Analyzing Parallel Programs Using Performance-Property Traces

Sebastian Flott[1], Felix Voigtländer[2], Bernd Mohr[1], and Felix Wolf[1,2]

[1] Jülich Supercomputing Centre
Institute for Advanced Simulation
Forschungszentrum Jülich
52425 Jülich, Germany
{s.flott,b.mohr,f.wolf}@fz-juelich.de

[2] Department of Computer Science
RWTH Aachen University
52056 Aachen, Germany

Abstract. The performance of parallel applications is often affected by wait states occurring when processes fail to reach synchronization points simultaneously. In the KOJAK project, we have shown that these wait states and other performance properties can be diagnosed by searching event traces for characteristic patterns and quantifying their severity, i.e., their influence on the program execution. However, our current search method only summarizes the calculated severities to inform about the overall performance penalty. Temporal and spatial relationships between individual pattern instances are lost, although these relationships can be essential to understand the detailed circumstances of a performance problem. In this paper, we describe how these relationships can be retained by writing a second event trace with events delimiting individual pattern occurrences. Guided by our summary report, these synthetic pattern traces can be interactively analyzed using standard trace browsers, taking advantage of their powerful time-line visualizations and rich statistical functionality.

1 Introduction and Motivation

A significant fraction of the time parallel applications spend in communication and synchronization routines can often be attributed to wait states that occur when processes fail to reach implicit or explicit synchronization points in a timely manner, for example, as a result of an unevenly distributed workload. Especially when trying to scale communication-intensive applications to large processor counts, such wait states can present severe challenges to achieving good performance. As a first step in reducing the impact of wait states, application developers need a diagnostic method that allows their localization, classification, and quantification.

In the KOJAK [1, 2] project, we have shown that wait states and related performance properties can be diagnosed by searching event traces for characteristic

patterns and quantifying their severity, i.e., their influence on the program execution. While the application is running, KOJAK records when user routines, MPI functions, and OpenMP constructs are entered or left and when messages are sent or received. The resulting events are written to a trace file, which is subsequently searched by KOJAK's trace analyzer for predefined *performance properties*. A performance property is defined by a event pattern which indicates a code behavior influencing the performance of the application. The search results are compiled into an analysis report, which can be displayed using the CUBE presenter [5]. For every performance property, the analysis report includes the associated severity broken down by call path and process or thread. The severity is expressed as the corresponding time penalty accumulated across the entire execution. As an alternative, the event traces can be converted and explored using time-line visualization tools such as PARAVER [4] or VAMPIR [3] to manually search for performance bottlenecks.

However, none of the two approaches are perfect: Although the automatic pattern search produces a compact output that quickly pinpoints the most severe performance problems and shows which code paths and system resources are affected, it ignores the temporal and spatial relationships between individual pattern instances by summarizing the performance behavior across the entire execution. In contrast, the visual time-line representation of trace data preserves all these relationships, although the analysis process is usually more time-consuming and poses the danger of overlooking important details in the huge amount of data.

In this paper, we describe a synthesis of the two methods: In addition to summarizing the penalties caused by all patterns detected in the trace, we let KOJAK write a second event trace with events describing individual pattern instances. Whenever a pattern instance is found, events describing the pattern (i.e., enter and exit events of regions instances involved) and events describing the extent of the severity (i.e., enter and exit events marking the begin and end of the phase during which time is wasted) are generated and written to the new trace. Guided by the summary report, the new performance-property trace can then be interactively analyzed alongside the original trace using standard trace browsers, taking advantage of their powerful time-line visualizations and rich statistical functionality.

While the basic idea sounds simple, its implementation is complicated by the following constraints: (i) the generated events have to be written to the final trace file in chronological order and (ii) the enter and exit events describing the begin and end of a region have to form a valid parenthesis expression (i.e., individual region instances must be properly nested). A naive approach that first generates all pattern events in main memory and sorts them before writing them to disk in one chunk would consume a prohibitively large amount of memory. We therefore resort to a sliding window approach and employ a sophisticated buffering scheme to ensure that the events belonging to overlapping pattern instances are written in the correct order. To keep the size of the output trace small enough that the trace browsers can handle it, the user can specify for which specific

patterns a performance-property trace should be generated and optionally indicate a minimum severity threshold for individual pattern instances to exclude insignificant instances from the output trace. Finally, the user can select whether only pattern events, only severity events, or both are generated. Another option is to generate the severity events only and merge the resulting trace with the original one, where they highlight problematic phases in the program behavior.

In the following, the traditional performance analysis process using KOJAK, VAMPIR, and PARAVER is presented. Then, the performance properties defined by KOJAK and the mechanism to locate them is explained. The main focus is placed on the realization of the new functionality and, in particular, on our approach to meet the challenges imposed by the ordering constraints mentioned above. Finally, the benefits of our approach are demonstrated using traces from both synthetic benchmarks and real-world applications.

2 Performance Analysis with KOJAK, PARAVER, and VAMPIR

Note: This section will be provided with the final version of the paper.

3 Generation of Performance Property Traces

Note: This section will be provided with the final version of the paper.

4 Application Example: WRF-NMM

WRF-NMM is a public-domain numerical weather prediction code developed by the U.S. National Oceanic and Atmospheric Administration (NOAA) National Centers for Environmental Prediction (NCEP), consisting of the Nonhydrostatic Mesoscale Model (NMM) within the Weather Research and Forecasting (WRF) system[6]. It consists of some 530+ source files running to over 300 thousand lines of code (75% Fortran, 25% C). Simulations were analyzed using the Eur-12km dataset with a default configuration, apart from varying the duration of the forecast and disabling intermediate checkpoints. The data shown here are from an experiment with 64 processors on the BSC's MareNostrum machine.

In Fig. 1, portions of a time line depicting the first 24 of 64 total processes executing a portion of the WRF main iteration loop is shown. The collective `MPI_Allreduce` call (left in the picture) is followed by non-blocking point-to-point communication. While the ragged appearance of the shape of the collective call indicates already an imbalance, it is totally unclear whether the point-to-point communication is effective or not. Fig. 2 shows the corresponding portion of the generated performance property trace. Now, one can clearly see, that only half of the point-to-point communication is affected by a performance problem called "Late Sender", which indicates that the receiver of a message is waiting for a message that has not been sent yet. However, the severity is only a small portion of the overall communication, leaving very little room for improvement.

Note: The final version of the paper will show another example we found, where a performance problem with a small severity triggers another problem with a much bigger influence. Fixing the small problem would remove also the big one. However, a simple automatic performance analysis would give the (wrong) impression that the problem with the high severity is more important.

5 Conclusion and Future Work

Performance property traces are a new and very powerful tool enhancing the performance analyst's toolbox. When the high-level information provided by KOJAK is not enough to locate and fix the performance problem a code may exhibit, a manual visual analysis of the original event trace combined or enriched with the event data of our new performance property trace, and guided by our summary report, can be a very effective means of answering remaining questions and addressing open issues.

In the future, we plan to parallelize the functionality described in this paper to make it more scalable (and therefore more applicable to larger problems) in the context of the SCALASCA project[7, 8]. Also, we want to investigate, whether the analysis of the performance property-trace can be automated.

References

1. F. Wolf, B. Mohr, Automatic performance analysis of hybrid MPI/OpenMP applications, Journal of Systems Architecture 49 (10-11) (2003) 421–439.
2. Wolf, F.; Mohr, B.; Dongarra, J.*; Moore, S.* Automatic analysis of inefficiency patterns in parallel applications Concurrency and Computation: Practice and Experience, 19 (2007) 11, 1481–1496
3. H. Brunst, W. E. Nagel, Scalable performance analysis of parallel systems: Concepts and experiences, in: Proc. of the Parallel Computing Conference (ParCo), Dresden, Germany, 2003.
4. J. Labarta, S. Girona, V. Pillet, T. Cortes, L. Gregoris, DiP : A parallel program development environment, in: Proc. of the 2nd International Euro-Par Conference, Springer, Lyon, France, 1996.
5. M. Geimer, B. Kuhlmann, F. Pulatova, F. Wolf, B. J. N. Wylie: Scalable Collation and Presentation of Call-Path Profile Data with CUBE. Parallel Computing : Architectures, Algorithms and Applications (Proceedings of ParCo 2007, Jülich/Aachen, Germany), 645–652, John von Neumann Institute for Computing (NIC series, 38), 2007.
6. Weather Research Forecast code. http://www.wrf-model.org/.
7. M. Geimer, F. Wolf, B. J. N. Wylie, B. Mohr, Scalable parallel trace-based performance analysis, in: Proc. 13th European PVM/MPI Users' Group Meeting, Bonn, Germany, LNCS 4192, Springer, Germany, 2006.
8. M. Geimer, F. Wolf, A. Knüpfer, B. Mohr, B. J. N. Wylie, A Parallel Trace-Data Interface for Scalable Performance Analysis Applied Parallel Computing. State of the Art in Scientific Computing: 8th International Workshop, PARA 2006, Umeå, Sweden, Revised Selected Papers. LNCS 4699, Springer, 2007.

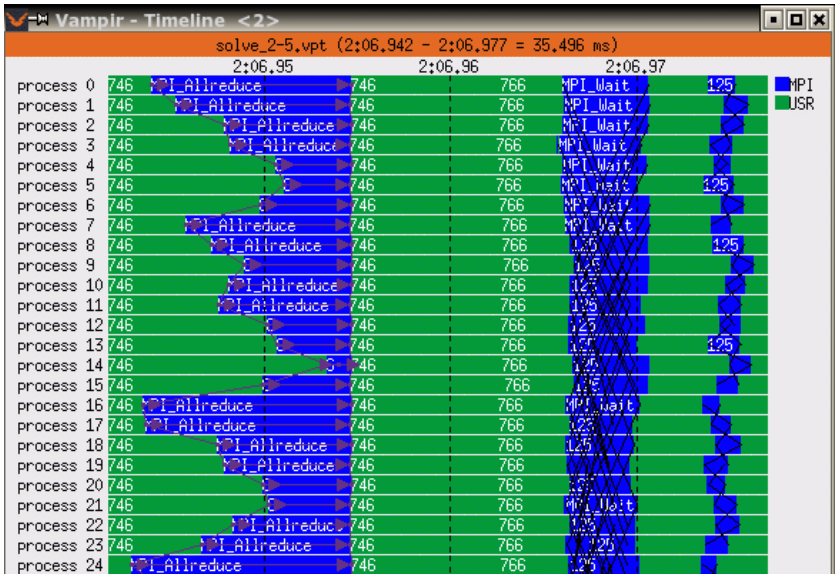

Fig. 1. Zoomed time line of the original trace showing the first 24 of 64 total processes executing a portions of the WRF main iteration loop. It shows a collective `MPI_Allreduce` call followed by non-blocking point-to-point communication.

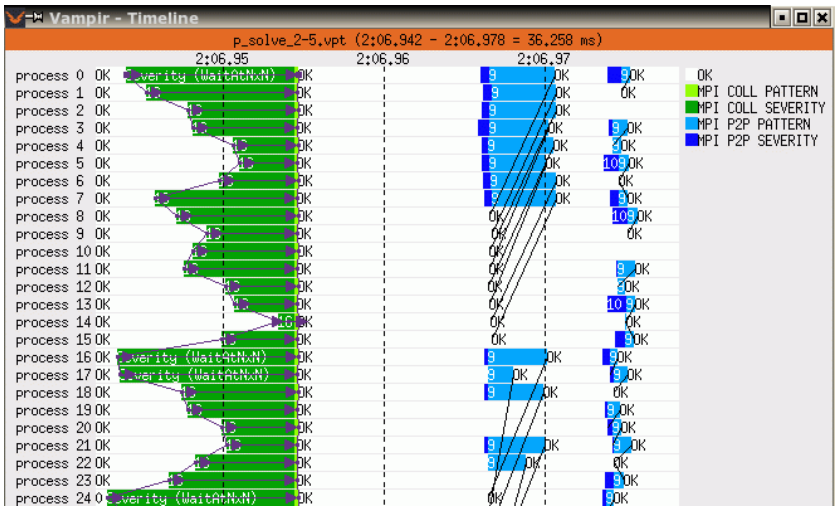

Fig. 2. Zoomed time line of the corresponding pattern trace showing the same region in the pattern trace. It clearly shows which portions of the MPI phases are synchronization (waiting) and communication (working).