Characterizing Load and Communication Imbalance in Large-Scale Parallel Applications

David Böhme, Felix Wolf
German Research School for Simulation Sciences
RWTH Aachen University, 52062 Aachen, Germany
Email: {d.boehme,f.wolf}@grs-sim.de

Markus Geimer
Jülich Supercomputing Centre
52425 Jülich, Germany
Email: m.geimer@fz-juelich.de

***Abstract*—Load or communication imbalance prevents many codes from taking advantage of the parallelism available on modern supercomputers. We present two scalable methods to highlight imbalance in parallel programs: The first method identifies delays that inflict wait states at subsequent synchronization points, and attributes their costs in terms of resource waste to the original cause. The second method combines knowledge of the critical path with traditional parallel profiles to derive a set of compact performance indicators that help answer a variety of important performance-analysis questions, such as identifying load imbalance, quantifying the impact of imbalance on runtime, and characterizing resource consumption. Both methods employ a highly scalable parallel replay of event traces, making them a suitable analysis instrument for massively parallel MPI programs with tens of thousands of processes.**

David Böhme is in his 4th year of pursuing a Ph.D. degree at RWTH Aachen University, Germany. His advisor is Prof. Dr. Felix Wolf; co-advisor is Dr. Markus Geimer.

I. Introduction

Rising numbers of cores per socket in combination with an architecture ecosystem characterized by increasing diversification and heterogeneity continue to complicate the development of efficient parallel programs. Particularly load imbalance, which frequently occurs during simulations of irregular and dynamic domains – a typical scenario in many engineering codes – presents a key challenge to achieving satisfactory parallel efficiency. Typically, load imbalances manifest themselves in the form of wait states at the next synchronization point following the imbalance, which allows a potentially large temporal distance between the cause and its symptom. Moreover, when complex point-to-point communication patterns are employed, wait states may propagate across process boundaries along far-reaching cause-effect chains that are hard to track manually and that complicate the assessment of the actual costs of an imbalance. These challenges creates a strong need for effective performance analysis methods that highlight load balancing issues that occur at larger scales. To be useful to an application developer, these methods must not only be scalable themselves but also require as little effort as possible to interpret the results.

In the course of this thesis, we significantly extended the capabilities of the Scalasca performance analysis toolset [1] with two new methods that identify load imbalance and accurately determine its performance impact in terms of runtime and resource waste. In the past, Scalasca analyzed MPI and hybrid MPI/OpenMP programs for MPI-related wait states. Directly complementing this wait-state analysis, the new *delay analysis* [2] characterizes load imbalance by its effect on wait states: starting at the end of the causation chain, it maps the costs of wait states onto the delays that originally caused them. In addition, we developed a set of compact *performance indicators* based on the detection of the critical path [3] that intuitively guide the analysis of complex load-imbalance phenomena. By replaying event-traces in parallel, Scalasca can efficiently handle even large-scale processor configurations.

Being a fundamental property of parallel performance, load imbalance has been adressed by various performance tools. Many tools, e.g. [4], [5], [6], analyze imbalance based on per-process profiles. While profile-based solutions are generally lightweight and introduce little storage and run time overhead, the aggregation of performance data over time hides dynamic performance effects, such as the shift of load imbalance over time. Trace-based solutions, such as Vampir [7], capture the entire dynamic behavior, but often present performance data in its entirety, forcing users to search for inefficiency patterns manually. Carnival [8], a prior automatic solution to identify root causes of wait states in event traces, used a serial approach which does not scale. Our scalable, automatic performance analysis approaches accurately determine the performance impact of both static and dynamic imbalances, and guide developers directly to targets whose load or communication balance optimization would have the highest benefit. Moreover, prior solutions that specifically target imbalance only function for SPMD (single-program multiple data) applications, whereas our solutions apply equally well to both SPMD and MPMD codes.

II. Identifying Root Causes Of Wait States

Wait states, which are intervals through which a process is idle while waiting for a delayed process, are a primary symptom of load imbalance in parallel programs. The delay analysis identifies the root causes of wait states and calculates the costs of delays in terms of the waiting time that they induce. Wait states can also delay subsequent communication operations and produce further indirect wait states, adding to the total costs of the original delay. However, while wait states as symptoms of delay can be easily detected, the potentially

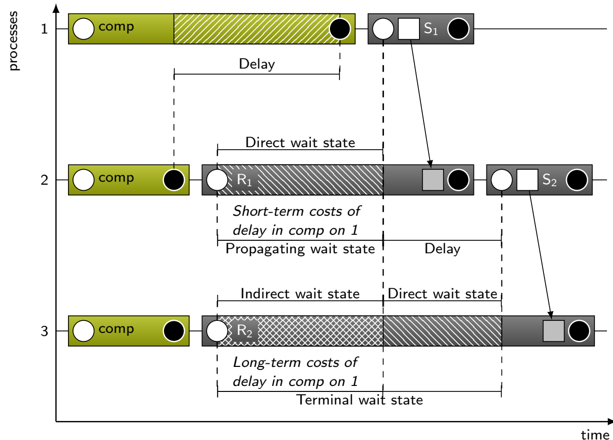
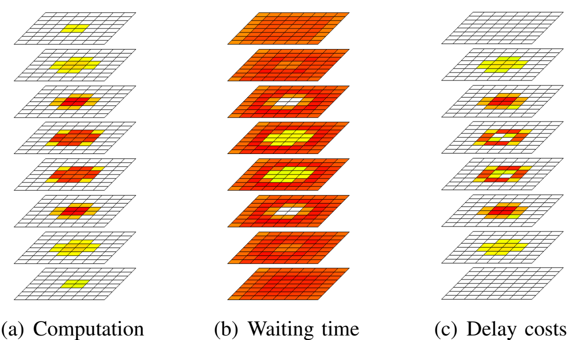


Fig. 1. Time-line diagram showing the activities of three processes and their interactions. The execution of a certain code region is displayed as a shaded rectangle and the exchange of a message as an arrow pointing in the direction of the transfer. Rank 1 delays rank 2 due to an imbalance in function comp(), inducing a wait state in the receive operation R_1 of rank 2. The wait state in R_1 subsequently delays process 3. Thus, the total costs of the delay on rank 1 correspond to the total amount of wait states caused by it directly (short-term costs) or indirectly (long-term costs).



(a) Computation (b) Waiting time (c) Delay costs

Fig. 2. Distribution of computation time, waiting time, and total delay costs in Zeus-MP/2 across the 8x8x8 three-dimensional computational domain. Red colors indicate high values.

large temporal and spatial distance in between constitutes a substantial challenge in deriving helpful conclusions from this knowledge with respect to remediating the wait states. To close this gap, the delay analysis contributes (1) a terminology to describe the formation of wait states and a cost model that allows delays to be ranked according to their associated resource waste, and (2) a scalable algorithm that identifies wait-state inducing delays and calculates their costs.

The time-line diagram in Figure 1 helps illustrate our wait-state formation model. A *wait state* is an interval during which a process sits idle. Wait states typically occur inside a communication operation when a process is waiting to synchronize with another process that has not yet reached the synchronization point. Wait states can be classified in two different ways, depending on the direction from where we start analyzing the chain of causation that leads to their formation. If we start from the cause, we can divide wait states into direct and indirect wait states. A *direct* wait state is a wait state that is caused by some "intentional" extra activity that does not include waiting time itself, whereas an *indirect* wait state is caused by a preceding wait state that propagated across the process boundary. If we look at wait-state formation starting from the effect, we can distinguish between wait states at the end and those in the middle of the causation chain. A *terminal* wait state is a wait state that does not propagate any further and is, thus, positioned at the end of the causation chain. In contrast, *propagating* wait states are those which cause further wait states later on.

A *delay* is the original source of a wait state, that is, an interval that causes a process to arrive belatedly at a synchronization point, causing one or more other processes to wait. Besides simple computational overload, delays may include a variety of behaviors such as serial operations or centralized coordination activities that are performed only by a designated process. The *costs* of a delay are the total amount of wait states it causes. Since the delay costs define a perspective from the beginning of the causation chain, we believe that the following refinement is most useful: Short-term costs cover the direct wait states, whereas long-term costs cover the indirect wait states. The total delay costs are simply the sum of the two.

The result of the delay analysis is a mapping of the costs of a delay onto the call paths and processes where the delay occurs, offering a high degree of guidance in identifying promising targets for load or communication balancing. Together with the analysis of wait-state propagation effects, the delay costs enable a precise understanding of the root causes and the formation of wait states in parallel programs. By extending Scalasca's parallel trace replay approach with an additional replay in backward direction, as described in Section IV, we can detect delays and calculate their costs in a highly scalable manner.

We applied the delay analysis to a variety of real-world MPI programs. One example is the astrophysics code Zeus-MP/2, where we studied the formation of wait states in a simulation of a 3D blast wave over 100 time steps on 512 processes. Around 12.5% of the program's total CPU allocation time is waiting time. Scalasca's report browser can visualize the Cartesian process topology of a program, which we use in Figure 2 to illustrate the relation between waiting and delaying processes in terms of their position within the computational domain. Obviously, there is a computational load imbalance between the central and outer ranks of the domain. Accordingly, the underloaded processes exhibit a significant amount of waiting time (Figure 2(b)). Our analysis shows that about 70% of the waiting time was indirectly caused by wait-state propagation. Examining the delay costs reveals that almost all the delay originates from the border processes of the central, overloaded region (Figure 2(c)). The distribution of the workload explains this observation: Within the central and outer regions, the workload is relatively well balanced. Therefore, communication within the same region

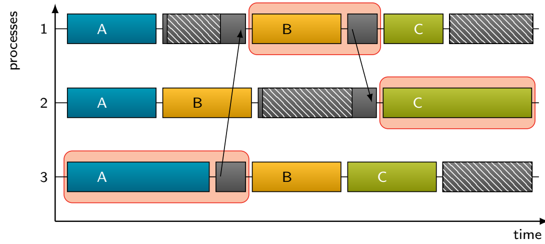


Fig. 3. Time-line diagram of a parallel program run. Each rectangle represents an activity. Arrows between processes denote communication; the hatched areas inside communication activities represent wait states. The activities highlighted in red are on the critical path.

is not significantly delayed. In contrast, the difference in computation time between the central and outer region causes wait states at synchronization points along the border, which subsequently propagate towards the outer domain border.

By pinpointing one subroutine and three computational loops with particularly high delay costs, the delay analysis also helped isolating the imbalanced source-code regions that lead to the wait states.

III. Detecting Imbalance Using the Critical Path

Our search for compact yet powerful means to uncover load imbalance in parallel programs has also led us to revisit the critical path as a key performance structure. Although the power and expressiveness of the critical path has been demonstrated in previous work (e.g. [9], [10], [11]), critical-path techniques only play a minor role in current performance analysis tools. This arises partly from the difficulty in isolating the critical path, but also from the inability to extract intuitively accessible insight from the available information. Our work addresses both issues. We leverage Scalasca's parallel trace replay technique to isolate the critical path in a highly scalable way. Also, instead of exposing the lengthy critical-path structure to the user in its entirety, we combine the critical-path with per-process profiles to derive a set of compact performance indicators, which provide intuitive guidance about load-balance characteristics and quickly draw attention to potentially inefficient code regions. The critical path provides an overview of the most time-consuming activities, but does not capture important parallel performance characteristics such as load balance by itself. Per-process profiles, on the other hand, do not capture dynamic effects that characterize a program's execution. However, a combination of critical-path and per-process profiles characterizes load balance and highlights typical parallelization issues more reliably than per-process profiles alone do.

The critical path is the longest path through a program activity graph that does not include wait states. Thus, it determines the length of program execution. Prolonging activities on the critical path increases program runtime, whereas shortening them (usually) reduces it. In contrast, optimizing an activity not on the critical path only increases waiting time, but does not affect the overall runtime. Figure 3 illustrates the concept. Our critical-path analysis produces two groups of performance indicators. The first group, the *critical-path profile* and the *critical-path imbalance indicator*, describes the impact of program activities on (wall clock) execution time. The critical-path profile represents the time an activity spends on the critical path. The critical-path imbalance corresponds to the time that is lost due to inefficient parallelization in comparison with a perfectly balanced program. As such, it provides similar guidance as prior profile-based load imbalance metrics (e.g., the difference of maximum and average aggregate workload per process), but the critical-path imbalance indicator can draw a more accurate picture. The critical path retains dynamic effects in the program execution, such as shifting of imbalance between processes over time, which per-process profiles simply cannot capture. Because of this, purely profile-based imbalance metrics regularly underestimate the actual performance impact of a given load imbalance. As an extreme example, consider a program in which a function is serialized across all processes but runs for the same amount of time on each. Purely per-process profile based metrics would not show any load imbalance at all, whereas the critical-path imbalance indicator correctly characterizes the function's serialized execution as a performance bottleneck.

The second group of indicators, the *performance impact indicators*, describes how program activities influence resource consumption. These indicators are especially useful for the analysis of MPMD programs. MPMD programs often combine multiple SPMD codes which run in separate process partitions. Achieving optimal load balance in MPMD codes typically also involves runtime configuration adjustments, such as finding optimal process partition sizes. Developers currently must use trial-and-error methods for finding suitable configurations due to the lack of proper tool support. Our performance impact indicators simplify the search for optimal configurations by distinguishing between imbalance that occurs within an SPMD process partition (*intra-partition imbalance*) or between different partitions (*inter-partition imbalance*).

We also evaluated the critical-path analysis on a variety of real-world MPI codes. For the Zeus-MP/2 example, the four code regions that the delay analysis proved being responsible for the bulk of waiting time are indeed the most imbalanced ones according to the critical-path imbalance indicator. Specifically, the imbalance indicator shows that the program execution would finish 42 seconds earlier (12% of its current run time) if these four regions were perfectly load-balanced. In our IPDPS 2012 paper [3], we also demonstrate how the performance impact indicators help finding optimal configurations for the MPMD code ddcMD.

IV. Scalable Event-Trace Replay

Both delay analysis and critical-path analysis are implemented as extensions to the automatic wait-state detection of the Scalasca performance analysis toolset, leveraging its scalable, post-mortem event-trace analysis. Figure 4 illustrates Scalasca's trace-analysis workflow. To collect event traces, the target application is instrumented so that it records relevant

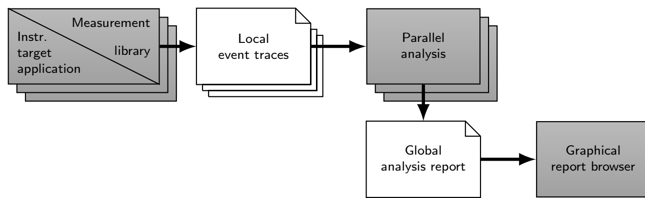


Fig. 4. Scalasca's parallel trace-analysis workflow; gray rectangles denote programs and white rectangles denote files. Stacked symbols denote multiple instances of programs or files, run or processed in parallel.

events at runtime, such as entering and leaving of source-code regions and sending or receiving of messages. After the target application finishes, we launch the trace analyzer with one analysis process per (target) process. This approach exploits the distributed memory and processing capabilities of the underlying parallel system, which is the key to achieving good scalability. The analyzer then traverses the traces in parallel, iterating over each process-local trace, and exchanges data required for the performance analysis at each recorded synchronization point using a communication operation similar to the one originally used by the program.

Other than the pure wait-state analysis, the delay and critical-path analysis require an additional, backward replay over the trace. A backward replay processes a trace backwards in time, from its end to its beginning, and reverses the role of senders and receivers. Overall, the analysis now consists of two stages: (1) a parallel forward replay that performs the wait state analysis and annotates communication events with information on synchronization points and waiting time incurred; and (2) a parallel backward replay that identifies the delays causing each of the wait states detected during forward replay, calculates their costs, and extracts the critical path. Starting at the endmost wait states, the backward replay allows delay costs to travel from the point where they materialize in the form of wait states back to the place where they are caused by delays. The backward replay also facilitates the critical-path analysis, since the route of the critical path through the program cannot be determined without knowing the end of the execution. For MPI programs, the critical path runs between MPI_Init and MPI_Finalize. Our critical-path search begins by determining the MPI rank that entered MPI_Finalize last, which marks the endpoint of the critical path, and then exploits the lack of wait states on the critical path: whenever a wait state is found on the currently active path, the search proceeds on the MPI rank that caused the wait state. This way, we follow the entire critical path backwards through the trace. After Scalasca completes the critical-path extraction, it calculates the performance indicators using a parallel algorithm.

To demonstrate the scalability of our approach, we analyzed an execution of the Sweep3D benchmark on 262,144 processes of the Blue Gene/P system at the Jülich Supercomputing Centre. The benchmark itself ran for 473 seconds. The subsequent analysis took 700 seconds in total, of which 591 were file I/O. The actual trace analysis, including wait-state search, delay analysis, and critical-path analysis, took less than 100 seconds.

V. Conclusion and Outlook

Load and communication imbalance remains a major scalability challenge for applications on their way to deployment on peta- or exa-scale systems. Imbalance typically leads to wait states later on in the execution, but intricate propagation effects may obscure the link between observable wait states and their original root causes. Therefore, the actual performance impact of an imbalance is hard to discover. Our performance indicators derived from the critical path help to identify imbalance in the execution and determine its impact on run time or resource usage, while the delay analysis determines the contribution of imbalance to the formation of wait states. Both methods provide valuable guidance in identifying promising optimization targets. In comparison to prior and related work, our parallel trace-analysis approach incorporates dynamic effects that profile-based solutions miss, thus providing more accurate results; provides deep insights at a high level of abstraction that is easily accessible; and scales to system sizes of more than 100,000 processes.

In the future, we plan to extend the delay and critical-path analysis to OpenMP and hybrid OpenMP/MPI programs, and integrate the current prototype implementations into upcoming Scalasca releases.

References

[1] M. Geimer, F. Wolf, B. J. N. Wylie, and B. Mohr, "A Scalable Tool Architecture for Diagnosing Wait States in Massively Parallel Applications," *Parallel Computing*, vol. 35, no. 7, pp. 375–388, Jul. 2009.

[2] D. Böhme, M. Geimer, F. Wolf, and L. Arnold, "Identifying the root causes of wait states in large-scale parallel applications," in *Proc. of the 39th International Conference on Parallel Processing (ICPP, San Diego, CA, USA)*. IEEE Computer Society, Sep. 2010, pp. 90–100.

[3] D. Böhme, B. R. de Supinski, M. Geimer, M. Schulz, and F. Wolf, "Scalable critical-path based performance analysis," in *Proc. of the 26th IEEE International Parallel & Distributed Processing Symposium (IPDPS 2012) (to appear)*, 2012.

[4] N. R. Tallent, L. Adhianto, and J. Mellor-Crummey, "Scalable Identification of Load Imbalance in Parallel Executions Using Call Path Profiles," in *Supercomputing 2010*, New Orleans, LA, USA, Nov. 2010.

[5] L. DeRose, B. Homer, and D. Johnson, "Detecting application load imbalance on high end massively parallel systems," in *Euro-Par 2007 Parallel Processing*, ser. Lecture Notes In Computer Science, vol. 4641. Springer, 2007, pp. 150–159.

[6] M. Calzarossa, L. Massari, and D. Tessera, "A methodology towards automatic performance analysis of parallel applications," *Parallel Computing*, vol. 30, no. 2, pp. 211–223, Feb. 2004.

[7] W. E. Nagel, A. Arnold, M. Weber, H. C. Hoppe, and K. Solchenbach, "VAMPIR: Visualization and Analysis of MPI Resources," *Supercomputer*, vol. 12, no. 1, pp. 69–80, 1996.

[8] W. Meira Jr., T. J. LeBlanc, and V. A. F. Almeida, "Using cause-effect analysis to understand the performance of distributed programs," in *SPDT '98: Proceedings of the SIGMETRICS symposium on Parallel and distributed tools*. New York, NY, USA: ACM, 1998, pp. 101–111.

[9] M. Schulz, "Extracting Critical Path Graphs from MPI Applications," in *Proceedings of the 7th IEEE International Conference on Cluster Computing*, September 2005.

[10] C. A. Alexander, D. S. Reese, J. C. Harden, and R. B. Brightwell, "Near-Critical Path Analysis: A Tool for Parallel Program Optimization," in *Proceedings of the First Southern Symposium on Computing*, 1998.

[11] J. K. Hollingsworth, "An Online Computation of Critical Path Profiling," in *Proceedings of the SIGMETRICS Symposium on Parallel and Distributed Tools (SPDT '96)*, 1996.