Identifying the root causes of wait states in large-scale parallel applications

DAVID BÖHME, Lawrence Livermore National Laboratory, USA
MARKUS GEIMER and LUKAS ARNOLD, Jülich Supercomputing Centre, Germany
FELIX VOIGTLAENDER, RWTH Aachen University, Germany
FELIX WOLF, Technische Universität Darmstadt, Germany

Driven by growing application requirements and accelerated by current trends in microprocessor design, the number of processor cores on modern supercomputers is increasing from generation to generation. However, load or communication imbalance prevents many codes from taking advantage of the available parallelism, as delays of single processes may spread wait states across the entire machine. Moreover, when employing complex point-to-point communication patterns, wait states may propagate along far-reaching cause-effect chains that are hard to track manually and that complicate an assessment of the actual costs of an imbalance. Building on earlier work by Meira Jr. et al., we present a scalable approach that identifies program wait states and attributes their costs in terms of resource waste to their original cause. By replaying event traces in parallel both forward and backward, we can identify the processes and call paths responsible for the most severe imbalances even for runs with hundreds of thousands of processes.

CCS Concepts: •**Computing methodologies** → **Parallel computing methodologies**; •**Software and its engineering** → *Software performance*; •**Theory of computation** → Massively parallel algorithms;

Additional Key Words and Phrases: Performance analysis, root-cause analysis, load imbalance, event tracing, MPI, OpenMP

ACM Reference Format:
David Böhme, Markus Geimer, Lukas Arnold, Felix Voigtlaender, and Felix Wolf. 2016. Identifying the root causes of wait states in large-scale parallel applications. *ACM Trans. Parallel Comput.* 3, 2, Article 11 (July 2016), 24 pages.
DOI: 10.1145/2934661

1. INTRODUCTION

Driven by growing application requirements and accelerated by current trends in microprocessor design, the number of processor cores on modern supercomputers is increasing from generation to generation. With today's leadership computing systems featuring more than a million cores, writing efficient codes that exploit all the available parallelism has become increasingly difficult. Load and communication imbalance, which frequently occurs on supercomputers during simulations of irregular and dynamic domains that are typical

Financial support from the Deutsche Forschungsgemeinschaft (German Research Foundation) through Grant GSC 111, the Helmholtz Association of German Research Centers through Grant VH-NG-118, and the G8 Research Councils Initiative on Multilateral Research, Interdisciplinary Program on Application Software towards Exascale Computing for Global Scale Issues is gratefully acknowledged. This work was partially performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344 (LLNL-JRNL-663039).

DOI: 10.1145/2934661

of many engineering codes, presents a key challenge to achieving satisfactory scalability. As "the petascale manifestation of Amdahl's law" [Vetter (Ed.) 2007], even delays of single processes may spread wait states across the entire machine, and their accumulated duration can constitute a substantial fraction of the overall resource consumption. In general, wait states materialize at the next synchronization point following a delay, which allows a potentially large temporal distance between the cause and its symptom. Moreover, when message-passing applications employ complex point-to-point communication patterns, wait states may propagate across process boundaries along far-reaching cause-effect chains. These chains are hard to track manually and complicate the assessment of the actual costs of a delay in terms of the resulting resource waste. For this reason, there may not only be a large temporal but also a large spatial distance between a wait state and its root cause.

Essentially, a *delay* is an interval during which one process performs some additional activity not performed by other processes, thus delaying those other processes by the time span of the additional activity at the next common synchronization point. Besides simple computational overload, delays may result from a variety of behaviors such as serial operations or centralized coordination activities that are performed only by a designated process. In a message-passing program, the costs of a delay manifest themselves in the form of *wait states*, which are intervals during which another process is prevented from progressing while waiting to synchronize with the delaying process. During collective communication, many processes may need to wait for a single latecomer, which exacerbates the the costs of a delay. 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 the symptoms of delays can be easily detected, the potentially large temporal and spatial distance between symptoms and causes constitutes a substantial challenge in deriving helpful conclusions from this knowledge with respect to remediating the wait states.

The Scalasca performance-analysis toolset [Geimer et al. 2009] searches for wait states in executions of large-scale MPI or hybrid MPI/OpenMP programs by measuring the temporal displacement between matching communication and synchronization operations that have been recorded in event traces. Since event traces can consume a prohibitively large amount of storage space, this analysis is intended either for short runs or for short execution intervals out of longer runs that have been previously identified using less space-intensive techniques such as time-series profiling [Szebenyi et al. 2009]. To efficiently handle even very large processor configurations, the wait-state search occurs in parallel by *replaying* the communication operations recorded in the trace to exchange the timestamps of interest. Building on earlier work by Meira Jr. et al. [Meira,Jr. et al. 1996; Meira,Jr. et al. 1998], we describe how Scalasca's scalable wait-state analysis was extended to also identify the delays responsible and their costs. Now, users can see at first glance where load balancing should be improved to most effectively reduce the waiting time in their programs. To this end, our paper makes the following specific contributions:

— A terminology to describe the formation of wait states
— A cost model that allows the delays to be ranked according to their associated resource waste
— A scalable algorithm that identifies the delays responsible for wait states in SPMD-style applications and calculates the costs of such delays

A version of our method assuming pure MPI programs with two-sided communication was already introduced earlier [Böhme et al. 2010]. Here, we present an extended version capable of handling not only MPI two-sided communication but also OpenMP as well as hybrid programs that use a combination of the two. The hybrid mode is demonstrated in a new case study involving a climate code. Finally, we include updated scalability results, now demonstrating the scalability of our approach with up to 262,144 processes, four times

larger than in our prior work. A version for MPI one-sided communication was published elsewhere [Hermanns et al. 2013] and is not a subject of this article.

The paper is organized as follows. We start with a discussion of related work in Section 2, and introduce the Scalasca tracing methodology in Section 3. In Section 4, we describe the theoretical foundation of the delay analysis, before we present our scalable delay detection approach in detail in Section 5. Section 6 demonstrates the value of our method using three examples, showing its scalability and illustrating the additional insights it offers into performance behavior. Building upon the concepts for the MPI-only case, Section 7 then shows how the delay analysis can be extended to support both pure OpenMP and hybrid OpenMP/MPI programs as well, including an additional case study. Finally, in Section 8, we present conclusions and outline future work.

2. RELATED WORK

Our approach was inspired by the work of Meira Jr. et al. in the context of the Carnival system [Meira,Jr. et al. 1996; Meira,Jr. et al. 1998]. Using traces of program executions, Carnival identifies the differences in execution paths (call paths) leading up to a synchronization point and explains waiting time to the user in terms of those differences. Our analysis is very similar on a conceptual level, but offers far greater scalability than Carnival does. While Carnival employs very moderate pipeline parallelism, our approach has been designed to exploit both distributed memory parallelism on massively parallel systems, which allows it to be applied to codes running on hundreds of thousands of processor cores. Moreover, we define a concise terminology and cost model that is both simple in that it requires only a few orthogonal concepts and powerful in that it can explain the most important questions related to the formation of wait states: which parts of the program (i.e., which call paths) and which parts of the system (i.e., which processes) cause wait states and what are their costs? While Carnival does incorporate wait-state propagation effects into its characterization of waiting time, our approach explicitly distinguishes between short- and long-term wait-state causes. Finally, the visual mapping of delays and their costs onto the virtual process topology offers insights into relationships between the simulated domain and the formation of wait states, as we demonstrate in Section 6.

Using postmortem event-trace analysis, Jafri [Jafri 2007] applies a modified version of vector clocks to distinguish between direct wait states that are caused by some form of imbalance and indirect wait states that are caused by direct wait states via propagation. However, neither does his analysis identify the responsible delays, as ours does, nor does his sequential implementation address scalability on large systems. While his sequential implementation may take advantage of a parallel replay approach to improve its scalability, the general idea of using vector clocks to model the propagation of waiting time from one process to another, which is inherently a forward analysis, may be faced with excessive vector sizes when propagating waiting times across large numbers of processes.

Also influenced by Meira Jr. et al., Morajko et al. [Morajko et al. 2008] determine waiting times and their root causes at runtime. Their approach is based on parallel task-activity graphs that connect communication activities either locally via (computational) process edges or remotely via message edges. Waiting times are calculated on-the-fly using piggyback messages and their values are accumulated separately for every node in the graph. The graph data is extracted at regular intervals to statistically infer the root causes of the aggregated waiting times. While relieving the user from the burden of collecting space-intensive event traces, the piggyback exchange of timestamps (i) requires synchronized clocks and (ii) may introduce substantial intrusion [Schulz et al. 2008]. Furthermore, the statistical inference process may prove inaccurate for applications with highly time-dependent performance behavior [Szebenyi et al. 2008]. Tallent et al [Tallent et al. 2010] describe a method integrated in HPCToolkit [Adhianto et al. 2010] to detect load imbalance in call-path profiles that attributes the costs of wait states occurring within globally balanced call-tree nodes

(balance points) to imbalanced call-tree nodes descending from the same balance point. Similar to Scalasca, their approach uses a scalable parallel program to perform the analysis in a postmortem analysis step. However, by using aggregate profiles as a basis, Tallent's approach can only correlate global wait states with call paths that exhibit global, static imbalance, but cannot identify dynamic effects which may also lead to wait states. It also does not integrate the effect of wait-state propagation into its calculation of imbalance costs.

In the context of performance analysis, critical-path analysis has been widely studied as an approach to optimally direct optimization efforts. Notably, Hollingsworth [Hollingsworth 1996] and Schulz [Schulz 2005] use piggyback messages to extract critical-path data from MPI programs at runtime. Hollingsworth generates a critical-path profile to gain a high-level overview of code routines whose optimization promises the largest runtime reduction, whereas Schulz reconstructs the entire critical path of the program to allow a more fine-grained analysis. Some of the authors present an approach to extract the critical path from event traces in Scalasca [Böhme et al. 2012], from which we derive *performance indicators* that highlight inefficient parallelism in the program. We see the critical-path and delay analyses as complementary approaches. Both methods highlight performance hotspots, but delay analysis specifically sheds light on the formation of wait states.

Recognizing load imbalance as a major concern for parallel performance, several authors have developed approaches to observe and assess uneven workload distributions. Calzarossa et al. [Calzarossa et al. 2004] rank code regions based on their dispersion across the process space to identify the most promising optimization target. Phase profiling [Malony et al. 2005] can expose time-varying workload distributions that would otherwise be hidden when performance metrics are summarized along the time axis. To address the storage implications of the two-dimensional process-time space, Gamblin et al. [Gamblin et al. 2008] apply wavelet transformations borrowed from signal processing to obtain fine-grained but space-efficient time-series load-balance measurements for SPMD codes. Concentrating exclusively on the time axis to avoid communication at runtime, Szebenyi et al. [Szebenyi et al. 2009] use a clustering algorithm to compress time-series call-path profiles online as they are generated. By design, profile-based approaches operate on aggregate data, which can hide the effects of dynamic, temporal load imbalance. Our trace-based approach delivers insights into the actual costs of an imbalance with respect to the formation of wait states, which is a non-trivial undertaking especially in the presence of complex point-to-point communication patterns. However, since the above-mentioned profiling techniques do not require detailed event traces that are too costly to generate for longer runs, they may serve as a basis for identifying suitable candidate execution intervals for our delay analysis.

3. TRACING APPROACH

As the foundation for our subsequent considerations, we start with a review of Scalasca's event-tracing method [Geimer et al. 2009]. Scalasca is a performance-analysis toolset specifically designed for large-scale systems. The current version provides a diagnostic method that supports the localization and quantification of wait states by scanning event traces of parallel applications. Scalasca supports pure MPI, pure OpenMP, and hybrid MPI/OpenMP programs. At present, there are some restrictions: OpenMP programs may not use nested parallel regions or tasks, and hybrid MPI/OpenMP programs must conform to MPI's "funneled" mode, that is, only perform MPI calls on the master thread. Until we discuss the peculiarities of OpenMP and hybrid applications in Section 7, we restrict our narrative to pure MPI.

Scalability is achieved by making the Scalasca trace analyzer a parallel program in its own right. Instead of sequentially processing a single global trace file, Scalasca processes separate process-local trace files in parallel by *replaying* the original communication on as many processes as were used to execute the target application itself. Since trace processing capabilities (i.e., processors and memory) grow proportionally with the number of appli-

cation processes, we were able to complete trace analyses of runs with up to 294,912 MPI ranks on a 72-rack IBM Blue Gene/P system [Wylie et al. 2010] as well as more than one million concurrent threads (16,386 MPI ranks with 64 OpenMP threads each) on a 28-rack IBM Blue Gene/Q system [Wylie 2012].

3.1. Event Model

An event trace is an abstract representation of execution behavior codified in terms of events. Every event includes a timestamp and additional information related to the action it describes. The event model underlying our approach specifies the following event types:

— Entering and exiting code regions. The region entered is specified as an event attribute. The region that is left is implied by assuming that region instances are properly nested.
— Sending and receiving messages. Message tag, communicator, and the number of bytes are specified as event attributes.
— Exiting collective communication operations. This special exit event carries event attributes specifying the communicator, the number of bytes sent and received, and the root process if applicable.

MPI point-to-point operations appear as either a send or a receive event enclosed by enter and exit events marking the beginning and end of the MPI call, whereas MPI collective operations appear as a set of enter/collective-exit pairs (one pair for each participating process). The attributes of the communication events are essential for the parallel replay: they allow us to recreate the original sequence of messages based on the sender and receiver ranks, message tags, and communicator definitions.

3.2. Trace-Analysis Workflow

Figure 1 illustrates Scalasca's trace-analysis workflow. Before any events can be collected, the target application is instrumented, that is, extra code is inserted to intercept relevant events at runtime, generate event records, and store them in a memory buffer before they are flushed to disk. Usually, instrumentation is added in an automated fashion during compilation and linking. In view of the I/O bandwidth and storage demands of tracing on large-scale systems, and specifically the perturbation caused by processes flushing their trace data to disk in an unsynchronized way while the application is still running, it is generally desirable to limit the amount of trace data per application process, so that the size of the available trace buffer is not exceeded. This can be achieved via selective tracing, for example, by recording events only for intervals of particular interest or by limiting the number of time steps during which measurements take place. In this sense, our method should be regarded as an in-depth analysis technique to investigate shorter intervals (e.g., critical iterations of the time-step loop) that were previously identified on the basis of coarser performance data. Since the dilation induced by the instrumentation is roughly proportional to the frequency of measurement routine invocations, it is highly application-dependent and therefore hard to quantify in general terms. For the case studies presented in this paper, the instrumentation-induced runtime dilation did not exceed 7.5%. In our experience, the aforementioned selective tracing techniques suffice to keep measurement dilation reasonably low (below 10%) for the majority of real-world programs. After a target application has terminated and its trace data has been flushed to disk, the trace analyzer is launched using the same number of processes as the target application, with one analysis process per target application process, and loads the entire trace data into its distributed memory address space. In a typical usage scenario, the analyzer is launched within the same batch job as the target application. In principle, however, the analyzer can be launched at any time. Since the computational requirements of the trace analysis are low, it is also feasible to oversubscribe nodes (i.e., allocate multiple processes to each CPU core) as long as the

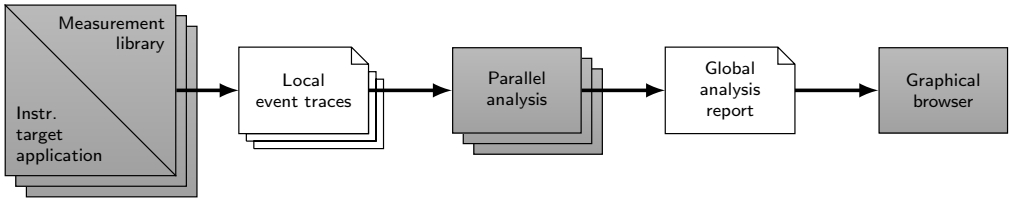


Fig. 1. Scalasca's parallel trace-analysis workflow. Gray rectangles denote programs and white rectangles with the upper right corner turned down denote files. Stacked symbols denote multiple instances of programs or files running or being processed in parallel.

available memory permits performing the trace analysis on fewer physical nodes than the original program.

While traversing the traces in parallel, the analyzer generates a classic call-path profile, accumulating the times spent in individual call-path instances. In addition, the analyzer performs a replay of the application's communication behavior, allowing analysis data to travel along the application's original communication paths. During the replay, the analyzer identifies wait states in communication operations by measuring temporal differences between local and remote events after their timestamps have been exchanged using a similar operation. Every wait-state instance detected by an analysis process is categorized by type (e.g., late sender) and the associated waiting time is accumulated in a local [type, call path] matrix. At the end of the trace traversal, the local matrices are merged into a three-dimensional [type, call path, process] structure characterizing the whole experiment. This global analysis report is then written to disk and can be interactively examined in Scalasca's report explorer. The explorer can visualize the distribution of accumulated waiting times across two- or three-dimensional Cartesian process topologies for every combination of wait-state type and call path (Figures 5, 9, 10, and 12).

To support trace analyses on systems without globally synchronized timers, linear interpolation based on clock offset measurements during initialization and finalization of the target program accounts for major differences in offset and drift. In addition, an extended and parallelized version of the controlled logical clock algorithm [Becker et al. 2009] is optionally applied to compensate for drift jitter and other more subtle sources of inaccuracy.

4. DELAY ANALYSIS

In sharp contrast to the simple wait-state analysis explained above, 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.

4.1. Terminology and Cost Model

To better understand our delay-detection algorithm and the associated cost model, the reader may imagine the execution of a parallel program represented as a timeline diagram with a separate timeline for every process, as shown in Figure 2. The accumulated execution time or resource consumption can then be modeled as the aggregated length of the intervals occupied by some process's activity. In a typical MPI program, this is the wall-clock execution time multiplied by the number of processes under the slightly simplifying assumption that all processes start and end simultaneously. In the following, we will define the terminology underlying our algorithm and cost model.

Wait state. A wait state is an interval during which a process sits idle. The *amount* of a wait state is simply the length of the interval it covers. Wait states typically occur inside a communication operation when a process is waiting to synchronize with another process

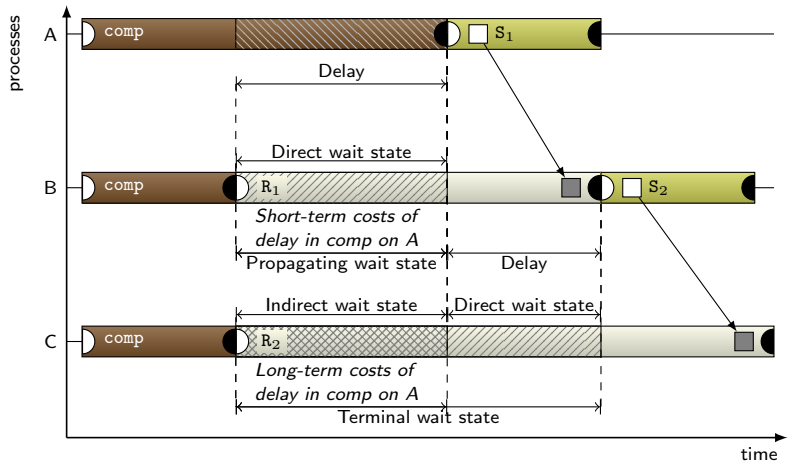


Fig. 2. Timeline diagram illustrating the activities of three processes and their interactions. The execution of a certain code region is displayed as a shaded rectangle and the exchange of a message as an arrow pointing in the direction of the transfer. Regions labeled `S` and `R` represent send and receive operations, respectively. Events recorded in the trace are symbolized as small circles (enter and exit) or squares (send and receive). Process A delays process B due to an imbalance in function `comp`, inducing a wait state in the receive operation $\texttt{R}_1$ of B. The wait state in B subsequently delays process C. Thus, the total costs in terms of waiting time attributed to the delay on process A is defined as the sum of the waiting time it causes directly (on B) and indirectly (on C, via propagation). The former component is called short-term costs and the latter long-term costs.

that has not yet reached the synchronization point. In Figure 2, processes B and C exhibit wait states that are shown as hatched areas. In both cases, the waiting occurs because they are trying to receive a message that has not been sent yet, a situation commonly referred to as *late sender*.

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 was caused by some "intentional" extra activity that does not include waiting time itself. In our example, the wait state in $\texttt{R}_1$ on process B is a direct wait state because it was caused by excess computation of process A in function `comp`. However, by inducing a wait state in process B, this excess computation is indirectly responsible for a wait state in $\texttt{R}_2$ on process C, which is why we call this second wait state *indirect*. The example thus illustrates that wait states may propagate across multiple processes. On the other hand, process C also exhibits a direct wait state produced by communication imbalance: the actual receipt of the message on B delays the dispatch of the message to C.

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 Figure 2, the wait states of process C would be terminal wait states because they do not induce any subsequent wait states. Other examples include wait states at globally synchronizing collective operations (e.g., barrier or allreduce), and wait states that do not lead to further waiting time because the peer process of a subsequent communication operation is also delayed (e.g., due to excess computation or communications with another process). In contrast, *propagating* wait states are those which cause further wait states later on. In the example, the wait state of process B is a propagating wait state because it is responsible for one of the wait states of process C. Both classification schemes

fully partition the set of wait states, but each in different ways. For instance, a terminal wait state can be direct or indirect, but it can never be propagating.

A concept related to wait states and waiting time is the *communication time* of a communication operation. It denotes the total time spent inside that operation minus the waiting time, see for example the time after the wait state in $\texttt{R}_1$ on process B in Figure 2. While communication time is also a form of parallelization overhead, our model considers it a "necessary evil". Hence, if communication time induces a wait state on a peer process later on, such as the communication time of $\texttt{R}_1$ on process B, it is also classified as delay (see below). Thus, our classification scheme is also able to identify inefficient communication that may benefit from overlapping computation and communication using non-blocking operations. We present an example in Section 6.2.3.

Delay. A delay is the original source of a wait state, that is, an interval or a set of intervals that cause a process to arrive belatedly at a synchronization point, causing one or more other processes to wait. In this context, the noun delay refers to the act of delaying as opposed to the state of being delayed. As outlined above, a delay is not necessarily computational in nature and may also involve communication. For example, the decomposition of irregular domains can easily lead to excess communication when processes have to interact with different numbers of neighbors. However, a delay does not include any wait states. Instead, such wait states would be classified as propagating wait states in our taxonomy. In Figure 2, a delay appears on the timeline of process A in region `comp`. This delay is responsible for the direct wait state in $\texttt{R}_1$ on process B and for the indirect wait state in $\texttt{R}_2$ on process C. In addition, some delay is introduced by the actual message receipt (i.e., the communication time) in the receive operation $\texttt{R}_1$.

Costs. To identify the delay whose remediation will yield the highest execution-time benefit, we need to know the amount of wait states it causes. This notion is expressed by the delay costs: the costs of a delay are the total amount of wait states it causes. Since the delay marks 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 a delay causes. The total delay costs are simply the sum of the two. In Section 6, we will see that the long-term delay costs can be much higher than the short-term costs, a distinction that our analysis facilitates. A separation of the delay costs in terms of propagating and terminal wait states is also possible in theory but according to our experience only of minor value to the user. As we show, the result of our 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.

5. SCALABLE DELAY DETECTION AND COST ACCOUNTING

To ensure scalability, delay analysis follows the same parallelization strategy as wait-state analysis, leveraging the principle of a parallel replay of the communication recorded in the trace. However, unlike wait-state analysis, delay analysis requires 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 roles of senders and receivers. Starting at the latest wait states, the backward replay propagates the costs of waiting from the place where they materialize in the form of wait states back to the place where they are caused by delays. Overall, the analysis now consists of two stages:

(1) A parallel forward trace replay that performs the original wait-state analysis and additionally annotates communication events with information on synchronization points and waiting times incurred.

(2) A parallel backward trace replay that for all wait states detected during the forward replay identifies the delays causing them. This stage also divides wait states into propagating vs. terminal and direct vs. indirect wait states.

In the following, we explain our delay analysis in greater detail. During the forward stage, the analysis processes annotate (i) each wait state with the amount of waiting time they measure and (ii) each synchronization point (i.e., communication event of synchronizing nature) with the ranks of the remote processes involved. The annotations are needed later to identify the communication events and synchronization intervals where delays occurred. The actual delay analysis is performed during the backward stage. For each wait state, the algorithm identifies the delays and propagating wait states that contribute to waiting, and calculates short-term and long-term delay costs. As the communication operations are re-enacted in the backward direction in the course of the algorithm's execution, costs are accumulated and transferred back to their source.

Listing 1 outlines the backward stage of our algorithm. Whenever the annotations generated during the forward replay stage indicate that a synchronization point S has been identified in the trace, the root-cause analysis algorithm is invoked simultaneously on each processing location that participated in the corresponding communication or synchronization operation. Synchronization points can occur at MPI point-to-point or collective operations, as well as OpenMP synchronization constructs. For the sake of simplicity, we use MPI terminology for the remainder of this section.

As a first step, all ranks involved in the synchronization point S determine the corresponding synchronization interval I_S (lines 4–5). A *synchronization interval* covers the time between two consecutive synchronization points of the same ranks, where differences in runtime and communication time can cause wait states at the end of the interval. While the communication event associated with the wait state marks the end point of the interval, its beginning is defined by the previous synchronization point where the same group of ranks was involved. Delay costs are calculated and stored on the rank where the delay occurred, that is, the rank which *caused* wait states at the synchronization point. For the common point-to-point *late-sender* case, this is the rank of the sender (see Figure 2). For n-to-1 and n-to-n communication and synchronization operations, such as barrier, all-to-all or (all)gather/(all)reduce, it is the last process that enters the operation. For 1-to-n communications (broadcast), delay costs are assigned to the root process of the operation.

Next, each rank involved in the synchronization point obtains a local processing-time profile p^{local} of the call paths found in the synchronization interval by simply subtracting the waiting times from the runtime profiles (lines 6–8). This allows us to distinguish between delays and propagating wait states. If a call path is not present within the synchronization interval on a process, its processing-time profile entry is implicitly set to zero. The delaying rank now receives the remote processing-time profile p^{rem} as well as the short-term and long-term wait-state costs $\gamma^{\text{rem}}_{\text{short}}$ and $\gamma^{\text{rem}}_{\text{long}}$ from each rank r that incurred a wait state at the synchronization point S. Here, $\gamma^{\text{rem}}_{\text{short}}$ represents rank r's waiting time at S itself, whereas $\gamma^{\text{rem}}_{\text{long}}$ represents the amount of indirect waiting time caused by propagation of that wait state further on.

The delaying rank then calculates the difference profile Δ_p between its local call-path profile and the remote call-path profile (lines 19–21). The runtime difference may be negative for some call paths, for example, when a smaller excess load of the waiting process in some call paths is overridden by a larger excess load of the delaying process in other call paths. Since only positive time differences can contribute to waiting time, we set negative elements to zero. The remaining positive entries represent call paths with delay.

We then update the short-term and long-term delay costs of the call paths in Δ_p by scaling the difference profile entries to match the short-term and long-term wait-state costs $\gamma^{\text{rem}}_{\text{short}}$ and $\gamma^{\text{rem}}_{\text{long}}$, respectively (lines 23–26). The scaling factor s (line 22) divides the remote

Algorithm 1 Delay cost accounting in backward replay.

1: $\gamma_{\text{long}}[*] \leftarrow 0$ ▷ Initialize wait-state cost factors
2: $\text{Delay}_{\text{short}}[*] \leftarrow 0$, $\text{Delay}_{\text{long}}[*] \leftarrow 0$ ▷ Initialize delay-cost profile maps
3: **for all** synchpoints S in local trace **do**
▷ Determine synchronization interval I_S
4: $S_{prev} \leftarrow$ previous synchpoint with same ranks
5: $I_s \leftarrow (S_{prev}, S)$
▷ Determine local processing-time profile p^{local}
6: **for all** call paths c in I_s **do**
7: $p^{\text{local}}[c] \leftarrow \text{runtime}^{\text{local}}(c) - \text{waiting time}^{\text{local}}(c)$
8: **end for**
▷ Transfer processing-time profile and wait-state costs
9: **if** my rank has wait state at (S) **then**
10: $r \leftarrow$ rank causing wait state at (S)
11: send (p^{local} to r)
12: send ($\gamma_{\text{short}} =$ waiting time (S) to r)
13: send ($\gamma_{\text{long}}[S]$ to r)
14: **else if** my rank is causing wait state at (S) **then**
15: **for all** remote ranks r of (S) where r has wait state at (S) **do**
16: receive (p^{rem} from r)
17: receive ($\gamma^{\text{rem}}_{\text{short}}$ from r)
18: receive ($\gamma^{\text{rem}}_{\text{long}}$ from r)
▷ Calculate difference profile
19: **for all** call paths c in I_s **do**
20: $\Delta_p[c] \leftarrow \max(p^{\text{local}}[c] - p^{\text{rem}}[c], 0)$
21: **end for**
▷ Scale difference profile and update delay costs
22: $s \leftarrow 1/(\text{Sum}(\Delta_p) + \text{Sum}(\text{waiting time}(I_s)))$
23: **for all** call paths c in I_s **do**
24: $\text{Delay}_{\text{short}}[c] \leftarrow \text{Delay}_{\text{short}}[c] + \gamma^{\text{rem}}_{\text{short}} \cdot s \cdot \Delta_p[c]$
25: $\text{Delay}_{\text{long}}[c] \leftarrow \text{Delay}_{\text{long}}[c] + \gamma^{\text{rem}}_{\text{long}} \cdot s \cdot \Delta_p[c]$
26: **end for**
▷ Update long-term costs of propagating wait states in I_s
27: **for all** wait states w in I_s **do**
28: $\gamma_{\text{long}}[w] \leftarrow \gamma_{\text{long}}[w] + (\gamma^{\text{rem}}_{\text{short}} + \gamma^{\text{rem}}_{\text{long}}) \cdot s \cdot \text{waiting time}(w)$
29: **end for**
30: **end for**
31: **end if**
32: **end for**

wait-state costs proportionally across all immediate wait-state causes on the process, that is, call paths with excess processing time and propagating wait states. Finally, we update the local long-term wait-state costs γ_{long} of all propagating wait states within the synchronization interval (lines 27–29). As the analysis proceeds, the long-term wait state costs of the propagating wait states in this synchronization interval are passed on to the processes that caused these wait states. Thus, the recursive application of the algorithm accumulates wait-state costs along the original communication path until they reach the beginning of the propagation chain.

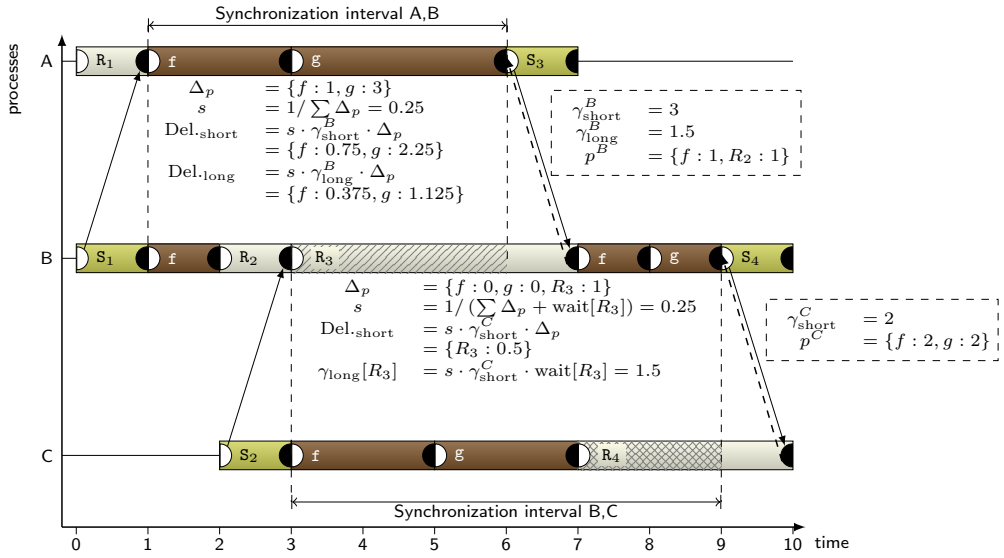


Fig. 3. Source-related accounting of long-term costs via backward replay and successive accumulation of indirectly induced waiting time. The wait state in R_3 receives a cost factor $\gamma_{\text{long}}[R_3]$ for being partially responsible for wait state R_4. Process A receives both short-term and long-term cost factors of wait state R_3 from B, and uses them to calculate short- and long-term delay costs of `f` and `g`, respectively. The x-axis uses an artificial time unit to better illustrate the inner workings of our algorithm by using concrete numbers.

Figure 3 illustrates the delay-cost accounting. The example contains a wait state of length 3 at the receive operation $\mathtt{R}_3$ on process B and a subsequent wait state of length 2 at the receive operation $\mathtt{R}_4$ on process C. Working backward, our algorithm begins its analysis with the wait state at $\mathtt{R}_4$. Process B, whose delayed send operation $\mathtt{S}_4$ leads to this wait state, receives the waiting time costs γ^C_{short} of $\mathtt{R}_4$ and the processing time profile p^C of the corresponding synchronization interval (B, C) from process C. Process B then calculates the difference profile Δ_p by subtracting the remote processing time profile p^C from its local processing time profile for the call paths within the synchronization interval. Since the processing times for functions `f` and `g` are larger on process C than on process B, they do not contribute to the delay and their entries in the difference profile are set to zero. Instead, the only direct cause of the wait state at $\mathtt{R}_4$ is the receive operation $\mathtt{R}_3$, which consists of both waiting and processing time. Using a scaling factor $s = 0.25$ (determined as the inverse of the sum of positive difference profile entries and propagating waiting time), we divide the waiting time costs γ^C_{short} proportionally among the processing and waiting parts of the receive operation $\mathtt{R}_3$: the processing part receives short-term delay costs of length 0.5; and the long-term wait-state costs γ_{long} of the propagating wait state in $\mathtt{R}_3$ are set to length 1.5. Hence, our algorithm determines that 0.5 units of the length 2 wait state at synchronization point $\mathtt{R}_4$ were caused by delay in the communication part of the receive operation $\mathtt{R}_3$, and an amount of 1.5 units was caused by the propagation of waiting time from $\mathtt{R}_3$.

Next, the algorithm proceeds in the backward direction with the analysis of the wait state at synchronization point $\mathtt{R}_3$. Both the short-term and long-term wait-state costs of wait state R_3, γ^B_{short} and γ^B_{long}, are subsequently passed on to the cause of that wait state on process A. Here, again, the algorithm determines the difference profile Δ_p and calculates a scaling factor s. Because of the imbalance in function `f` and the absence of function `g` in the synchronization interval on process A, the difference profile Δ_p now contains $2-1=1$ units of excess processing time for function `f` and 3 units of excess processing time for function `g`. There is no propagating wait state in the synchronization interval. We then calculate the

short-term and long-term delay costs by dividing the short-term and long-term wait-state costs γ^B_{short} and γ^B_{long} proportionally among the entries of the difference profile. Overall, function `f` on process A receives 0.75 units of short-term and 0.375 units of long-term delay costs; function `g` on process A receives 2.25 and 1.125 units of short-term and long-term delay costs, respectively; and the receive operation $\mathtt{R}_3$ on process B receives 0.5 units of short-term delay costs. The sum of delay costs equals the overall length of waiting time (5 units), and their distribution reflects the relative contribution of the respective call paths and processes to the observed waiting time.

Note that a single delay can appear in multiple, overlapping synchronization intervals. The total costs of the delay are then simply aggregated. Also, while the pseudo-code description of our algorithm suggests linear runtime complexity over the number of ranks involved in a collective synchronization point, our actual implementation employs reduction operations to aggregate delay costs across ranks with logarithmic complexity. For point-to-point communication, the algorithm coalesces different payloads (i.e., p^{local}, γ_{short} and γ_{long}) into a single message to reduce the number of message exchanges.

6. EVALUATION

We evaluate our approach with respect to both scalability and functionality. For this purpose, we conducted experiments using three different MPI codes – the ASCI benchmark Sweep3D [Accelerated Strategic Computing Initiative 1995], the astrophysics simulation Zeus-MP/2 [Hayes et al. 2006], and the plasma-physics code Illumination [Geissler et al. 2006; Geissler et al. 2007]. All measurements were taken on the 72-rack IBM Blue Gene/P supercomputer Jugene at the Jülich Supercomputing Centre.

6.1. Scalability

As it is the most scalable code of our ensemble, the scalability of the delay analysis is demonstrated using Sweep3D. This code is an MPI benchmark performing the core computation of a real ASCI application, a 1-group time-independent discrete ordinates neutron transport problem. It calculates 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, corresponding to one of the eight directed diagonals of the grid. Sweep3D uses an explicit two-dimensional decomposition (i, j) of the three-dimensional computational domain, resulting in point-to-point communication of grid-points between neighboring pro-

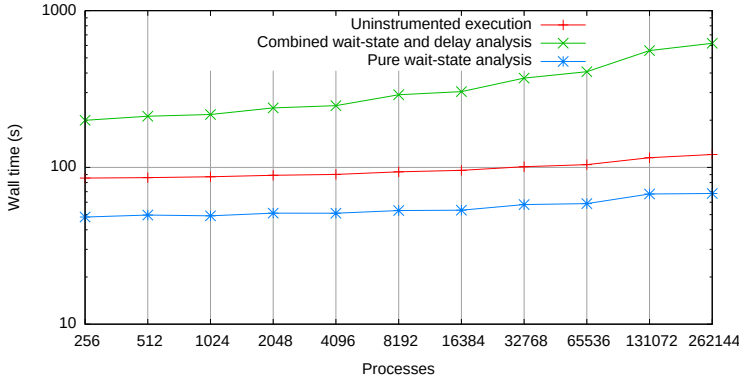


Fig. 4. Comparison of Sweep3D application execution time, combined wait-state and delay analysis time, and pure wait-state analysis time at various scales.

Table I. Uncompressed trace sizes for Sweep3D scalability experiments.

# Processes	256	1,024	4,096	16,384	65,536	262,144
Trace size (GiB)	6.3	26.2	106.3	428.7	1,721.6	6,900.0

cesses, and reflective boundary conditions. A wavefront process is employed in the i and j directions, combined with pipelining of blocks of k-planes and octants, to expose parallelism.

To demonstrate the scalability of our approach, we performed analyses of traces collected with up to 262,144 processes, configured in weak scaling mode with a constant problem size of $32 \times 32 \times 512$ cells per process. The experiments produced about 27 MiB of uncompressed trace data per process, which amounts to 6900 GiB for the largest configuration with 262,144 processes (Table I). Note that traces are stored in compressed form, which typically reduces disk space requirements by a factor of two compared to the uncompressed trace sizes.

The elapsed times for the benchmark runs with all user and MPI routines instrumented for trace collection were within 5% of the times for the uninstrumented versions, which suggests an acceptably small measurement dilation. Figure 4 compares the wall-clock execution times of the combined wait-state and delay analysis with (i) the uninstrumented Sweep3D application and (ii) the pure wait-state analysis (i.e., executing only the forward stage of our two-pass algorithm). The 10-fold increase in the number of processes and the resulting large range of times necessitates a log-log scale in the plot. Since it is not the subject of this study, the analysis times do not include loading the traces, which took on average 206 s at the largest scale. The time required for the trace analysis heavily depends on the amount of communication in the original program. In our case, the combined analysis needed appreciably more time than the pure wait-state analysis. Because our two-phase analysis performs more message exchanges than the original program, the combined trace analysis can sometimes take more time than the original application run for communication-intensive programs like Sweep3D in our configuration. This is typically not the case for computation-bound HPC programs. Importantly, both pure wait-state and combined wait-state and delay analysis scaled almost equally well. As expected when replaying the original communication, both curves run similar to the uninstrumented execution, although the combined analysis shows some divergence at larger scales. We believe that the increase in trace-analysis time can be tolerated even for configurations larger than 262,144 – justified by the improved understanding of wait-state formation, as demonstrated below. Nevertheless, our prototypical implementation still leaves room for improvement with respect to algorithmic optimizations. We are confident that the additional runtime required to execute the backward stage of the delay analysis can be further reduced in a production version.

6.2. Functionality

The functional capabilities of our approach, that is, the insights it offers into the formation of wait states, are shown using all three codes.

6.2.1. Sweep3D. This benchmark has been comprehensively modeled and examined on a variety of systems and scales [Hoisie et al. 1999; Sundaram-Stukel and Vernon 1999], and was also the subject of a scaling study [Wylie et al. 2010] on Jugene using Scalasca but without our delay analysis. While the study generally confirmed the good scaling behavior of Sweep3D, it also identified computational load imbalance and MPI waiting time growing with scale. We successively refined the analysis by manually subdividing the `sweep` subroutine into more fine-grained code blocks using Scalasca's manual instrumentation API, which pointed to the section of code applying "fixup" corrections as the source of the load imbalance. To assess the influence of the load imbalance on the formation of wait states, we applied our delay analysis to a trace acquired during the scaling study on 16,384 cores. In this configuration, 11% of the total execution time was spent in late-sender wait states.

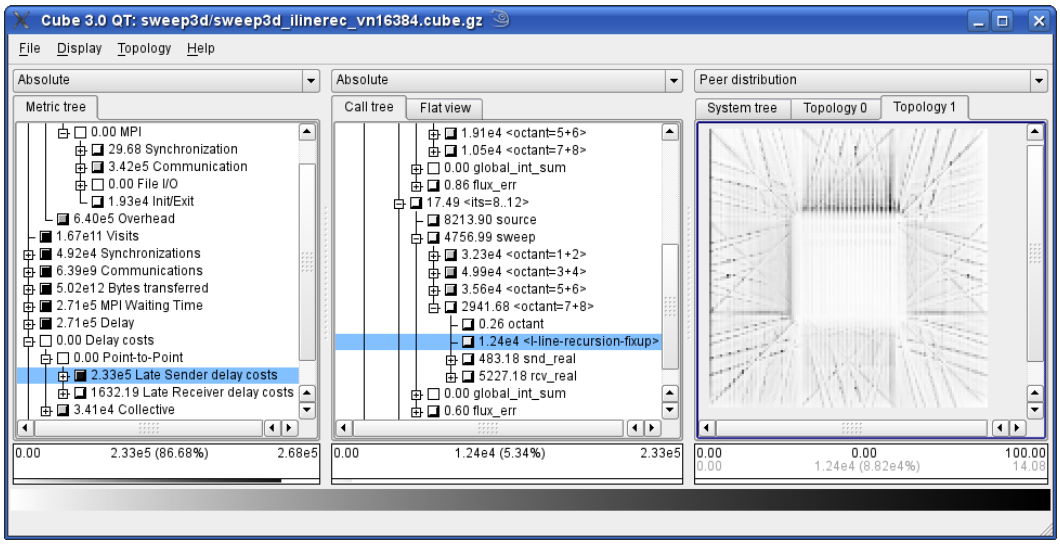
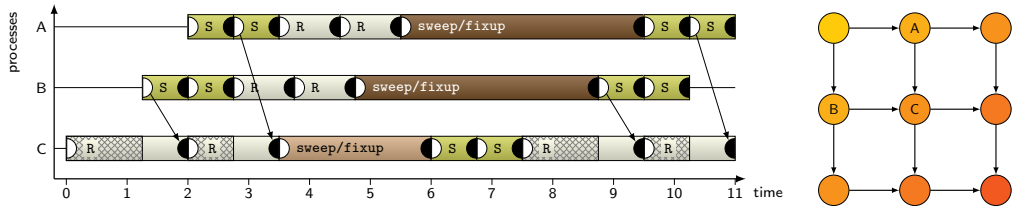


Fig. 5. Sweep3D analysis result in the Scalasca report browser. The delay costs metric (left pane) identifies fixup sections as a major cause of wait states (middle pane). As the sweep in the selected octant originates in the upper left corner of the virtual process grid, delays occur on the upper and left edge of the underloaded rectangular inner region and along intricate line patterns of overloaded processes (right pane).



(a) Timeline (b) Location

Fig. 6. Wait-state formation in Sweep3D: At the boundary of over- and underloaded processes, process C has to wait for its upper and left neighbors (processes A and B) because of delays in the `fixup` computations.

To advance the wavefronts across the two-dimensional grid, Sweep3D employs a complex communication pattern with blocking point-to-point communications that facilitate the propagation of wait states. And indeed, our delay analysis confirmed that almost 90% of the waiting time is indirect. To isolate the locations of its root causes, we examined the delay cost metric in the Scalasca report browser (Figure 5). The results confirm delays in the `fixup` code sections as primary causes of wait states.

Figure 6(a) shows an abstract reconstruction of the problem with three processes sitting at the boundary between over- and underloaded processes. Imagine process C's location at the upper left corner of the underloaded inner square visible in Figure 5. The wavefront is traveling from the upper left to the lower right corner. Processes A and B are C's upper and left neighbors, respectively, already belonging to the outer square (Figure 6(b)). Since the time spent in `fixup` is reduced at the transition from the outer to the inner square, processes A and B spend more time in `fixup` during `sweep` than process C. This leads to late-sender wait states on C. Even worse, these wait states delay process C's own send operations, so that the waiting time propagates along the wavefront's direction of travel. Actually, the border between the inner and outer square is not a sharp line but rather a range of processes with a steep gradient of the time spent in fixup. For this reason, the

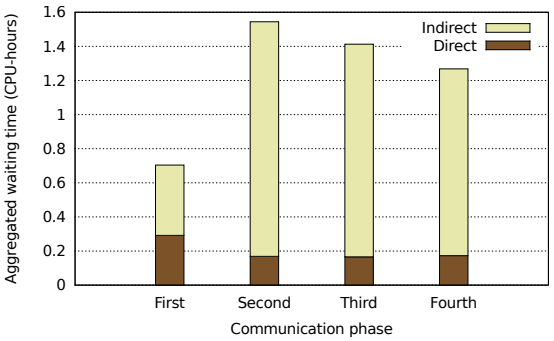


Fig. 7. Composition of waiting time in different communication phases of Zeus-MP/2.

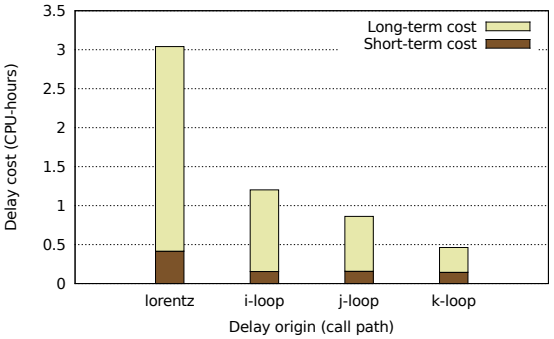


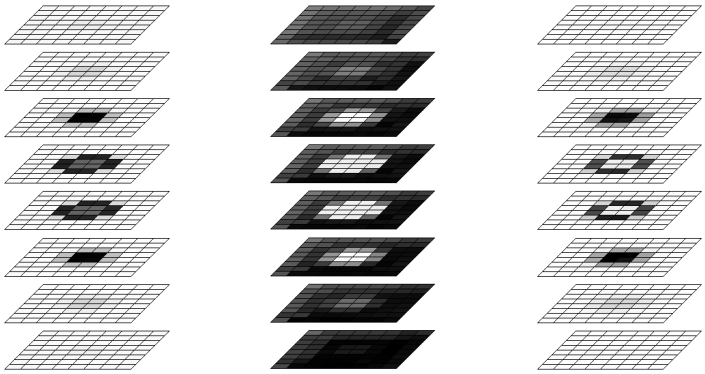
Fig. 8. Short- and long-term costs of delays in four critical call paths of Zeus-MP/2.

wait states not only propagate but also accumulate as they follow the wavefront down the slope of the gradient towards the center.

Overall, the `fixup` delays are responsible for almost 37% of the total waiting time, thus forming the largest singular root cause. This is especially remarkable since the fixups are only applied in 5 out of 12 iterations. Other major causes are delays in the non-waiting parts of the communication functions (responsible for 25% of the waiting time) and delays in the remaining, only slightly imbalanced computation inside the `sweep` routine (responsible for another 31% of the overall waiting time). The first category can be explained by the inability of a process to satisfy horizontal and vertical neighbors at the same time. The exchange of data first in the horizontal direction assigns a special role to the vertical border of the grid from where most of this waiting time is spread (not shown).

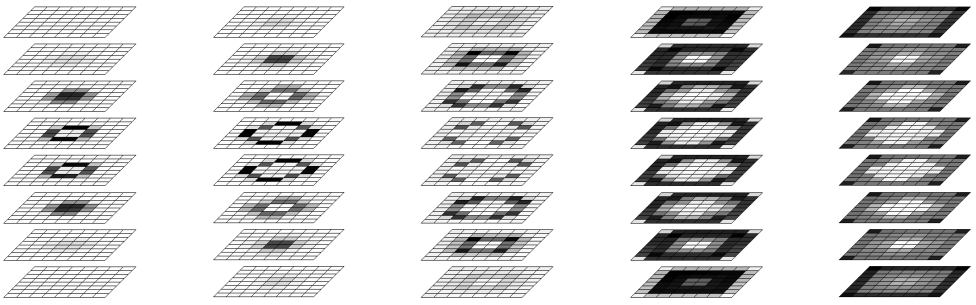
All in all, the delay analysis proved the major role of the imbalanced `fixup` in the formation of wait states. In spite of the code's challenging communication pattern with its far-reaching wait-state propagation, the delay analysis was able to identify different root causes and to quantify their contribution to the performance problem. This allowed even the noticeable impact of small computational delays within the `sweep` routine to be identified.

6.2.2. Zeus-MP/2. Our second case study is the astrophysical application Zeus-MP/2. The Zeus-MP/2 code performs hydrodynamics, radiation-hydrodynamics (RHD), and magnetohydrodynamics (MHD) simulations on 1-, 2-, or 3-dimensional grids. For parallelization, Zeus-MP/2 decomposes the computational domain regularly along each spatial dimension and employs a complex point-to-point communication scheme using non-blocking MPI operations to exchange data between neighboring cells in all active directions of the com-



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

Fig. 9. Variation of computation time, waiting time, and total delay costs between processes in Zeus-MP/2 across the three-dimensional computational domain. Darker colors indicate higher values. A time profile shows the workload imbalance pattern (a) and the distribution of waiting times (b). The delay analysis (c) locates the precise wait-state root causes on the border of the inner, imbalanced region.



(a) Delay costs (b) Direct waiting time in 1st comm. phase (c) Indirect waiting time in 1st comm. phase (d) Indirect waiting time in 2nd comm. phase (e) Indirect waiting time in 3rd comm. phase

Fig. 10. Propagation of wait states caused by delays in the `lorentz` subroutine. These delays cause direct wait states in the first communication phase, which induce indirect wait states at the surrounding layer of processes and travel further outward during the second and third communication phase.

putational domain. We analyzed the 2.1.2 version on 512 processes simulating a three-dimensional MHD wave blast, based on the provided "mhdblast_XYZ" example configuration We limited the number of time steps to 100 to constrain the size of the recorded event trace. As in the previous example, our analysis targets the origins of wait states.

Using 512 processes, the wave-blast simulation requires 52.2 hours of CPU allocation time in total, 12.5% of which is waiting time. Most of this waiting time can be attributed to late-sender wait states in four major communication phases occurring in each iteration of the main loop, denoted in the following as first to fourth communication phases. As Figure 7 shows, the predominant part of the waiting time in these communication phases is indirect. Regarding the root causes of the waiting time, our delay analysis identified four call-path locations as major origins of delay costs: the `lorentz` subroutine and three computational loops within the `hsmoc` subroutine, which we refer to as `i`-loop, `j`-loop, and `k`-loop in the remainder of this paper. Within the main loop, the `lorentz` subroutine is placed before the first communication phase, the `i`-loop before the second, and the `j`-loop and `k`-loop before the third and fourth communication phases, respectively. Figure 8 illustrates the mapping

of short- and long-term delay costs onto the call paths responsible for the delay. Especially the `lorentz` routine and the `i`-loop region exhibit a high ratio of long- versus short-term delay costs, indicating that delays in these call paths indirectly manifest themselves as wait states in later communication phases.

The visualization of the virtual process topology in the Scalasca report browser allows us to study the relationship between waiting and delaying processes in terms of their position within the computational domain. Figure 9(a) shows the distribution of workload (computational time, without time spent in MPI operations) within the main loop across the three-dimensional process grid, as determined by classic profiling. The arrangement of the processes in the figure reflects the virtual process topology used to map the three-dimensional computational domain onto the available MPI ranks. Obviously, there is a load imbalance between ranks of the central and outer regions of the computational domain, with the most underloaded process spending 76.7% (151.5 s) of the time of the most overloaded process (197.4 s) in computation. Accordingly, the underloaded processes exhibit a significant amount of waiting time in MPI communication as determined by Scalasca's wait-state search during the forward replay stage (Figure 9(b)).

Examining the distribution delay costs reveals the root causes of the wait states (Figure 9(c)) within the computational domain. In comparison to the computation time distribution overview obtained with classic profiling, the delay analysis is able to pinpoint the locations of the wait-state inducing processes more precisely: they lie on a hollow sphere on the border of the central, overloaded region. This observation is easily explained: within the central and outer regions, the workload is relatively well balanced. Therefore, communication within the same region is not significantly delayed. In contrast, the large difference in computation time between the central and outer regions causes wait states at synchronization points along the border.

Our findings thus indicate that the majority of waiting time originates from processes at the border of the central topological region. Indeed, visualizing direct and indirect wait states separately confirms the propagation of wait states. Figure 10 shows how delay in the `lorentz` subroutine at the border of the central region causes direct wait states in the surrounding processes during the first communication phase, which in turn cause indirect wait states within the next layer of processes and propagate further to the outermost processes during the second and third communication phases.

6.2.3. Illumination. Our last pure-MPI case study is Illumination, a 3D parallel relativistic particle-in-cell code for the simulation of laser-plasma interactions [Geissler et al. 2006; Geissler et al. 2007], where our method was able to shed light onto an otherwise obscure performance phenomenon. The code uses MPI for communication and I/O. In addition, the Cartesian topology features of MPI simplify domain decomposition and the dynamic distribution of tasks, allowing the code to be easily executed with different numbers of cores. The three-dimensional computational domain is mapped onto a two-dimensional logical grid of processes. As in the case of Sweep3D, the logical topology can be conveniently visualized in the Scalasca report browser.

We examined a benchmark run over 200 time steps on 1024 processors. The traditional wait-state analysis showed that the application spent 55% of its runtime in computation and 44% in MPI communication, of which more than 90% was waiting time in point-to-point communication (Figure 11). In particular, a large amount of time – 91% of the waiting time, and 36% of the overall runtime – was spent in *late-receiver* wait states, in which a send operation is blocked until the corresponding receive on the peer process has been posted. This can happen during the transfer of voluminous messages, when buffer space is scarce enough to demand synchronous exchange. There was also a notable computational load imbalance in the program's main loop, as shown in Figure 12(a) by the distribution of computation time across the process grid, where processes within a circular inner region obviously need

more time than those outside. The measured computation time varies between 16 and 22 seconds per process.

Using the delay analysis, we were able to determine the root cause of the late-receiver wait states. Interestingly, the delay costs show only a negligible influence of the computational imbalance on the waiting time. Instead, more than 75% of the delay costs are assigned to `MPI_Send` and `MPI_Recv` operations, which indicates that the MPI communication itself is the dominant cause of the wait states. Moreover, more than 95% of the waiting time is indirect; hence individual delays heavily impact subsequent communication operations. Finally, the distribution of delay costs in Figure 12(b) shows that the processes with the highest delay costs form stripes across the process grid with no resemblance to the circular imbalance pattern, which is only faintly visible in the background. This confirms that the dominant performance bottleneck is in fact unrelated to the computational imbalance. Overall, our findings suggest that the main problem was actually an inefficient communication pattern: the blocking `MPI_Send` imposes a strict order in which messages are processed, so that any delay or wait state early on in the message chain impedes the communication progress and rapidly spreads wait states to neighboring processes.

After replacing the blocking MPI communication routines in the code with their non-blocking counterparts and using `MPI_Waitall` to complete outstanding requests in an arbitrary order, the waiting time is substantially reduced. Against the background of our analysis, this now seems plausible because wait states in one operation no longer delay subsequent communication calls. We repeated our performance analysis with the revised version of the code. As Figure 11 illustrates, this version indeed shows a significant performance improvement. More than 80% of the program runtime is now consumed by computation, 11% by wait states in collective communication, and only 5% by wait states in point-to-point communication. The computational load imbalance remains, but the delay analysis now identifies delay within the computational part of the main loop as the source of the waiting time. Also, the topological distribution of the delay costs across the process grid, as depicted in Figure 12(c), now resembles a ring pattern around the inner imbalanced region of the domain, and accentuates the computation time gradient between the overloaded inner and the underloaded outer regions. Hence, the delay analysis confirms the load imbalance as the single root cause of the bulk of waiting time and thus indicates that the waiting time cannot be significantly reduced any further without actually resolving the load imbalance itself.

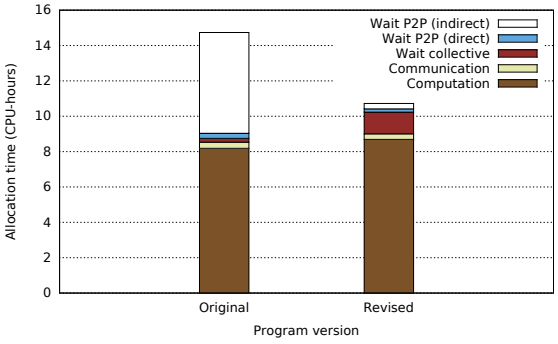
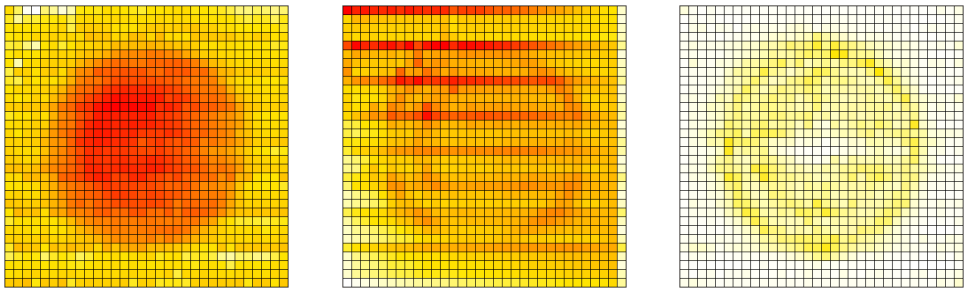


Fig. 11. Runtime composition and quantitative comparison of the original versus the revised version of Illumination. In the revised version, the indirect waiting time was significantly reduced and wait states were partially shifted from point-to-point to collective communication. A slight increase in computation time was caused by additional memory copies needed in the context of the switch to non-blocking communication.



(a) Computation (original) (b) Delay costs (original) (c) Delay costs (revised)

Fig. 12. Comparing the original with the revised (i.e., optimized) version of Illumination by visually mapping computational load and delay costs onto the two-dimensional virtual process topology. The distribution of delay costs in the original version (b) shows stripes of processes with high delay costs, which suggests a bottleneck that is unrelated to the circular computational imbalance pattern (a). In the optimized version (c), delay costs are significantly lower overall and match the imbalance pattern.

7. HYBRID MPI/OPENMP PROGRAMS

Identifying wait-state root causes is further complicated in hybrid, multi-paradigm programs, where wait states may spread across synchronization points belonging to different parallelization paradigms. In this section, we show how our delay analysis supports the identification of wait-state root causes across paradigm borders for the combination of MPI and OpenMP.

7.1. Delay Cost Model for Hybrid Programs

For OpenMP parallelization, we distinguish two different inefficiency patterns that result in wasted computing resources. The first pattern comprises wait states at explicit or implicit barriers at the end of a parallel region, which occur as a result of load imbalance between threads within the parallel region. The second inefficiency pattern comprises *idle threads*, which describes the inactive state of non-master threads outside of parallel regions. This inefficiency pattern often occurs in legacy applications which have been retrofitted with OpenMP parallelization of compute-intensive parts, but still execute a significant portion of work outside of parallel regions in a single thread. As a result of Amdahl's law, these serial phases eventually limit scalability and waste computing resources.

These considerations are especially relevant for hybrid MPI/OpenMP programs, where MPI communication is often funneled through a single thread. Here, wait states in the MPI communication also extend the time where the non-communicating threads of the waiting process are idle, thereby increasing the costs of the original delay. Conversely, delays within parallel regions may affect subsequent MPI communication. Figure 13 illustrates possible interactions at the boundary of OpenMP and MPI paradigms and how they are captured by the delay analysis. There, we see a delay on the non-master thread of process A, which leads to a wait state at the following OpenMP barrier. After the end of the first parallel region, the program performs MPI communication. During that time, the non-master threads on both processes are idle. This idle phase is prolonged on the second process by a wait state in the receive operation. Similar to the characterization of delay costs in the MPI-only case, the delay analysis determines the costs of the delays in terms of resources wasted in MPI or OpenMP wait states as well as worker-thread idleness.

Extending the backward trace replay approach for the detection of delays in OpenMP constructs is straightforward. We follow the same algorithm as outlined in Listing 1, except that the acting agents are now OpenMP threads instead of MPI ranks, and data is passed across threads via shared variables instead of explicit communication operations. OpenMP

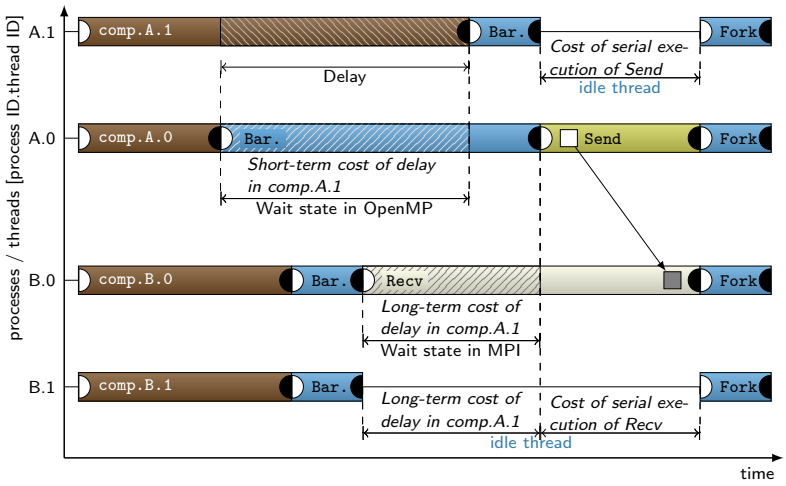


Fig. 13. Interactions of wait states in a hybrid program using two MPI processes with two OpenMP threads each. Delay costs cover wait states in OpenMP barriers and MPI operations as well as worker-thread idleness. Note that the master thread timelines are both shown in the middle.

synchronization points, such as forking a thread team, or implicit or explicit barriers, are represented as artificial regions enclosed by an enter and exit event record in the trace. The length of, for example, a barrier region represents the time the corresponding thread spent in the barrier. The characterization of delays responsible for wait states in OpenMP barriers is similar to the characterization of MPI barriers, that is, delay costs are assigned to the last thread entering the OpenMP barrier. Likewise, delay costs of worker-thread idleness in between two parallel regions are assigned to the master thread.

7.2. Case Study: CESM Sea Ice Model

To demonstrate the usefulness of our approach with respect to hybrid MPI/OpenMP programs, the delay analysis was applied to a hybrid version of the sea-ice component of the Community Earth System Model (CESM) [University Corporation for Atmospheric Research (UCAR) 2012]. The measured configuration used 1024 MPI processes with 4 OpenMP threads each on the Blue Gene/P system Jugene. Using the delay analysis, we shed light on cross-paradigm delay propagation effects that lead to a significant amount of wasted resources in this configuration.

For the following discussion, we ignore the initialization and cleanup phases of the program and focus on the main iteration. Here, the program switches between multi-threaded computation phases and single-threaded communication phases, where data between processes is exchanged via MPI. Overall, the examined configuration occupied $\sim$ 274 hours of CPU allocation time. The left bar in Figure 14 shows the distribution of the allocation time. Only 27% of the allocation time is used for useful work (computation and MPI communication, excluding wait states), whereas more than half of the allocation is wasted idling while worker threads wait for single-threaded phases of the program to complete. An additional 13% of the allocation time is spent in late-sender wait states in MPI communication, while the time spent waiting at OpenMP barriers is negligible.

With the delay analysis, we can examine the root causes that lead to the idle threads. The three bars on the right in Figure 14 show the sum of the runtime and delay costs attributed to the application's two main kernels (`dyn_evp` and `transport_driver`) and the nearest-neighbor exchange routine (`haloupdate`). The nearest-neighbor data exchange is performed only on the master thread, while the worker threads are idle. Unsurprisingly,

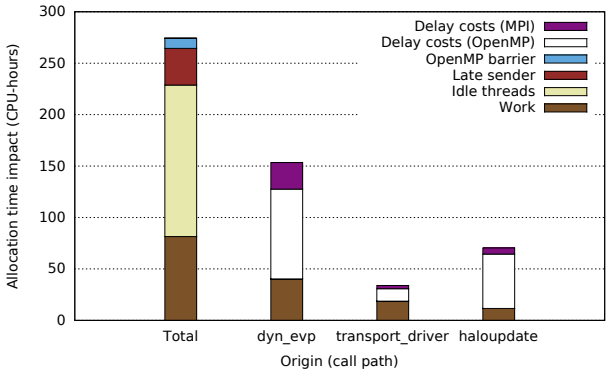


Fig. 14. Distribution of overall CPU allocation time in the CESM Sea Ice Model (left bar), and CPU allocation time / delay costs of three major kernels.

the MPI communication and computation within the `haloupdate` function therefore incur a significant amount of direct delay costs for the resources wasted in idling worker threads. Moreover, late-sender wait states in the MPI synchronization points within `haloupdate` prolong the single-threaded phases on the affected processes further. The distribution of delay costs shows that the `dyn_evp` kernel is responsible for most of the late-sender waiting time, and indirectly also for the largest part of the worker-thread idleness. This is because load imbalance between processes causes late-sender wait states at MPI synchronization points, during which the OpenMP worker threads on the waiting process also remain idle. The delay analysis accurately captures these cross-paradigm effects in the calculation of the delay costs. For the `dyn_evp` kernel, which requires 40 CPU-hours of computation time, the wait states caused by load imbalance in this kernel occupy another 113 CPU-hours of the allocated partition – almost three times as many as the actual computation. Hence, by highlighting the effects of delays – even across interactions between different parallelization paradigms – the delay analysis allows the true performance impact of a kernel to be understood.

8. CONCLUSION AND OUTLOOK

Wait states induced in the wake of load or communication imbalance present a major scalability challenge for applications on their way to deployment on peta- and exascale systems. Our work contributes towards a solution of the problem by allowing delays responsible for the formation of wait states both (i) to be identified and (ii) to be quantified in terms of the amount of waiting time they cause – even if those wait states materialize in different program components or much later in the course of the program's execution. This cost attribution is essential, since the resulting wait states may consume far more resources than the delaying operation itself. Compared to earlier work, our approach is based on a parallel replay of event traces both in the forward and in the backward direction, which allowed non-trivial insights into the wait-state propagation occurring in three MPI and one hybrid MPI/OpenMP example codes running on up to 262,144 cores.

Unfortunately, the excess workload identified as a delay usually cannot simply be removed. To achieve a better balance, optimization hypotheses drawn from a delay analysis typically propose the redistribution of the excess load to other processes instead. A variety of load-balancing strategies exist; however, implementing them often requires significant re-engineering of the application, which is typically too costly for extensive evaluation trials with multiple options. Therefore, we want to develop a load-balancing simulator that can be used to compare different load balancing strategies in a program without having to

modify the application source code. The simulation would rather employ a dynamic model of communication and computation phases previously extracted from the application via measurement.

ACKNOWLEDGMENTS

We would like to thank John Dennis and Rich Loft from the National Center for Atmospheric Research, Boulder, Colorado, USA, and Monika Harlacher, at that time staff at the German Research School for Simulation Sciences, for their assistance in running our experiments with the sea-ice component of CESM. Finally, the authors would like to express their gratitude for the computing time granted on the supercomputer Jugene at the Jülich Supercomputing Centre.

REFERENCES

Accelerated Strategic Computing Initiative. 1995. The ASCI SWEEP3D Benchmark Code. http://www.ccs3.lanl.gov/pal/software/sweep3d/sweep3d_readme.html. (1995).

Laksono Adhianto, Sinchan Banerjee, Michael W. Fagan, Mark Krentel, Gabriel Marin, John Mellor-Crummey, and Nathan R. Tallent. 2010. HPCToolkit: Tools for performance analysis of optimized parallel programs. *Concurrency and Computation: Practice and Experience* (April 2010).

Daniel Becker, Rolf Rabenseifner, Felix Wolf, and John C. Linford. 2009. Scalable timestamp synchronization for event traces of message-passing applications. *Parallel Comput.* 35, 12 (2009), 595–607.

David Böhme, Bronis R. de Supinski, Markus Geimer, Martin Schulz, and Felix Wolf. 2012. Scalable Critical-Path Based Performance Analysis. In *Proc. of the 26th IEEE International Parallel & Distributed Processing Symposium (IPDPS), Shanghai, China*. 1330–1340.

David Böhme, Markus Geimer, Felix Wolf, and Lukas Arnold. 2010. 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, 90–100. DOI:http://dx.doi.org/10.1109/ICPP.2010.18 Best Paper Award.

Maria Calzarossa, Luisa Massari, and Daniele Tessera. 2004. A methodology towards automatic performance analysis of parallel applications. *Parallel Comput.* 30, 2 (Feb. 2004), 211–223. DOI:http://dx.doi.org/10.1016/j.parco.2003.08.002

Todd Gamblin, Bronis R. de Supinski, Martin Schulz, Rob Fowler, and Daniel A. Reed. 2008. Scalable Load-Balance Measurement for SPMD Codes. In *Proc. of the ACM/IEEE Conference on Supercomputing (SC08, Austin, TX)*.

Markus Geimer, Felix Wolf, Brian J. N. Wylie, and Bernd Mohr. 2009. A scalable tool architecture for diagnosing wait states in massively-parallel applications. *Parallel Comput.* 35, 7 (2009), 375–388.

Michael Geissler, S Rykovanov, Jörg Schreiber, Jürgen Meyer ter Vehn, and G D Tsakiris. 2007. 3D simulations of surface harmonic generation with few-cycle laser pulses. *New Journal of Physics* 9, 7 (2007), 218.

Michael Geissler, Jörg Schreiber, and Jürgen Meyer ter Vehn. 2006. Bubble acceleration of electrons with few-cycle laser pulses. *New Journal of Physics* 8, 9 (2006), 186.

John C. Hayes, Michael L. Norman, Robert A. Fiedler, James O. Bordner, Pak Shing Li, Stephen E. Clark, Asif Ud-Doula, and Mordecai-Mark MacLow. 2006. Simulating Radiating and Magnetized Flows in Multi-Dimensions with ZEUS-MP. *Astrophysical Journal Supplement* 165 (2006), 188–228.

Marc-André Hermanns, Manfred Miklosch, David Böhme, and Felix Wolf. 2013. Understanding the formation of wait states in applications with one-sided communication. In *EuroMPI '13: Proc. of the 20th European MPI Users' Group Meeting, Madrid, Spain, September 15–18, 2013*. ACM, New York, NY, USA, 73–78. DOI:http://dx.doi.org/10.1145/2488551.2488569

Adolfy Hoisie, Olaf Lubeck, and Harvey Wasserman. 1999. Performance analysis of wavefront algorithms on very-large scale distributed systems. In *Proceedings of the workshop on wide area networks and high performance computing (Lecture Notes in Control and Information Sciences)*, Vol. 249. Springer Berlin / Heidelberg, 171–187. DOI:http://dx.doi.org/10.1007/BFb0110074

Jeffrey K. Hollingsworth. 1996. An Online Computation of Critical Path Profiling. In *Proceedings of the 1st ACM SIGMETRICS Symposium on Parallel and Distributed Tools*. 11–20.

Hassan M. Jafri. 2007. Measuring Causal Propagation of Overhead of Inefficiencies in Parallel Applications. In *Proc. of the 19th IASTED International Conference on Parallel and Distributed Computing and Systems*. Cambridge, MA, USA, 237–243.

Allen D. Malony, Sameer S. Shende, and Alan Morris. 2005. Phase-Based Parallel Performance Profiling. In *Proc. of the Conference on Parallel Computing (ParCo, Malaga, Spain) (NIC Series)*, Vol. 33. John von Neumann Institute for Computing, 203–210.

Wagner Meira,Jr., Thomas J. LeBlanc, and Virgílio A. F. Almeida. 1998. Using cause-effect analysis to understand the performance of distributed programs. In *Proc. of the SIGMETRICS Symposium on Parallel and Distributed Tools (SPDT'98)*. ACM, New York, NY, USA, 101–111. DOI:http://dx.doi.org/10.1145/281035.281046

Wagner Meira,Jr., Thomas J. LeBlanc, and Alexandros Poulos. 1996. Waiting time analysis and performance visualization in Carnival. In *Proc. of the SIGMETRICS Symposium on Parallel and Distributed Tools (SPDT'96)*. ACM, New York, NY, USA, 1–10. DOI:http://dx.doi.org/10.1145/238020.238023

Oleg Morajko, Anna Morajko, Tomas Margalef, and Emilio Luque. 2008. On-Line Performance Modeling for MPI Applications. In *Proc. of the 14th Euro-Par Conference (Las Palmas de Gran Canaria, Spain) (Lecture Notes in Computer Science)*, Vol. 5168. Springer, 68–77.

Martin Schulz. 2005. Extracting Critical Path Graphs from MPI Applications. In *Proc. of the IEEE Cluster Conference*. Boston, MA, USA.

Martin Schulz, Greg Bronevetsky, and Bronis R. de Supinski. 2008. On the Performance of Transparent MPI Piggyback Messages. In *Proc. 15th European PVM/MPI Users' Group Meeting (Dublin, Ireland) (Lecture Notes in Computer Science)*, Vol. 5205. Springer, 194–201.

David Sundaram-Stukel and Mary K. Vernon. 1999. Predictive analysis of a wavefront application using LogGP. In *Proceedings of the 7th ACM SIGPLAN Symposium on Principles and Practices of Parallel Programming*, Vol. 34. 141–150.

Zoltán Szebenyi, Felix Wolf, and Brian J. N. Wylie. 2009. Space-Efficient Time-Series Call-Path Profiling of Parallel Applications. In *Proc. of the ACM/IEEE Conference on Supercomputing (SC09, Portland, OR)*.

Zoltán Szebenyi, Brian J. N. Wylie, and Felix Wolf. 2008. Scalasca Parallel Performance Analyses of SPEC MPI2007 Applications. In *Proc. of the 1st SPEC Int'l Performance Evaluation Workshop (SIPEW, Darmstadt, Germany) (Lecture Notes in Computer Science)*, Vol. 5119. Springer, 99–123.

Nathan R. Tallent, Laksono Adhianto, and John Mellor-Crummey. 2010. Scalable Identification of Load Imbalance in Parallel Executions using Call Path Profiles. In *Supercomputing 2010*. New Orleans, LA, USA.

University Corporation for Atmospheric Research (UCAR). 2012. The Community Earth System Model. http://www.cesm.ucar.edu/. (Februar 2012).

Jeffrey Vetter (Ed.). 2007. Report of the Workshop on Software Development Tools for Petascale Computing. (August 2007). US Department of Energy, http://www.csm.ornl.gov/workshops/Petascale07/sdtpc_workshop_report.pdf.

Brian J. N. Wylie. 2012. Parallel performance measurement & analysis scaling lessons. SC'12 Workshop on Extreme-Scale Performance Tools (Salt Lake City, UT). (Nov. 2012).

Brian J. N. Wylie, David Böhme, Bernd Mohr, Zoltán Szebenyi, and Felix Wolf. 2010. Performance analysis of Sweep3D on Blue Gene/P with the Scalasca toolset. In *Proc. 24th Int'l Parallel & Distributed Processing Symposium and Workshops (IPDPS, Atlanta, GA)*. IEEE Computer Society.

Received July 2013; revised October 2014; accepted May 2016