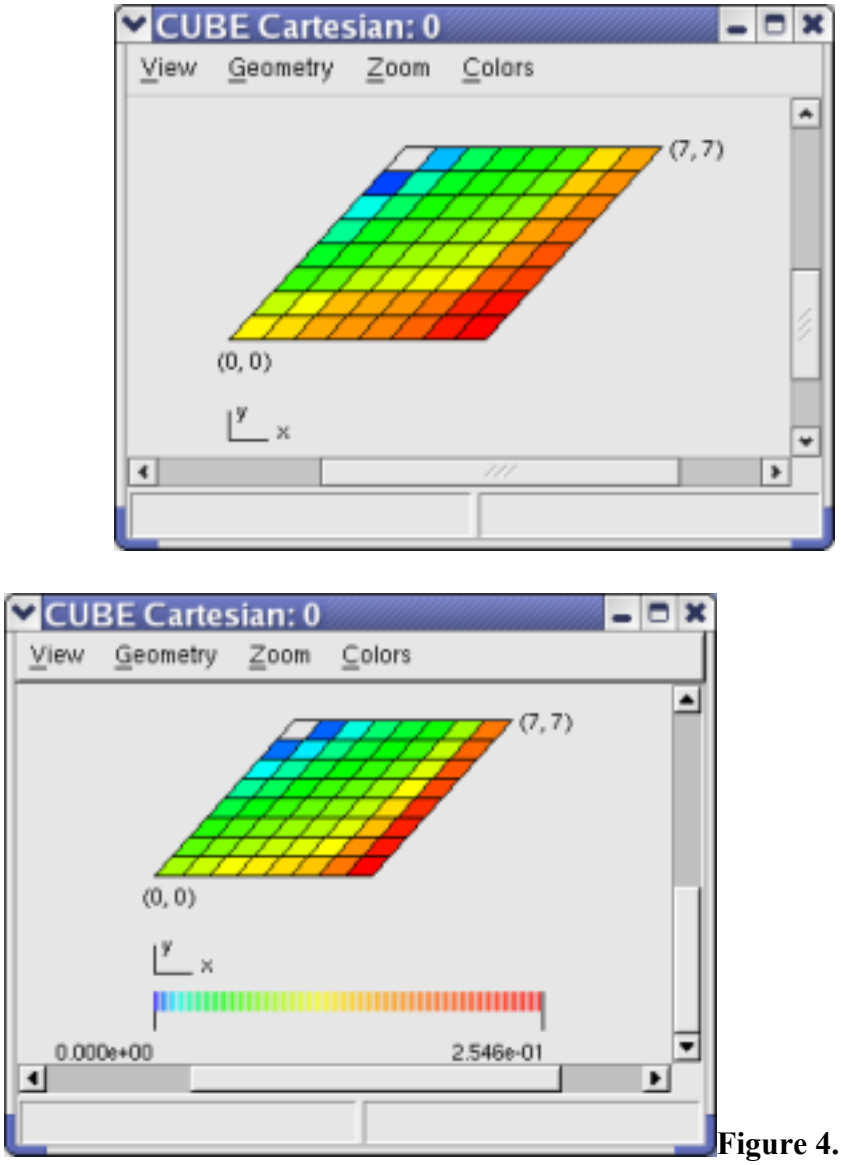


Figure 4. CUBE topology display

A topology view, as depicted in Figure 4, has been added to the CUBE tree view of processes and threads (Figure 3, right pane). The topological view can be accessed through a menu and shows the distribution of the time lost due to the selected pattern while the program was executing in the selected call path. The view is automatically updated as soon as the user selects another pattern or another call path. In this fashion, the user can study the distribution of a large variety of patterns across virtual topologies. The topology view can display one-, two-, and three-dimensional Cartesian topologies.

To illustrate how the virtual topology can be used to classify certain wait states, we applied our tool extension to the DOE ASCI SWEEP3D benchmark MPI code [9]. The example shows (i) that topological knowledge can be used to identify higher-level events related to distinct phases of the parallelization scheme used in an application and (ii) how these events influence the severity of certain inefficiency patterns. The SWEEP3D benchmark code 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.

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. As can be seen from the code structure inside routine sweep() (you might want to put the pseudo code into the document), the receive calls 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 EXPERT's late-sender pattern.

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 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, each wavefront direction has a horizontal as well as a vertical component, involving messages in two different orthogonal directions, each of them corresponding to one of the two receive and send statements in routine sweep(). We therefore need to consider two potential wait states at the moment of a direction change, each resulting from one of the two receive statements. Figure 4 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.

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.

Our second example illustrates a pattern-based search for nested loop structures with load/store bound inner loops. Although optimizing compilers can unroll inner loops, nested loop structures with load/store bound inner loops may require outer loop unrolling which is often not done by the compiler. The classic example of this problem is matrix multiplication, where hand unrolling is required to achieve the best performance.

To be able to detect load/store bound loops, we defined a new pattern named LSBoundInnerLoop that uses hardware counter data recorded in the trace file to compute the ratio of floating point operations to load/store operations for instrumented nested loop

structures. If this ratio falls below a certain threshold, for example two on the IBM POWER4, then the loop is load/store bound. Figure 5 illustrates the result of our analysis on a blocked but not unrolled version of matrix multiply. Although this is a very simple example, automatically instrumenting the loop structures of more complex applications and applying this pattern search could point to similar situations where hand unrolling of complex loops structures may be required.

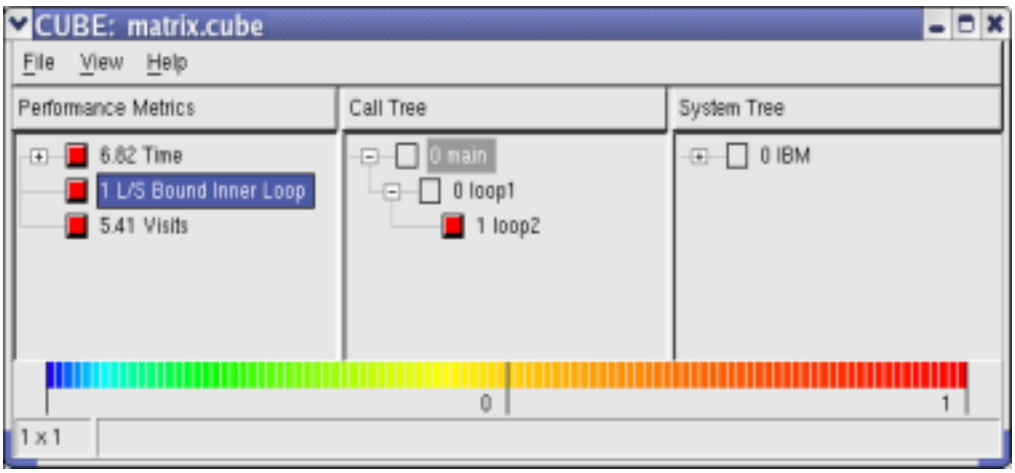


Figure 5. CUBE display of load/store bound loop in matrix multiplication

For future work, we plan to extend the KOJAK pattern matching approach in several directions. Similar to our approach for wavefront algorithms, we plan to develop more patterns specific to particular algorithmic classes, such as mixed finite element methods. We plan to develop more patterns based on hardware counter data, for example on events related to use of the cache and memory hierarchy. We plan to develop a pattern prototyping tool that will allow application developers to define their own patterns, or to specialize existing EXPERT patterns, using semantic knowledge of their application, and have these new patterns incorporated into EXPERT's search. Furthermore, we are working on extending KOJAK to analyze and correlate performance data from multiple experiments and from multiple sources.

References

1. Intel Cluster Tools web site, http://www.intel.com/software/products/cluster/index.htm

2. Jumpshot web siute, http://www-unix.mcs.anl.gov/perfvis/software/viewers/index.htm

3. Paraver web site, http://www.cepba.upc.es/paraver/

4. KOJAK web site, http://icl.cs.utk.edu/kojak/
5. PAPI web site, http://icl.cs.utk.edu/papi/

6. Browne, S., et al., A Portable Programming Interface for Performance Evaluation on Modern Processors. *International Journal of High-Performance Computing Applications* 14(3), 2000, pp. 189-204.

7. Wolf, F., et al. Efficient Pattern Search in Large Traces through Successive Refinement, in *European Conference on Parallel Computing (Euro-Par)*. Pisa, Italy. August-September, 2004.

8. Wolf, F. and B. Mohr, Automatic performance analysis of hybrid MPI/OpenMP applications. Journal of Systems Architecture 49(10-11), 2003, pp. 421-439.

9. The ASCI SWEEP3D Benchmark Code, http://www.llnl.gov/asci_benchmarks/