Understanding the Formation of Wait States in Applications with One-sided Communication

Marc-André Hermanns*
German Research School for Simulation Sciences
RWTH Aachen University
Aachen, Germany
m.a.hermanns@grs-sim.de

Manfred Miklosch
Dept. of Computer Science
University of Hagen
Hagen, Germany

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

ABSTRACT

To better understand the formation of wait states in MPI programs and to support the user in finding optimization targets in the case of load imbalance, a major source of wait states, we added in our earlier work two new trace-analysis techniques to Scalasca, a performance analysis tool designed for large-scale applications. In this paper, we show how the two techniques, which were originally restricted to two-sided and collective MPI communication, are extended to cover also one-sided communication. We demonstrate our experiences with benchmark programs and a mini-application representing the core of the POP ocean model.

Keywords

performance analysis, performance optimization, one-sided communication, root cause, critical path

1. INTRODUCTION

The Scalasca performance analysis toolset [4] helps programmers in identifying and understanding parallel performance problems in MPI programs, such as inefficient communication patterns or load imbalance. Recently, we demonstrated how we employ Scalasca's scalable event-trace analysis approach to study the root causes of wait states [3] and to extract the critical path [2]. Rather than just showing symptoms, these analysis methods pinpoint the origins (e.g., load imbalance) of parallel performance bottlenecks. The root-cause analysis sheds light on the formation of wait states by following the wait-state-propagation chain back to the delays that originally caused them. The critical path identifies the program activities that determine the program runtime, and therefore highlights promising optimization targets. However, both techniques were so far limited to programs that use only MPI point-to-point or collective communication.

*Corresponding author

EuroMPI'13, September 15–18 2013, Madrid, Spain

http://dx.doi.org/10.1145/2488551.2488569

With version 2.0, the one-sided communication interface was added to MPI. One-sided communication offers flexibility when implementing certain algorithms and can offer superior performance for certain use cases [13, 16]. To enable such performance gains, deep understanding of an application's dynamic behavior is very important. However, tool support for the analysis of one-sided communication performance beyond simple time profiling is still in its infancy. We previously targeted this support gap in [6], where we showed how Scalasca can detect wait states in MPI-2 one-sided communication constructs in a scalable manner. In this paper, we describe further extensions to Scalasca that enable the root-cause and critical-path analyses for programs that use MPI-2 one-sided communication, again with scalability in mind. With these extensions, it is now possible to identify delays that cause wait states in active-target synchronization constructs. Moreover, in programs that use one-sided in combination with point-to-point or collective communication, we also take into account interactions between synchronization constructs pertaining to any of the three different communication paradigms. This allows us to extract the complete critical path and uncover all long-distance distance effects in the formation of wait states in multi-paradigm programs. As our case studies show, uncovering such cross-paradigm interactions greatly improves the understanding of wait-state formation in such programs.

The remainder of this paper is organized as follows. Section 2 discusses related work with respect to analyzing the performance of one-sided communication, of wait states, and of the critical path. In Section 3, we recap our approach to root-cause and critical path analysis as well as the identification of wait states in MPI active target synchronization. In Section 4, we present the implementation of the infrastructure necessary to enable critical-path and root-cause analysis for active-target communication. We present experimental results, showing both the scalability and the utility of our approach, with synthetic benchmarks and application kernels in Section 5, including a one-sided variant of the CGPOP kernel [17] extracted from the CESM climate code. Finally, we provide a summary and a brief outlook on future work in Section 6.

2. RELATED WORK

By now, many vendors of HPC performance analysis tools support basic performance metrics for one-sided communication. The well-known performance tools Paradyn [11] and

TAU [15] support the analysis of one-sided communication through time-profiling and transfer counts. Vampir [12] supports the fine-grained, manual analysis of traces including one-sided communication in combination with communication statistics, however, leaves the identification of inefficiencies to the expert user. In its support of partitioned global address space languages and runtime engines, the Parallel Performance Wizard supports the analysis of one-sided communication and the identification of bottlenecks [18], yet, relies on the GASP performance interface [19] to obtain measurement data, which is currently only supported by the GASNet communication substrate. Projections [8], as part of the Charm++ programming framework, provides deeper insight into the performance of its one-sided communication infrastructure and overall load balancing, however, it does not support other communication interfaces.

Our root-cause analysis was inspired by the work of Meira Jr. et al. [9, 10] in the context of the Carnival system. Like our analysis, their *cause-effect analysis* characterizes the impact of imbalances on wait states by comparing the execution paths that lead up to a synchronization point. However, our analysis, based on parallel processing of distributed traces, offers far greater scalability than Meira's serial approach. Tallent et al. [20] describe a method integrated in HPCToolkit that characterizes imbalance in call-path profiles of MPI programs by mapping the cost of idleness (i.e., wait states) occurring within globally balanced call-tree nodes (balance points) to imbalanced call-tree nodes descending from the same balance point. Being based on profiles, Tallent's solution can only detect suspects for wait-state root causes within call paths that exhibit global, static imbalance, whereas our trace-based approach accounts for both static and dynamic imbalances. To the best of our knowledge no prior work on imbalance characterization specifically targets the MPI-2 one-sided communication interface.

The identification of the critical path of a parallel application has already been subject to research in the past. ParaGraph [5] is an early tool that could visualize the critical path in a parallel program. Recognizing that the critical path by itself is not overly expressive, Alexander et al. [1] compute near-critical paths to determine the impact of changes on the critical path on the program performance. Schulz [14] and Hollingsworth [7] explore piggybacking as a means to dynamically extract the critical path from MPI and PVM programs, respectively. However, none of these approaches target one-sided communication.

3. APPROACH

Wait states materialize at *synchronization points* of the program, which occur during communication or synchronization whenever one process has to wait for another. Scalasca's trace analysis detects such wait states and classifies them according to various patterns. The two advanced analysis methods, which are subject of this paper, help users point out root causes of wait states and identify promising optimization targets—beyond the mere identification of wait states. The original cause of a wait state is a *delay*, that is, an interval or a set of intervals that causes a process to arrive belatedly at a synchronization point, causing one or more processes to wait. Besides simple computational overload, delays may include a variety of behaviors such as serial operations or centralized coordination activities performed only by one process. Scalasca identifies the locations of delays and calculates their *costs*, which represent the overall amount of waiting time caused by the delay. The delay costs can be much higher than the delay itself, as shown by the example in Figure 1. There, a delay in function `foo` on process A leads to a *Late Sender* wait state on process B, which itself delays the `MPI_Win_post` call on process B and causes a subsequent *Late Post* wait state in `MPI_Win_start` on process C. Hence, the delay costs assigned to `foo` cover the waiting time in `MPI_Recv` on process B for which the delay is directly responsible, and also the bulk of the waiting time in `MPI_Win_start` on process C for which the delay is indirectly responsible. The delay costs therefore highlight those functions in the execution that contribute the most to the waiting time. In addition, the root-cause analysis classifies wait states into direct and indirect, providing valuable insight into wait-state propagation.

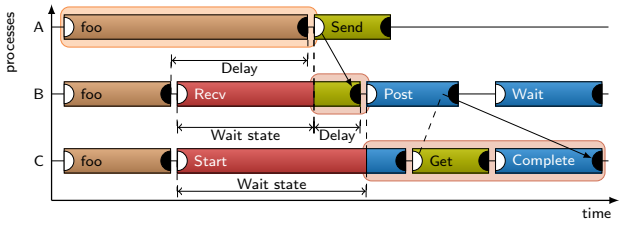
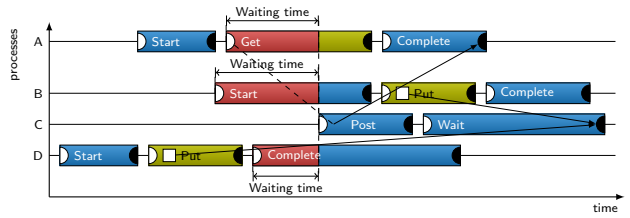
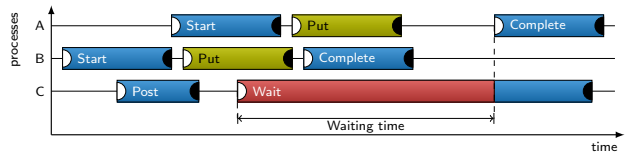


Figure 1: Delay costs and the critical path. The critical path (rectangle with rounded corners) is computed in a backward replay (right to left). Whenever a wait state is encountered on the process currently on the critical path, the critical path changes to the originator of the wait state. Delay costs are also exchanged in a backward replay to allow for the aggregation wait states on the path to the root cause. Indirect wait states are identified with the help of a forward replay, where the originator of a wait states communicates its own waiting time to the waiting process.

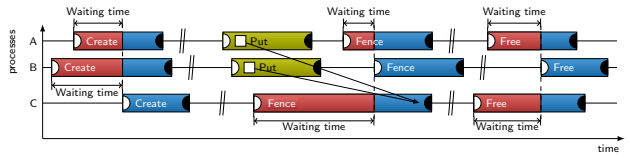
The critical path is the longest path through a program activity graph without wait states. Therefore, it determines the length of the program execution. Prolonging activities on the critical path increases the program runtime, whereas shortening them usually reduces it. In contrast, shortening activities that are not on the critical path only increases waiting time, but does not affect the overall runtime. Hence, the critical path highlights promising optimization targets. In an MPI program, the critical path runs between `Init` and `Finalize`. The critical path ends at the process which entered `Finalize` last. From there, Scalasca follows the critical path backwards through the execution trace. Whenever a wait state is encountered on the current process on the critical path during the backward traversal, the critical path changes to the process causing the wait state. Hence, in Figure 1 the critical path switches from process C to B when C waits for the `Post`, and again from process B to A when B waits for the `Send` on A. The execution times of the functions executed by the process on the critical path are aggregated in a *critical-path profile*. Hence, the critical-path profile contains the time each function and process spent on the critical path, which corresponds to the actual wall-clock execution time for which these functions are responsible. Moreover, a large difference between the critical-path time and the average per-process execution time of a function—the *critical-*



(a) The Late Post pattern. The origins (processes A, B, and D) in general active target synchronization have to wait for the target (process C) to start its corresponding exposure epoch.



(b) The Early Wait pattern. The target (process C) in general active target synchronization closes its exposure epoch before the last origin (process A) completes its access epochs.



(c) The Wait at Create/Fence/Free pattern. Processes have to wait for the last process to enter this collective operation.

Figure 2: Synchronization points in MPI active target synchronization.

path imbalance—pinpoints functions that spend a disproportionate amount of time on the critical path, indicating a parallel bottleneck.

Both root-cause and critical-path analysis require knowledge of synchronization points. In our earlier work [6], we have shown how synchronization points in MPI active target synchronization can be identified at large scale. Identifying synchronization in one-sided communication is challenging—for performance reasons, MPI grants a lot of freedom to the implementation in regards to when synchronization may occur. Only the synchronization functions ending access and exposure epochs are required to block until all pending accesses are completed. For general active target synchronization (GATS), this means that calls to `MPI_Win_start` or any subsequent RMA operation may or may not block. The implementation is free to buffer these until the call to `MPI_Win_complete` occurs and complete all pending operations within this call. To still identify synchronization points within active target synchronization calls, we use a heuristic that assumes a call to block for completion when it overlaps with its remote synchronization counterpart.

Figure 2 shows the three main wait states that may occur in one-sided programs: *Late Post*, *Early Wait*, and the collective *Wait at Fence* and its equivalent in creating and freeing a window. The different synchronization scenarios in the *Late Post* pattern are shown in Figure 2a. The *Early Wait* pattern (see Figure 2b) occurs when the target waits for the last origin to complete its access epoch before closing the exposure epoch. For the collective operations `MPI_Win_create`, `MPI_Win_free`, and `MPI_Win_fence`, we assume a call to be synchronizing when the call is overlapping on all participating processes, as shown in Figure 2c. Through the detection of these wait states, the foundation for identifying their root causes and the analysis of the critical path is laid.

4. IMPLEMENTATION

The extensions to support root-cause and critical-path analysis for one-sided communication build upon the existing implementation of these analyses for point-to-point and collective communication in Scalasca's trace analysis framework [2, 3], leveraging its scalable, post-mortem analysis approach. During execution, each rank of an instrumented target executable records relevant events, such as entering or leaving code regions or sending and receiving messages. After the target application finishes, we launch the trace analyzer with one analysis process per application process. The analyzer performs multiple trace traversals (*replays*), where it iterates over all process-local traces in parallel, and exchanges data required for the performance analysis at each recorded synchronization point. Root-cause and critical-path analysis are performed in a backward replay. A backward replay traverses the trace backwards in time, from end to beginning, and reverses the roles of senders and receivers. Starting at the endmost wait states, the backward replay allows delay costs to travel from the point where they materialize back to the point where they were 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 before knowing the end of the execution.

Given that root-cause and critical-path analysis require the knowledge of the synchronization points on all analysis processes associated with a synchronizing operation, the analysis consists of two phases: the first phase detects and exchanges synchronization information between the analysis processes, and the second phase performs the critical-path detection and delay cost calculation. To enable our advanced analysis techniques for MPI's active target synchronization, these analysis steps needed to be implemented for both active-target synchronization variants—collective fences and group-based GATS. The collective synchronization using fences is very similar to MPI's $N \times N$ collective communication calls. In fact, as their collective nature also allows the use of collective communication during the analysis, the existing infrastructure for collective communication could be reused. However, the potentially dynamic, group-based synchronization available with general active-target synchronization introduces a new dependency scenario: origins may synchronize with multiple targets and targets synchronize with multiple origins. Therefore, it acts as a potential $1 \times N$ or $N \times 1$ synchronization similar to a broadcast or gather, but without the full collective nature of these operations. Unlike other cases in our analysis, where we typically employ the original communication paradigm to exchange information required for their analysis during delay-cost and critical path assessment, we use point-to-point communication to exchange information related to general active-target synchronization. Point-to-point communication provides the flexibility to account for the potentially dynamic list of communication partners in

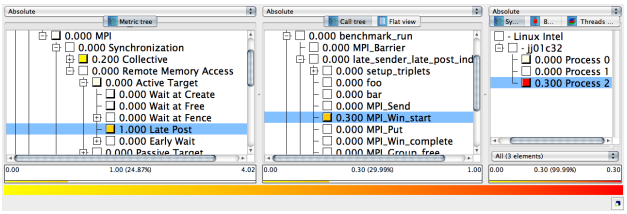
GATS constructs and the dynamic message sizes required for the analysis. Overall, the major adjustments to our analysis infrastructure needed to implement the root-cause and critical-path analysis steps for MPI active-target synchronization are threefold. First, beyond the mere detection of waiting time, as presented in our earlier work [6], we need to identify and store the corresponding rank of the remote process causing the waiting time. Second, we need to build the synchronization graph—a sub-graph of the communication graph comprising only those communications and synchronizations where wait states occured—by communicating synchronization information detected on one analysis process to the corresponding remote process. Third, we need to exchange rank-specific data for the calculation of delay costs along the paths in the synchronization graph.

The identification of the remote process causing the wait state on the waiting process proved to be straight forward. For the *Late Post* pattern, the origin uses get calls to obtain the timestamps of the respective post calls on the targets and now additionally stores the rank associated with the waiting time. For the *Early Wait* pattern, the origin uses the built-in reduction operator `MPI_MAXLOC` where it used a one-sided reduction earlier to determine the largest completion timestamp. These changes enable the association of wait states with the corresponding remote process. That process, however, does not know whether it caused remote waiting time at this point and still needs to be notified. For *Late Post* patterns, each origin sends a message to each target of the current access epoch, indicating either waiting time or no waiting time. Clearly, only one process receives waiting time greater zero, the others receive zero. For *Early Wait* each target likewise informs every origin whether or not the closure of the exposure epoch had to wait for any of its corresponding access epochs to complete.

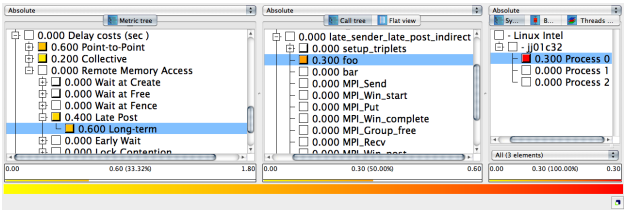
Once the synchronization graph is available, the information required for the root-cause and critical-path analyses can be communicated among its processes in the backward replay. Overall, the delay-cost calculation for general active-target synchronization is structurally similar to the delay-cost calculation for point-to-point *Late Sender* and *Late Receiver* wait-state patterns, which is explained in detail in [3]. In short, when a synchronization point is encountered during the backward replay, the involved processes determine the previous synchronization point with the same set of processes. The period in between those synchronization points is called the *synchronization interval*. Next, the process causing the wait state at the synchronization point receives time profiles of the synchronization interval from each process that incurred a wait state, compares these with its own synchronization-interval time profile, and thereby determines the code regions that exhibited delay. Also, additional costs are assigned to wait states in the synchronization interval on the wait-state causing process, to account for propagating waiting time. This additional cost is communicated further backwards when the trace replay reaches that synchronization point, and used to incorporate long-distance effects into the calculation of delay costs.

5. RESULTS

We tested the implementation of our methods for one-sided communication on different platforms with distinct MPI implementations: the Juropa Linux cluster at Forschungszentrum Jülich using Parastation MPI over Infiniband, the IBM Blue Gene/Q supercomputer Juqueen at Forschungszentrum Jülich using an MPICH-based MPI, and the Linux Infiniband-Cluster of RWTH Aachen University using OpenMPI.



(a) Process 2 waits for 0.3 seconds when starting its access epoch on process 1.



(b) Although process 1 is causing the Late Post on process 2, the delay in `foo` on process 0 is correctly identified as the root-cause.

Figure 3: Waiting time and the delay causing it in a propagation scenario like the one shown in Figure 1.

5.1 Microbenchmarks

We verified our analysis techniques using test cases creating specific, predictable wait states. Figure 3 shows the results of one of our benchmarks in a setup similar to the one shown in Figure 1 in Section 3, highlighting the waiting time for *Late Post* on the origin and the corresponding delay responsible for the wait state. Processes 0 and 2 act as origins, while process 1 acts as the target. As in the wait state scenario explained in Section 3, prior to the one-sided data exchange processes 0 and 1 engage in point-to-point communication, where a delay in function `foo` causes a *Late Sender*, which in turn causes the exposure epoch on process 1 to be started late. Thus the *Late Post* on process 2 is indirectly caused by the delay in `foo` on process 0, which is shown in the analysis report.

5.2 SOR

The SOR benchmark solves the Poisson equation using a red-black successive over-relaxation method on a two-dimensional grid. The communication pattern includes a nearest-neighbor halo exchange and a collective reduction. The halo exchange, though originally using point-to-point communication, was adapted to use general active target synchronization for our work on the wait-state detection [6]. We used this benchmark for scaling tests, because it is easy to configure for weak and strong scaling—although we only performed weak scaling measurements here—while using two common communication patterns: a nearest neighbor exchange and a collective reduction to test for convergence. We configured the benchmark to perform 500 iterations with an error tolerance of 1×10^{-7} to prevent convergence and ensure a predictable number of iterations. This also ensured a significant amount of communication load on one side and

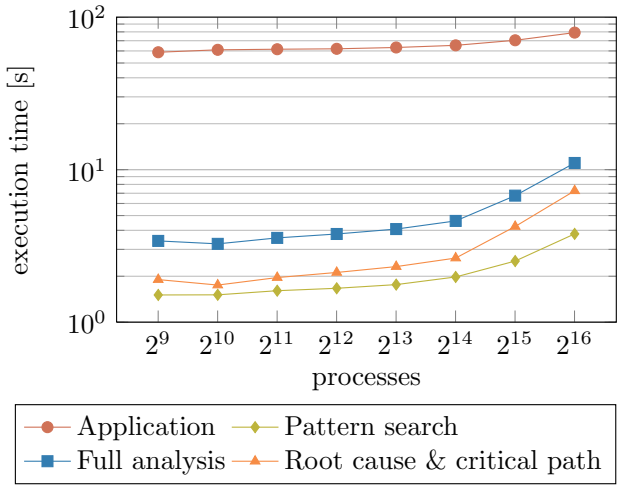


Figure 4: Pattern search as well as root-cause and critical-path analysis of the SOR benchmark at different scales on the Blue Gene/Q supercomputer Juqueen.

a predictable trace size on the other. We configured the application workload to result in approximately 60 seconds runtime for the benchmark itself.

Depending on whether a processes is on the border of the 2D domain or not, local trace sizes range from 5 MB to 10 MB each. Figure 4 shows the scaling behavior of the analysis. Full analysis indicates the sum of the (basic) pattern search and the root-cause and critical-path analyses. The data shows that over all, the analysis scales very well up to our largest measurement with 65,536 cores.

5.3 CGPOP

The CGPOP miniapp [17] represents the conjugate gradient solver of Los Alamos National Laboratory's Parallel Ocean Program (POP) 2.0, which is the ocean model of the Community Earth System Model (CESM), a major climate code developed at the National Center for Atmospheric Research. CGPOP was created to study the most critical part of the application on different platforms without having to port the whole ocean simulation. CGPOP is implemented in several different variants, one of which uses one-sided, point-to-point, and collective communication. As such, it provides a good test case for studying inter-paradigm influences. As a test kernel, CGPOP provides different communication drivers to identify the communication scheme that suits a given platform best. Next to 1D and 2D point-to-point variants, it also provides a 1D halo exchange using one-sided communication with general active target synchronization. The 1D decomposition uses a space-filling curve to partition the data. According to the developers, the one-sided kernel was not investigated deeply, as it did not seem to perform en-par with the two-sided kernels. It therefore posed an interesting test subject for our extended methods. To gain first insights on where the time was actually lost we tested the code with the 180 × 120 tile-set on 60 processes on the RWTH cluster.

To enable measurement of CGPOP, we first needed to modify the code slightly. Initially, group handles were frequently created and not freed, which exceeded the tracking capabilities of our measurement system. As our measurement system already gives us detailed insight into the code's performance, we also disabled any application-internal timing calls. We used this slightly modified version as the starting point of our study. Because the I/O time needed to read in the input data was non-deterministic and dominated the overall execution time, we isolated the solver steps in `solver.esolver` within the analysis report to focus our attention on the performed iteration rather than the initialization. The initial measurements revealed two issues: (1) the barrier, called in the solver step right before the one-sided data exchange, experienced severe waiting time, and (2) despite the barrier, some origins experienced *Late Post* wait states (see Table 1). Initially, the *Late Post* waiting time was not intuitive, as the barrier in front of it should have taken care of any imbalances leading to wait states. However, our root-cause analysis revealed that an imbalanced barrier completion is responsible for the *Late Post* wait state. This is an example of a cross-paradigm wait state, where wait states or imbalances in one communication paradigm influence other paradigms as well. The waiting time in the barrier is caused by delays in the function `matrix_mod_matvec` and its parent function `pcg_chrongear_linear`. For both functions, the delay costs identify two processes as the main contributors to the overall waiting time. The waiting time itself, however, is more wide-spread over the processes. With the underlying nearest neighbor exchange, this indicates that the barrier synchronization may be too heavy-weight in this case. As the barrier itself is not functionally necessary at this point in the code—the one-sided synchronization itself will take care of consistency—we removed it to see how the waiting time caused by the delay materializes in a more light-weight synchronization. As expected, the modifications partly dissolved wait states due to the lighter-weight synchronization, which partly reappeared as *Late Post* wait states. After all, the actual delay causing the initial barrier wait states was not changed. Overall, the waiting time decreased and the application core showed a significant runtime improvement. Furthermore, the critical-path profile shows both user functions to be on the critical path and indicates a significant imbalance. A logical next step would now be to find ways of removing this load imbalance, which, however, is outside the scope of this paper.

Performance Metric	Original [s]	Modified [s]
Time	125.21	108.26
MPI	59.78	45.50
Synchronization	56.20	34.28
Collective	52.98	-
Wait at Barrier	51.95	-
Barrier Completion	0.87	-
RMA	3.23	34.28
Late Post	1.56	26.21

Table 1: Performance metrics of the isolated call-tree of `solver.esolver` in the CGPOP benchmark with the 180 × 120 input tiles on 60 cores of the RWTH cluster.

6. CONCLUSION

When processes wait for one another at synchronization points, wait state occurs. Scalasca's classic search for inefficiency patterns can detect and quantify these wait states. However, to remedy these wait states, a developer has to

know their root cause. Then, a better understanding of the underlying performance problem can aid the optimization process. Furthermore, capturing the critical path can further aid the identification of viable subjects for optimization.

In this paper, we present extensions to our earlier work in root-cause and critical-path analysis. It enables developers to understand the formation of wait states that occur in applications with MPI one-sided communication and active target synchronization. Using microbenchmarks, we explained how these methods can be used to identify indirect wait states, which propagate across multiple processes and communication paradigms. We demonstrated the scalability of our methods with analysis runs on up to 65,536 cores. Furthermore, we showed how our approach found the reason for wait states in a one-sided variant of the conjugate gradient solver CGPOP.

Beyond extending our experiences with our methods on applications with active target synchronization, we plan to enable the identification of wait states in passive target synchronization. Those wait states, however, are not created by delays—i.e., imbalances—but through contention, which would be resolved by controlled imbalances rather than balancing execution times since the last synchronization point. Thus, those synchronization points, or rather *contention points*, need special handling in our advanced analysis methods, which is still subject to further research. Furthermore, as MPI-3 implementations become widely available on different platforms, we plan to extend our analysis to the new functionality introduced in the current MPI standard.

7. REFERENCES

[1] C. A. Alexander, D. S. Reese, and J. C. Harden. Near-critical path analysis of program activity graphs. In *Proc. of the 2nd Intl. Workshop on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS '94)*, pages 308–317, Jan. 1994.

[2] 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 Intl. Parallel & Distributed Processing Symposium (IPDPS), Shanghai, China*, May 2012.

[3] 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 Intl. Conference on Parallel Processing (ICPP), San Diego, CA, USA*, pages 90–100, Sept. 2010.

[4] M. Geimer, F. Wolf, B. J. N. Wylie, E. Abraham, D. Becker, and B. Mohr. The Scalasca performance toolset architecture. *Concurrency and Computation: Practice and Experience*, 22(6):702–719, Apr. 2010.

[5] M. T. Heath, A. D. Malony, and D. T. Rover. The visual display of parallel performance data. *IEEE Computer*, 28(11):21–28, November 1995.

[6] M.-A. Hermanns, M. Geimer, B. Mohr, and F. Wolf. Scalable detection of MPI-2 remote memory access inefficiency patterns. *Intl. Journal of High Performance Computing Applications*, 26(3):227–236, Aug. 2012.

[7] J. K. Hollingsworth. An online computation of critical path profiling. In *Proc. of the SIGMETRICS symposium on Parallel and distributed tools*, 1996.

[8] L. Kalé, S. Kumar, G. Zheng, and C. Lee. Scaling molecular dynamics to 3000 processors with Projections: A performance analysis case study. In *Computational Science — ICCS 2003*, volume 2660 of *LNCS*, pages 23–32. 2003.

[9] W. Meira, Jr., T. J. LeBlanc, and V. A. F. Almeida. Using cause-effect analysis to understand the performance of distributed programs. In *Proc. of the SIGMETRICS symposium on Parallel and distributed tools*, SPDT '98, pages 101–111, 1998.

[10] W. Meira Jr., T. J. LeBlanc, and A. Poulos. Waiting time analysis and performance visualization in carnival. In *SPDT '96: Proc. of the SIGMETRICS symposium on Parallel and distributed tools*, pages 1–10, 1996.

[11] B. P. Miller, M. D. Callaghan, J. M. Cargille, J. K. Hollingsworth, R. B. Irvin, K. L. Karavanic, K. Kunchithapadam, and T. Newhall. The Paradyn parallel performance measurement tool. *Computer*, 28:37–46, November 1995.

[12] M. S. Müller, A. Knüpfer, M. Jurenz, M. Lieber, H. Brunst, H. Mix, and W. E. Nagel. Developing scalable applications with Vampir, VampirServer and VampirTrace. In *PARCO*, pages 637–644, 2007.

[13] W. B. Sawyer and A. A. Mirin. The implementation of the finite-volume dynamical core in the community atmosphere model. *Journal of Computational and Applied Mathematics*, 203(2):387–396, 2007.

[14] M. Schulz. Extracting critical path graphs from MPI applications. In *Proc. of the 7th IEEE Intl. Conference on Cluster Computing*, September 2005.

[15] S. S. Shende and A. D. Malony. The TAU parallel performance system. *Intl. Journal of High Performance Computing Applications*, 20(2):287–311, 2006.

[16] C. Siebert and J. L. Träff. Efficient MPI implementation of a parallel, stable merge algorithm. In *Recent Advances in the Message Passing Interface*, volume 7490 of *LNCS*, pages 204–213, Sept. 2012.

[17] A. Stone, J. Dennis, and M. M. Strout. The CGPOP miniapp, version 1.0. Technical Report CS-11-103, Colorado State University, July 2011.

[18] H.-H. Su, M. Billingsley, and A. D. George. Parallel Performance Wizard: A performance system for the analysis of partitioned global-address-space applications. *Intl. Journal of High Performance Computing Applications*, 24:485–510, November 2010.

[19] H.-H. Su, D. Bonachea, A. Leko, H. Sherburne, M. Billingsley, III., and A. D. George. GASP! a standardized performance analysis tool interface for global address space programming models. In *PARA'06: Proc. of the 8th international conference on Applied parallel computing*, pages 450–459, 2007.

[20] N. R. Tallent, L. Adhianto, and J. Mellor-Crummey. Scalable identification of load imbalance in parallel executions using call path profiles. In *Supercomputing 2010*, Nov. 2010.