Automatic Experimental Analysis of Communication Patterns in Virtual Topologies*

Nikhil Bhatia [1], Fengguang Song [1], Felix Wolf [1], Jack Dongarra [1], Bernd Mohr [2], Shirley Moore [1]

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

Abstract

Automatic pattern search in event traces is a powerful method to identify performance problems in parallel applications. We demonstrate that knowledge about the virtual topology, which defines logical adjacency relationships between processes, can be exploited to explain the occurrence of inefficiency patterns in terms of the parallelization strategy used in an application. We show correlations between higher-level events related to a parallel wavefront scheme and wait states identified by our pattern analysis. In addition, we visually expose relationships between pattern occurrences and the topological characteristics of the affected processes.

Keywords: performance tools, event tracing, virtual topologies, visualization, wavefront algorithms

1 Introduction

Parallel applications often fail to exploit the full power of the underlying computing hardware. Their optimization, however, is extremely difficult due to the inherent complexity of parallel systems and their communication structures.

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. Often, the virtual topology also influences the order in which certain computations are performed. For example, wavefront algorithms [7] propagate data along the diagonals of a multi-dimensional grid of processes.

The MPI standard [9] offers a set of API functions to allow for an efficient mapping of virtual topologies onto the physical topology of the underlying machine so that communication speeds between neighbors can be optimized. Beyond that, however, topological knowledge can help identify performance problems more effectively, especially as many parallel algorithms are parametrized in terms of a virtual topology.

In our previous work [14], we 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 this article, 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:

1. Detecting higher-level events related to the parallel algorithm, such as the change of the propagation direction in a wavefront scheme.

*This work was supported by the U.S. Department of Energy under Grants DoE DE-FG02-01ER25510 and DoE DE-FC02-01ER25490

2. 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.

3. 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 an easy-to-use extension of the KOJAK toolkit [14]. KOJAK is a post-mortem trace analysis tool that enables application developers to search event traces for the possible occurrence of a large number of execution patterns indicating inefficient behavior. The extension provides a means to record topological information as part of the event trace and to visualize the severity of the analyzed behaviors mapped onto the topology. Moreover, we have enhanced the analysis by specifying additional patterns that exploit topological information to find performance problems related to wavefront algorithms.

The remainder of this article is organized as follows. In Section 2 we give a brief overview of the KOJAK toolkit and its underlying approach of analyzing patterns in event traces. After introducing the extension to record and analyze virtual topologies in Section 3, we demonstrate its usefulness using two practical examples in Section 4. Finally, we consider related work in Section 5 and present our conclusion plus future work in Section 6.

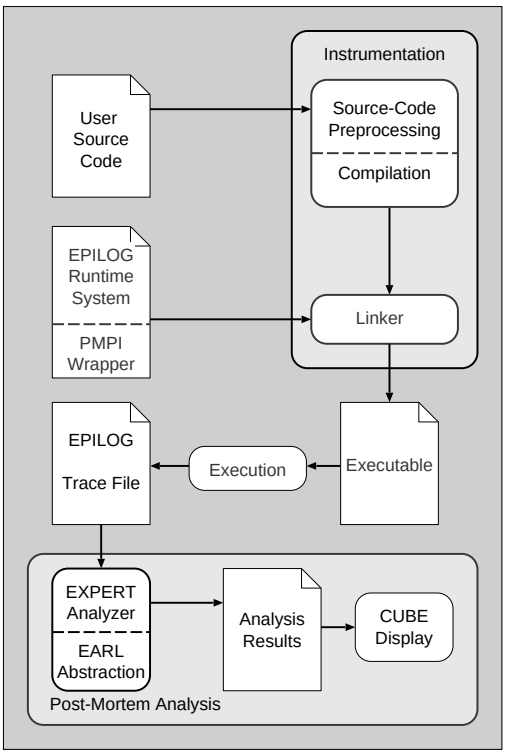

Figure 1. Overall architecture of the KOJAK system.

2 Pattern Analysis in Event Traces

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 offline with the help of software tools. As event traces preserve the temporal and spatial relationships of individual events, they allow a deeper understanding of interprocess communication and an easier identification of wait states associated with it [13]. Since event traces tend to be very large, the coverage of a purely manual analysis is often limited.

KOJAK is an automatic performance evaluation system for parallel applications that relieves the user from the burden of searching large amounts of trace data manually by automatically looking for inefficient communication patterns that force processes into undesired wait states. KOJAK can be used for MPI, OpenMP, and hybrid applications written in C/C++ or Fortran. It includes tools for instrumentation, event-trace generation, and post-processing of event traces plus a generic browser to display the analysis results.

Figure 1 shows the entire process of analyzing an application using KOJAK. Prior to trace generation, the application needs to be instrumented. Depending on the platform, this is done automatically using a combination of source-code preprocessing and compiler-based instrumentation. As a final step, the application is linked with the EPILOG runtime system, which includes a PMPI interposition library that intercepts MPI calls to perform measurements before and after each call. Finally, when the instrumented application is executed, it generates a trace file.

The trace file is written in the EPILOG format [4], which provides event types covering MPI point-to-point and collective communication as well as OpenMP parallelism change, parallel constructs, and synchronization. Also, the trace file may include data from hardware counters.

After program termination, the trace file is analyzed offline using EXPERT [15], which identifies execution patterns indicating low performance and quantifies them according to their severity. These patterns target problems resulting from inefficient communication and synchronization as well as from low CPU and memory performance. The analysis process automatically transforms the traces into a compact call-path profile that includes the time spent in different patterns.

To simplify the analysis, EXPERT accesses the trace

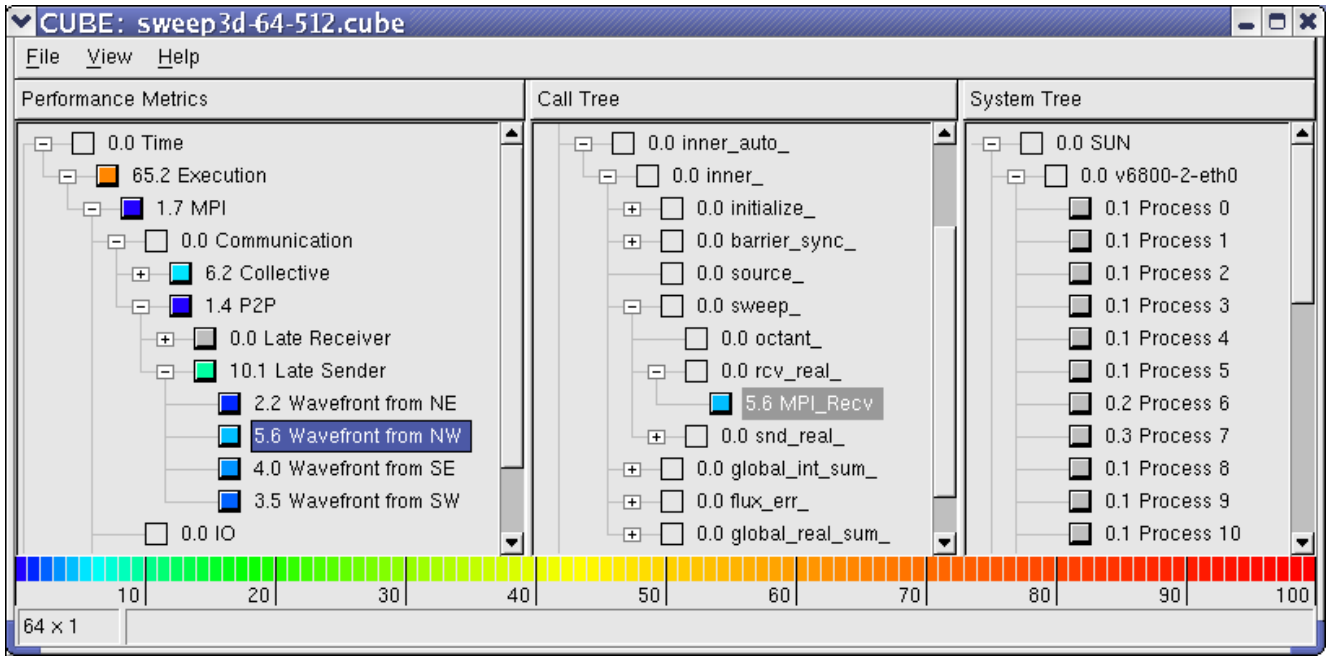

Figure 2. Visualization of performance problems using CUBE.

through the EARL library interface, which provides random access to individual events and precalculated abstractions supporting the search process. EARL is well documented and can be used for a large variety of analysis tasks beyond the analyses performed by EXPERT. The major benefits of using EARL as an intermediate layer between the analysis and the event trace are reduced size and increased readability of the pattern specifications. In EXPERT, patterns are specified separately from the actual analysis process as C++/Python classes [1] This design simplifies a later extension of the predefined pattern base, a feature we exploited here to integrate additional patterns suitable for studying wavefront algorithms.

Finally, the analysis results can be viewed in the CUBE performance browser [11], which is depicted in Figure 2. CUBE shows the distribution of performance problems across the call tree and the parallel system using tree browsers that can be collapsed and expanded to meet the desired level of granularity.

Our analysis will concentrate on a frequently occurring pattern called *late sender*. A process calls a blocking receive operation long before the message was sent and enters a wait state until the message arrives. The situation is depicted in Figure 3. Although this pattern involves two processes, the wait state can be associated with exactly one process, which is process A in this case.

EXPERT recognizes the late-sender situation by maintaining a message history to match the send and receive events and by tracking the call tree to associate the respective MPI calls with a call path. While searching the trace file, EXPERT maintains a matrix in which it accumulates the idle times incurred for a particular (call path, process) pair. After completion, the results are stored in an XML file that can be loaded into CUBE for visualization. For example, Figure 2 (left pane) shows that 5.6 % of the execution time was spent in a wait state caused by a special version of late sender called *wavefront from NW*, which is discussed in Section 4.1. The middle tree exposes the affected call path.

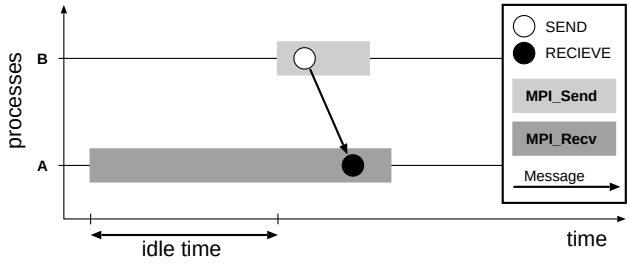

Figure 3. A process waiting for a message that was sent too late.

[1] In addition to a C++ version of the analyzer, we also maintain a Python version for prototyping purposes.

3 Virtual Topologies

To make the analysis topology-aware, we extended the following parts of the KOJAK system:

1. Trace format
2. Runtime system
3. Abstraction layer
4. Analyzer
5. Display

To keep the extension simple, we restricted ourselves to Cartesian topologies as a common case found in many of today's parallel applications. Other topology types, such as general graph topologies, might be included in the future. A Cartesian topology is essentially a multidimensional grid structure characterized by the following parameters:

- Number of dimensions
- Size of each dimension
- Periodicity of each dimension

The periodicity specifies whether the ends of a certain dimension are connected. The periodicity attribute is needed, for example, to specify a torus, a topology often found in physical networks used for point-to-point messaging in systems, such as IBM BlueGene/L. The periodicity attribute, however, is currently not used for our analysis.

Trace format. We added two record types that can be used to specify Cartesian topologies. One record type to define the general layout of a Cartesian topology and another one to map a process or thread onto a particular position within a previously defined topology. Note that the semantics of the topology can be arbitrary and that these records can be used to describe either virtual or physical topologies.

The record type used to define a topology includes fields to specify the number of dimensions, the size in each dimension, and whether a dimension is periodic or not. The record type also contains a field to specify an MPI communicator if the topology was created using the MPI_CART_CREATE function. Using this information, it is possible to filter communication operations by the communicator representing the topology.

The record type used to map a process or thread onto a position within a topology simply specifies a topology identifier and the coordinates of the process or thread within this topology. The mapping does not need to be surjective, that is, not all positions within the topology need to be filled. For example, the topology might represent the physical topology of the machine the application is running on, but without occupying all CPUs.

Runtime system. The runtime system has been extended to support the two new record types. There are two ways of defining a Cartesian topology and writing the corresponding records.

1. Automatically using MPI wrapper
2. Manually using a C/Fortran API

If the application uses MPI_CART_CREATE, the respective topology is automatically recorded as part of the trace file. This feature has been implemented by letting a PMPI wrapper, which is part of the runtime system, intercept the topology attributes and write the topology definition record. After processing the topology outline, the wrapper requests the coordinates of the calling process from the MPI runtime system and writes a corresponding coordinate-definition record.

Unfortunately, MPI topology support is rarely used. For this reason, EPILOG provides a C and Fortran API to perform exactly the steps that would otherwise be the MPI wrapper's responsibility. The API consists of two functions and allows the definition of an up to three-dimensional Cartesian topology. Using the API is fairly simple and requires only a minimal effort.

The following example defines a three-dimensional $4 \times 4 \times 4$ topology that is periodic in the first but not in the remaining two dimensions.

```
if (rank .eq. 0) then
  call elgf_cart_create(4,4,4,1,0,0)
endif
call elgf_cart_coords(x,y,z)
```

Every process executing these lines assigns itself coordinates defined through the variables `x`, `y`, `z`, containing values between 0 and 3.

Abstraction layer. The abstraction layer represented by EARL is a high-level interface for reading an event trace. Topology information can now be accessed through a class interface either in C++ or Python and used for a large variety of trace analysis tasks.

Analyzer. The analyzer has been enabled to read the topological information and to pass it on to the visualization component, which performs a general mapping of analysis results onto individual elements of the topology. In addition to this basic capability, four patterns specific to wavefront algorithms have been added to the analyzer. The patterns use topological knowledge to determine the direction of messages and to relate inefficient behavior to certain phases of the wavefront computation. Wavefront algorithms are an important class of algorithms commonly used to solve deterministic particle transport problems. The new patterns

along with their implementations are discussed along with an application example in Section 4.1.

Display. A topology view, as depicted in Figures 5 and 6, has been added to the original tree view of processes and threads (Figure 2, 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. Three-dimensional topologies are presented in parallel projection as a collection of grid-like planes arranged on top of each other. To make the display scalable, the user can adjust the size of individual grid cells, the distance between neighboring planes, and the angle used to generate a three-dimensional perspective.

The color assigned to a certain grid cell represents the time spent in a certain pattern. To make differences visible even across a large number of cells, the display is able to utilize the full spectrum of available colors for a single pattern by switching to a high-resolution color mode.

4 Examples

To study how the the virtual topology can be used to classify certain wait states, we applied our tool extension to two example MPI codes, the ASCI SWEEP3D benchmark [1] and an environmental science application called TRACE [5].

4.1 SWEEP3D

The first 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 benchmark code SWEEP3D 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 4 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.

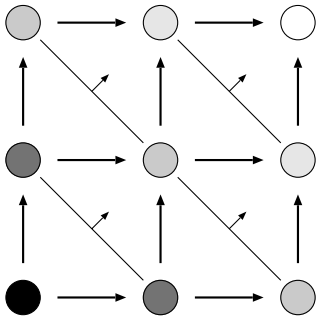

Figure 4. Wavefront propagation of data in SWEEP3D.

Responsible for the wavefront computation in the code is a subroutine called sweep(), which 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. Thus, the parallelization in SWEEP3D is based on concurrency among algorithmically independent processes and pipelining among algorithmically dependent processes. The basic code structure of routine sweep() is as follows:

```
DO octants
  DO angles in octant
    DO k planes
      ! block i-inflows
      IF neighbor(E/W) MPI_RECV(E/W)
      ! block j-inflows
      IF neighbor(N/S) MPI_RECV(N/S)
         ... compute grid cell ...
      ! block i-outflows
      IF neighbor(E/W) MPI_SEND(E/W)
      ! block j-outflows
      IF neighbor(N/S) MPI_SEND(N/S)
    END DO k planes
  END DO angles in octant
END DO octants
```

Performance models of wavefront processes, in particular as they appear in SWEEP3D, have been extensively stud-

ied [7, 12]. In this article, we analyze the characteristics of wavefront communication from an experimental viewpoint with emphasis on wait states resulting from the data dependencies illustrated in Figure 4.

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(), 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 the late-sender pattern illustrated in Figure 3.

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). Since these patterns constitute a specialization of late sender, which was already member of the pattern base, their specifications could take advantage of EXPERT's publish/subscribe mechanism [15] through registration for late-sender instances published by the simple late-sender pattern. In this way, only the changes of the pipeline direction needed to be specified, reducing the amount of code necessary to describe the combined situation. As the problem is highly symmetric, we specified the direction change in a parametrized fashion, further decreasing the lines of code needed.

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, as depicted in Figure 4, 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. Note that the horizontal component is always executed first, limiting the number of cases that need to be considered in order to detect a change.

However, special attention has to be paid to processes located at the border of the grid (Figure 4). 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. For this reason, our implementation distinguishes between different border areas to which it applies different detection rules. These rules are parameterized in terms of the wavefront origin and the area's horizontal/vertical orientation.

Note that we do not make any assumption about the order in which the different pipeline directions are scheduled, making the detection algorithm more general. To also cover the very first pipeline start, the queue is initially filled with pseudo direction symbols.

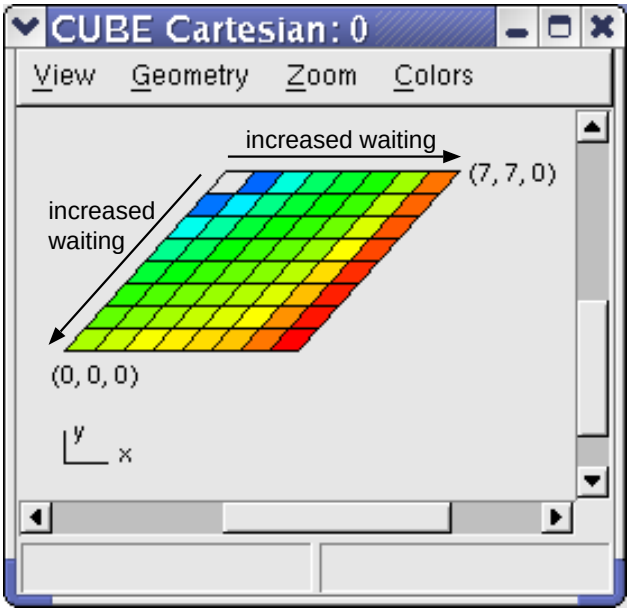

Figure 5. Distribution of late-sender wait states as a result of pipeline refill from North-West.

To validate our design, we chose a problem size $512 \times 512 \times 150$ grid points and ran the application with 64 processes on a Solaris Cluster equipped with UltraSPARC-III 750 MHz processors. The instrumentation of user functions was done fully automatically using the platform compiler's profiling interface. As the program does not take advantage of MPI topology support, we recorded the topology by manually inserting the EPILOG API calls described in Section 3 into the module responsible for the domain decomposition. Figure 2 shows the output of our analysis as rendered by the original KOJAK GUI. The new patterns appear in the metric tree on the left underneath *Late Sender* and are labeled with the percentage of execution time spent in wait states caused by the pattern. The total time spent in late-sender wait states, which can be obtained by collapsing the late-sender node, 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 5 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.

4.2 TRACE

The second example highlights how visually mapping the results of our pattern analysis onto the virtual topology can help the user identify semantically meaningful clusters of related behavior.

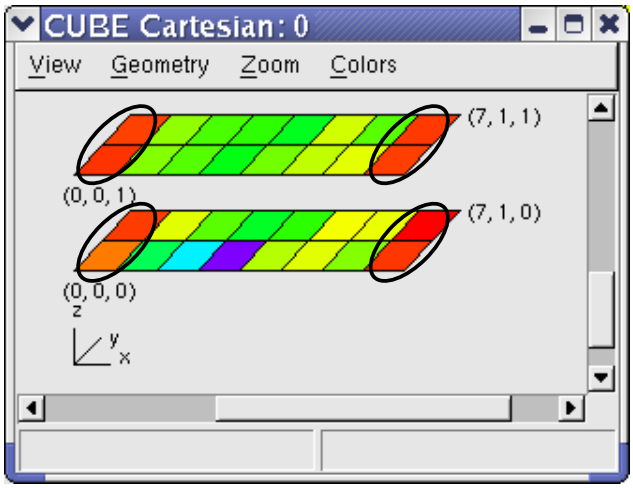

Figure 6. Distribution of wait states caused by inherently synchronizing all-to-all operations.

TRACE simulates the subsurface water flow in variably saturated porous media. It solves the generalized Richards equation in three spatial dimensions. The parallelization is based on a parallelized CG algorithm, which divides the grid into overlapping subgrids and communicates via MPI. The main computation is done in a subroutine called parallelcg(). We executed the application with 32 processes on a Linux cluster with 8 Pentium III Xeon (550 MHz) 4-way nodes. The resulting topology is a three-dimensional Cartesian $8 \times 2 \times 2$ grid (Figure 6).

The display shows the distribution of wait states in parallelcg() caused by inherently synchronizing all-to-all operations that occur when some processes enter the operation earlier than others. The pattern describing this situation is among the standard patterns included in the EXPERT analyzer.

The figure exhibits clusters of increased waiting times at the corners of the three-dimensional grid that due to their exposed location are assumed to have different computation as well as communication requirements. Without topological knowledge the affected processes would appear as arbitrary processes and the user would be unaware of the correlation between their particular role in the topology and the occurrence of specific inefficiencies.

5 Related Work

Topological information 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 four-dimensional hypercubes as well as up to three-dimensional grids and tori using a polygon-like vector representation and mapped certain communication parameters, such as the number of messages, onto their nodes to better observe communication activities in virtual shared memory systems [10].

The three-dimensional topological display developed as part of this work follows in its design the torus view included in TREND [6], a tool for supervising system utilization on the CRAY T3E. However, to accommodate a larger variety of grid sizes ranging from very small to very large, we made additional display parameters adjustable, such as the distance between planes or the angle used to create the three-dimensional perspective.

Topological knowledge has also 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 [8]. 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, a mechanism that could render the manual recording exercised in our examples unnecessary.

6 Conclusion and Future Work

Topological knowledge can be used to significantly raise the semantic level of the feedback given by KOJAK's method of scanning event traces for patterns of inefficient behavior.

Using wavefront algorithms as an example, we demonstrated that topological information enables the identification of higher-level events related to a programs parallelization scheme and the correlation of these higher-level events with wait states identified by our previous pattern analysis method. This correlation allowed us to reintroduce a time-dimension into an otherwise timeless data model of analysis results by letting pattern specifications refer to distinct algorithm-specific execution phases. We further showed that visually mapping wait states identified by our pattern analysis onto the topology enables the correlation of these wait states with topological characteristics of the affected processes. To make these capabilities generally available, they have been implemented as an easy-to-use extension of the KOJAK toolkit.

Future work will address the operations of wavefront processes in more detail by studying the overlap between pipelines coming from different directions. We also intend to extend the scope of the underlying principles to other algorithms, such as parallel multi-frontal methods [3]. To further enhance the automatic capabilities of our tool, we will investigate ways to recognize the algorithm in advance so that appropriate analysis patterns can be selected. Options to be considered include adding metadata to the event trace - for example, by instrumenting parallel libraries that are usually aware of the algorithms applied. Another approach would be the automatic detection in the trace itself similar to the automatic topology detection scheme described in [8].

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] Iain S. Duff. Parallel implementation of multifrontal schemes. *Parallel Computing*, 3:193–204, 1986.

[4] B. Mohr F. Wolf. EPILOG Binary Trace-Data Format. Technical Report FZJ-ZAM-IB-2004-06, Forschungszentrum Jülich, May 2004.

[5] Forschungszentrum Jülich. *Solute Transport in Heterogeneous Soil-Aquifer Systems*. `http://www.kfa-juelich.de/icg/icg4/Groups/Pollutgeosys/trace_e.html`.

[6] Forschungszentrum Jülich. *TREND - Torus REsources and Node Display*. `http://www.fz-juelich.de/zam/trend/`.

[7] 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.

[8] 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.

[9] Message Passing Interface Forum. *MPI: A Message Passing Interface Standard*, June 1995. `http://www.mpi-forum.org`.

[10] C. Müllender. Visualisierung der Speicheraktivitäten von parallelen Programmen in Systemen mit virtuell gemeinsamen Speicher. Master's thesis, RWTH Aachen, Forschungszentrum Jülich, May 1994.

[11] F. Song, F. Wolf, N. Bhatia, J. Dongarra, and S. Moore. An Algebra for Cross-Experiment Performance Analysis. In *Proc. of the International Conference on Parallel Processing (ICPP)*, Montreal, Canada, August 2004.

[12] 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.

[13] F. Wolf and B. Mohr. Specifying Performance Properties of Parallel Applications Using Compound Events. *Parallel and Distributed Computing Practices*, 4(3):301–317, September 2001. Special Issue on Monitoring Systems and Tool Interoperability.

[14] 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".

[15] 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.