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

David Böhme[*][†], Markus Geimer[*], Felix Wolf[*][†][‡], and Lukas Arnold[*]

[*] Jülich Supercomputing Centre, 52425 Jülich, Germany
[†] Graduate School AICES, RWTH Aachen University, 52062 Aachen, Germany
[‡] German Research School for Simulation Sciences, 52062 Aachen, Germany
{d.boehme, m.geimer, l.arnold}@fz-juelich.de, f.wolf@grs-sim.de

Abstract—**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 in forward and backward direction, we can identify the processes and call paths responsible for the most severe imbalances even for runs with tens of thousands of processes.**

I. 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 systems featuring more than a hundred thousand cores, writing efficient codes that exploit all the available parallelism becomes increasingly difficult. Load and communication imbalance, which frequently occurs during simulations of irregular and dynamic domains that are typical of many engineering codes, presents a key challenge to achieving satisfactory scalability. As "the petascale manifestation of Amdahl's Law" [1], 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 the delay, which allows a potentially large temporal distance between the cause and its symptom. Moreover, when complex point-to-point communication patterns are employed, wait states may propagate across process boundaries along far-reaching cause-effect chains that are hard to track manually and that complicate the assessment of the actual costs of 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 include 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 late comer, which has a multiplying effect on the costs. 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 in between constitutes a substantial challenge in deriving helpful conclusions from this knowledge with respect to remediating the wait states.

The Scalasca performance-analysis toolset [2] searches for wait states in large-scale MPI programs by measuring the temporal displacement between matching communication and synchronization operations that have been previously 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 [3]. 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. [4], [5], 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 and calculates the costs of such delays.

This paper is organized as follows. We start with a discussion of related work in Section II. Then we introduce the Scalasca tracing methodology in Section III, providing the context for the scalable delay analysis that we present in Section IV. In Section V, we demonstrate the value of our method using three examples, showing its scalability and illustrating the additional insights it offers into performance behavior. Finally, in Section VI, we present conclusions and outline future work.

II. Related Work

Our approach was inspired by the work of Meira Jr. et al. in the context of the Carnival system [4], [5]. Using traces of program executions, Carnival identifies the differences in execution paths leading up to a synchronization point and explains waiting time to the user in terms of those differences. This waiting-time analysis is implemented as a pipeline of four independent tools: The first stage extracts execution steps (i.e., region instances) from the trace file. These execution steps are combined to matching execution paths for every instance of waiting time during the second stage. The third stage partitions the execution paths into equivalence classes so that for every class only one representative needs to be stored. The fourth stage finally isolates the differences between matching paths, yielding one or more characterizations for each program location that exhibits wait states. While our analysis is very similar on a conceptual level, it offers far greater scalability, allowing it to be applied to codes running on tens of thousands of processor cores. Based on a parallel replay of the underlying traces using as many cores as have been used by the target application itself, our method exploits both distributed memory and processing capabilities available on massively parallel systems. 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? 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 V.

Again leveraging the postmortem analysis of event traces, Jafri [6] 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 waiting time is propagated across large numbers of processes.

Also influenced by Meira Jr. et al., Morajko et al. [7] determine waiting times and their root causes already 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 a global clock to be accurate and (ii) may introduce substantial intrusion [8]. Furthermore, the statistical inference process may prove inaccurate for applications with highly time-dependent performance behavior [9]. Finally, the lack of traces precludes the later simulation of imbalance smoothing to narrow the space of potential optimizations, as proposed by Hermanns et al. [10].

In the context of performance analysis, critical path analysis has been widely studied as an approach to optimally direct optimization efforts. Notably, Hollingsworth [11] and Schulz [12] 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. While critical path analysis effectively pinpoints critical optimization targets and typically requires less space than a full event trace, it lacks the “global view” required to study important performance characteristics such as resource utilization and parallel efficiency since it does not capture effects outside the critical path. In contrast, by incorporating delays of each process and call path, and ranking them according to their severity, our delay analysis provides a global picture of performance hotspots and detailed guidance to improve the overall resource utilization.

Recognizing load imbalance as a major concern for parallel performance, several authors have developed approaches to observe and assess uneven load distributions. Calzarossa et al. [13] rank code regions based on their dispersion across the process space to identify the most promising optimization target. Phase profiling [14] can expose time-varying load 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. [15] 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. [3] use a clustering algorithm to compress time-series call-path profiles online as they are generated. The added value of our approach compared to pure load-data acquisition is to deliver 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

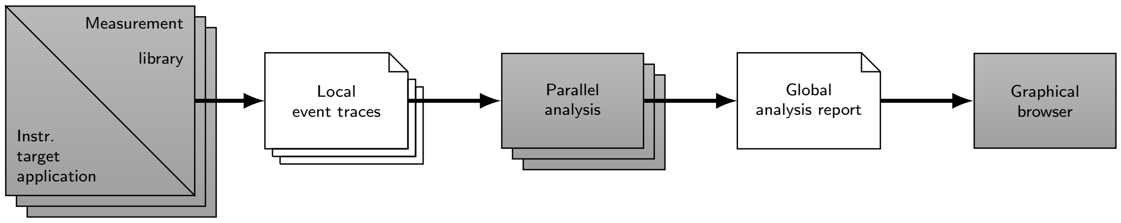


Fig. 1. Scalasca's parallel trace-analysis workflow. Gray rectangles denote programs and white rectangles with the upper right corner turned down denote files. Stacked symbols denote multiple instances of programs or files running or being processed in parallel.

for longer runs, they may serve as a basis for identifying suitable candidate execution intervals for our delay analysis.

III. Tracing Methodology

As the foundation for our subsequent considerations, we start with a review of Scalasca's event-tracing methodology [2]. Scalasca, a performance-analysis toolset specifically designed for large-scale systems, scans event traces of parallel applications for wait states that occur, for example, as the result of an unevenly distributed workload. Such wait states can present major obstacles to achieving good performance, especially when the aim is to scale communication-intensive applications to large processor counts. As a first step towards reducing their impact, the current version of Scalasca provides a diagnostic method that allows the localization and quantification of wait states especially at larger scales. Although this simple wait-state search already supports both pure MPI and hybrid MPI/OpenMP codes, our more advanced delay analysis is currently restricted to single-threaded MPI codes, which is why the remainder of the paper focuses exclusively on this programming model.

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 processor cores as were used to execute the target application itself. Since trace processing capabilities (i.e., processors and memory) grow proportionally with the number of application processes, we were able to complete trace analyses of runs with up to 294,912 cores on a 72-rack IBM Blue Gene/P system [16].

A. 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 appropriate event records, and store them in a memory buffer before they are flushed to disk. Usually, the instrumentation is performed in an automated fashion during compilation and linkage. 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 it is roughly proportional to the frequency of measurement routine invocations, the execution time dilation induced by the instrumentation is highly application-dependent and therefore hard to quantify in general terms.

After the target application has terminated and the trace data has been flushed to disk, the trace analyzer is launched with one analysis process per (target) application process and loads the entire trace data into its distributed memory address space. Future versions of Scalasca may exploit persistent memory segments available on systems such as Blue Gene/P to pass the trace data to the analysis stage without involving any file I/O. While traversing the traces in parallel, the analyzer performs a replay of the application's original communication behavior. 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 an operation of similar type. 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 the provided report explorer. Being the explorer view most relevant to this paper, 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 allow accurate 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 [17] is optionally applied to compensate for drift jitter and other more subtle sources of inaccuracy.

B. 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. In the next section, we will see typical event sequences produced by our event model.

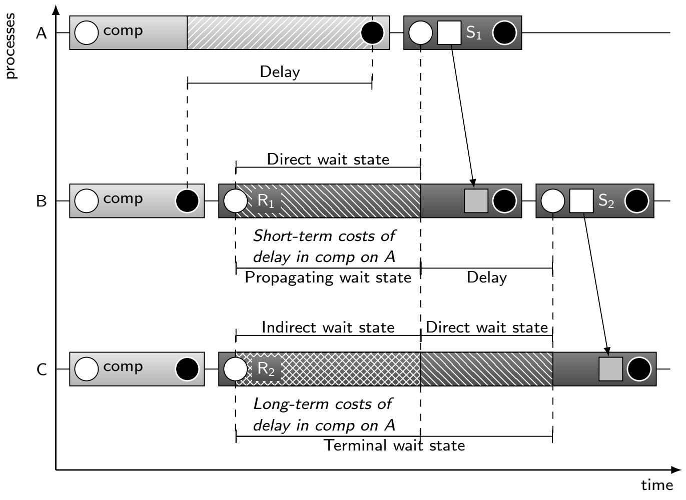


Fig. 2. Time-line diagram showing the activities of three processes and their interactions. The execution of a certain code region is displayed as a shaded rectangle and the exchange of a message as an arrow pointing in the direction of the transfer. Regions labeled S and R represent send and receive 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 R_1 of B. The wait state in B subsequently delays process C. Thus, the total costs of the delay on A correspond to the total amount of wait states caused by it directly (short-term costs) or indirectly (long-term costs).

IV. 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.

A. 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 time-line diagram with a separate time line 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.

a) 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 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 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 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 at 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 follow-up wait states. 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.

b) 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. A delay is not necessarily of computational nature and may also cover communication. For example, the decomposition of irregular domains can easily lead to excess communication when processes have to talk to 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, delay appears on the time line of process A in region comp(). This delay is responsible for the direct wait state in R_1 on process B and for the indirect wait state in R_2 on process C. In addition, some delay is introduced by the actual message receipt in the receive operation R_1.

c) 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 costs define a perspective from the beginning of the causation chain, we believe that the following refinement is most useful: Short-term costs cover the direct wait states, whereas long-term costs cover the indirect wait states. (Due to their established meaning in business administration, we deliberately avoid the terms direct or indirect costs.) The total delay costs are simply the sum of the two. In Section V, 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 for the user. As we will see, 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.

B. Scalable Delay Detection and Cost Accounting

To ensure scalability, the delay analysis follows the same parallelization strategy as the pure wait state analysis does, leveraging the principle of a parallel replay of the communication recorded in the trace. However, other than the pure wait-state analysis, the 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 endmost wait states, this allows delay costs to travel 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 wait state analysis and 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 the classes propagating vs. terminal and direct vs. indirect.

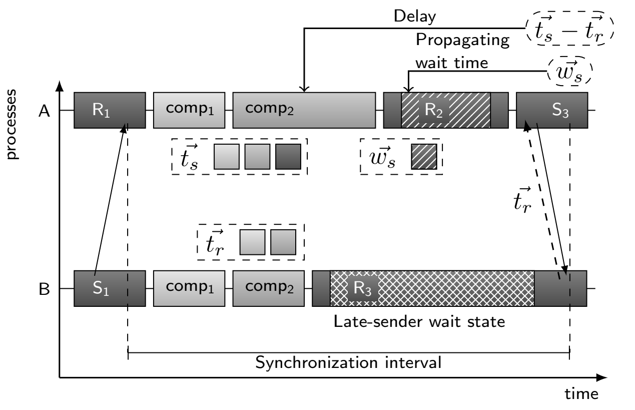


Fig. 3. Delay detection for a late-sender wait state via backward replay. Original messages are shown as solid arrows, whereas messages replayed in reverse direction are shown as dashed arrows.

In the following, we explain our delay analysis in greater detail, for simplicity initially concentrating exclusively on point-to-point communication. During the forward stage, the analysis processes annotate (i) each wait state with the amount of waiting time they measure and (ii) each synchronizing communication event with the rank of the remote process involved. A communication event of synchronizing nature is called a *synchronization point*. The annotations will be needed later to identify the communication events and synchronization intervals where delay occurred.

The actual delay analysis is performed during the backward stage, which traverses all the process-local traces simultaneously in backward direction. Whenever the annotations indicate a wait state identified during the forward stage, the algorithm determines the corresponding synchronization interval, identifies the delays and propagating wait states causing the waiting time, and calculates the short-term and long-term delay costs. A *synchronization interval* covers the time between two consecutive synchronization points of the same two ranks, where runtime differences can cause wait states at the end of the interval. Whereas the communication event associated with the wait state marks the end point of the interval, its beginning is defined by the previous synchronization point involving the same pair of ranks. As the communication operations are reenacted in backward direction in the course of the algorithm's execution, the costs are successively accumulated and transferred back to their source.

Figure 3 illustrates the delay analysis for a late-sender wait state. The example exhibits a delay in region (i.e., call path) instance comp$_2$() and a propagating wait state induced by some influence external to the scene in region instance R$_2$ of the sender (i.e., process A). This causes a late-sender wait state in region instance R$_3$ of the receiver (i.e., process B). To identify the delay during our analysis, both sender and receiver calculate their so-called *time vectors* $\vec{t_s}$ and $\vec{t_r}$ for the synchronization interval in the figure, with each vector element representing the accumulated time spent in a given call path between the two synchronization points – the communication operations at the end of the interval excluded. In addition, the sender also determines its waiting-

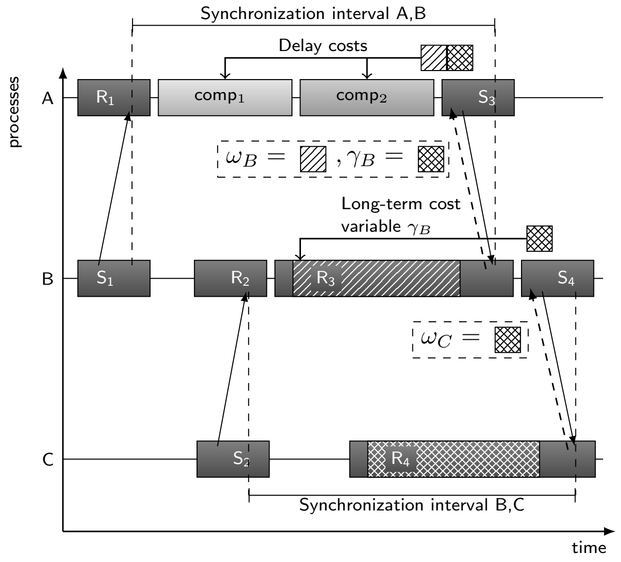

Synchronization interval A,B
Delay costs
processes
A
B
C
R1
comp1
comp2
S3
$\omega_B =$ ▨ $, \gamma_B =$ ▦
Long-term cost variable γ_B
S1
R2
R3
S4
$\omega_C =$ ▦
S2
R4
Synchronization interval B,C
time

Fig. 4. Source-related accounting of long-term costs via backward replay and successive accumulation of indirectly induced waiting time. The waiting time ω_C fist travels to its immediate cause, wait state R_3 on process B, from where it is propagated further to its ultimate cause, the delay on process A.

time vector $\vec{w_s}$, which contains the amount of waiting time in each (communication) call path visited during the interval. This is necessary to distinguish delay from propagating wait states. The receiver sends its time vector $\vec{t_r}$ via the reversed communication (dashed arrow) to the sender, which calculates the difference vector $\vec{d} = \vec{t_s} - \vec{w_s} - \vec{t_r}$. For every call path, the difference vector contains the differences in execution time (excluding waiting time) between sender and receiver. Under certain circumstances, some elements of the difference vector may be negative. This can happen if 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.

In the next step, the algorithm determines the short- and long-term costs of the detected delay and maps them onto the (call path, process) tuples where the delay occurred. The short-term costs simply correspond to the amount of direct waiting time incurred by process B in R_3. The amount of direct waiting is obtained by dividing the overall waiting time in R_3, which is transferred to process A during the backward replay, into direct and indirect waiting time at the ratio of the amount of delay in $\vec{d}$ versus the amount of waiting time in $\vec{w_s}$, respectively. The short-term costs are then mapped onto the delaying call paths by distributing the amount of direct waiting time proportionally across all call paths involved in the delay. Likewise, propagating waiting time is mapped onto the call paths suffering wait states in the synchronization interval on process A by proportionally distributing the amount of indirect waiting time across the call paths in $\vec{w_s}$.

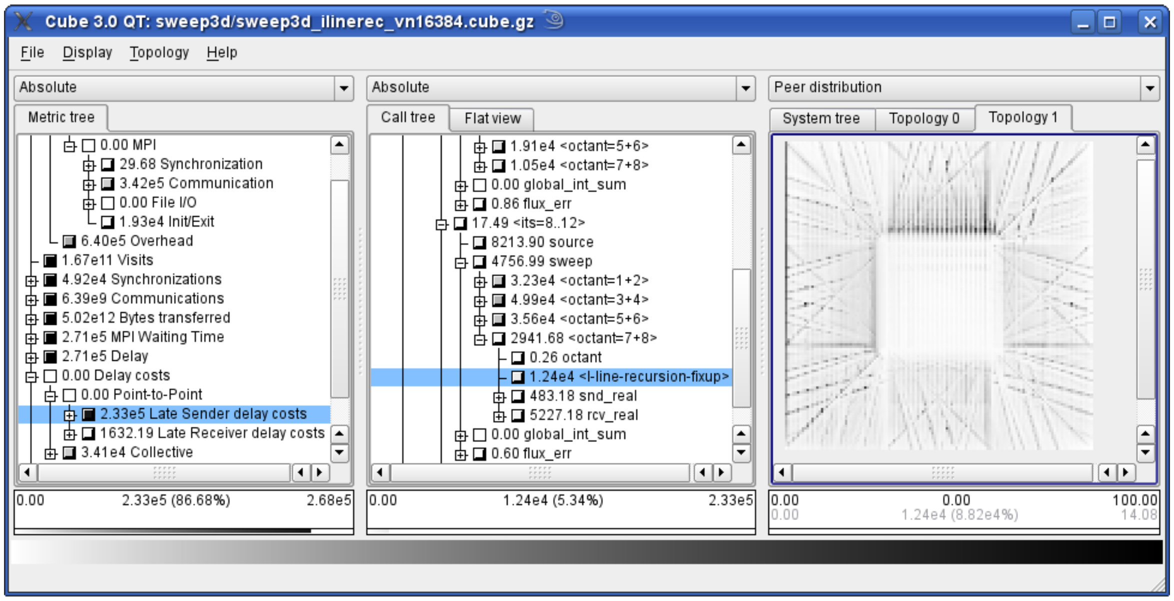
To calculate the long-term costs of the detected delay, we need to know the total amount of waiting time that was indirectly caused via propagation. Therefore, communication events where waiting time was detected are further annotated with a long-term cost variable γ, which represents the costs indirectly caused by this wait state later on. These cost variables are initialized with zero and updated in the course of the backward replay. The long-term costs are propagated backwards by transmitting the cost variable of a wait state back to the delaying process, where it is used to calculate long-term delay costs and to update the cost variables of wait states present in the synchronization interval. In this way, the delay costs are successively accumulated as they travel backward through the communication chain until they reach their root cause(s). Hence, we can accurately incorporate distant effects into the calculation of the overall delay costs in a highly scalable manner.

The more complex example in Figure 4 illustrates the data flow necessary to accomplish the source-related accounting of long-term costs. Here, delays in region instances $\text{comp}_1()$ and $\text{comp}_2()$ on process A cause a wait state in R_3 on process B, which in turn delays communication with process C, resulting in another wait state in R_4 on C. The backward replay starts at the wait state in R_4 on process C. The waiting time ω_C of this wait state is transmitted to process B via reverse communication. There, the long-term cost variable γ_B of the wait state in R_3 in synchronization interval B,C is updated to account for the amount of waiting time caused by its propagation. Next, both R_3's waiting time ω_B and the cost variable γ_B are transferred to A, where they are mapped onto the initial delay in $\text{comp}_1()$ and $\text{comp}_2()$ in synchronization interval A,B. The waiting time ω_B represents the short-term costs, and the cost variable γ_B represents the long-term costs.

The general principle of our backward-replay based accounting method also applies to collective operations, but with some subtle differences. For n-to-1 and n-to-n communication and synchronization operations, such as barrier, all-to-all or (all)gather/(all)reduce, delay costs are assigned to the last process that enters the operation. For 1-to-n communications (broadcasts), delay costs are assigned to the root process of the operation. In contrast to the point-to-point case, the time vector of the delaying process is broadcast to all processes participating in the operation. Now, every process determines the delaying call paths and calculates the delay costs for the amount of waiting time occurring locally on that process. The individual cost contributions are then accumulated in a reduction operation and finally assigned to the delaying process. For portability reasons, we do not make assumptions about the underlying implementation of collective operations. Therefore, our performance patterns for collective communication remain as generic as possible and do not consider wait states which might occur, for example, on intermediate processes in a software broadcast.

V. 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 [18], the astrophysics simulation Zeus-MP/2 [19], and the plasma-physics code Illumination [20], [21]. All



```
Cube 3.0 QT: sweep3d/sweep3d_ilinerec_vn16384.cube.gz
File  Display  Topology  Help

Absolute                          Absolute                          Peer distribution
Metric tree                       Call tree | Flat view             System tree | Topology 0 | Topology 1

      0.00 MPI                              1.91e4 <octant=5+6>
        29.68 Synchronization               1.05e4 <octant=7+8>
        3.42e5 Communication              0.00 global_int_sum
        0.00 File I/O                     0.86 flux_err
        1.93e4 Init/Exit                17.49 <its=8..12>
    6.40e5 Overhead                       8213.90 source
  1.67e11 Visits                          4756.99 sweep
  4.92e4 Synchronizations                   3.23e4 <octant=1+2>
  6.39e9 Communications                     4.99e4 <octant=3+4>
  5.02e12 Bytes transferred                 3.56e4 <octant=5+6>
  2.71e5 MPI Waiting Time                   2941.68 <octant=7+8>
  2.71e5 Delay                                0.26 octant
  0.00 Delay costs                            1.24e4 <l-line-recursion-fixup>
    0.00 Point-to-Point                       483.18 snd_real
      2.33e5 Late Sender delay costs          5227.18 rcv_real
      1632.19 Late Receiver delay costs   0.00 global_int_sum
    3.41e4 Collective                     0.60 flux_err

0.00   2.33e5 (86.68%)   2.68e5   0.00   1.24e4 (5.34%)   2.33e5   0.00   0.00                100.00
                                                                   0.00   1.24e4 (8.82e4%)    14.08
```

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).

measurements were taken on the 72-rack IBM Blue Gene/P supercomputer Jugene at the Jülich Supercomputing Centre.

A. Scalability

Because 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 processes, and reflective boundary conditions. A wavefront process is employed in the i and j directions,

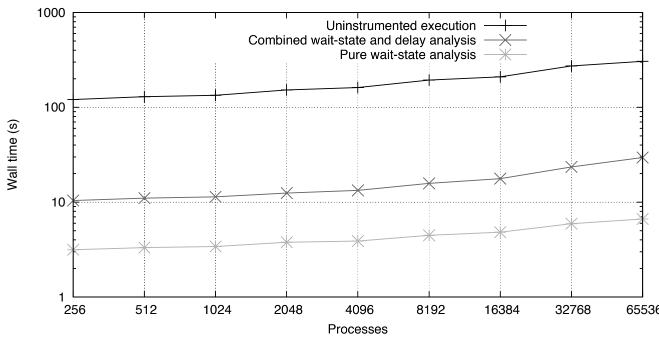


Fig. 6. Comparison of Sweep3D application execution time, combined wait-state and delay analysis time, and pure wait-state analysis time at various scales.

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 65,536 processes, configured in weak scaling mode with a constant problem size of $32 \times 32 \times 512$ cells per process. 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 6 compares the wall-clock execution times of the combined wait-state and delay analysis with (i) the uninstumented Sweep3D application and (ii) the pure wait-state analysis. The 8-fold doubling in the number of processes and the resulting large range of times necessitates a log-log scale in the plot. Since it is not subject of this study, the analysis times do not include loading the traces, which took roughly 110 s at the largest scale. Although the combined analysis needed appreciably more time than the pure wait-state analysis, it scaled equally well. As expected when replaying the original communication, both curves run in parallel to the uninstrumented execution. We believe that the increase in trace-analysis time can be tolerated even for configurations larger than 65,536 – justified by the improved understanding of wait-state formation, as demonstrated below.

B. 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.

1) Sweep3D: This benchmark has been comprehensively modeled and examined on a variety of systems and scales [22], [23], and was also the subject of a recent scaling study [16] on Jugene using Scalasca but without our delay analysis. While the study generally confirmed the good scaling behavior

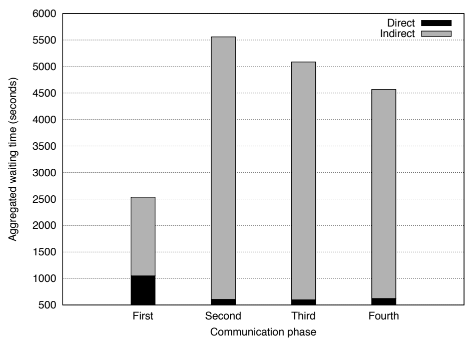


Fig. 7. Composition of waiting time in different communication phases of Zeus-MP/2.

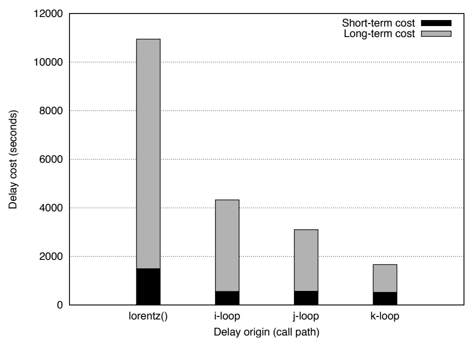


Fig. 8. Short- and long-term costs of delays in four critical call paths of Zeus-MP/2.

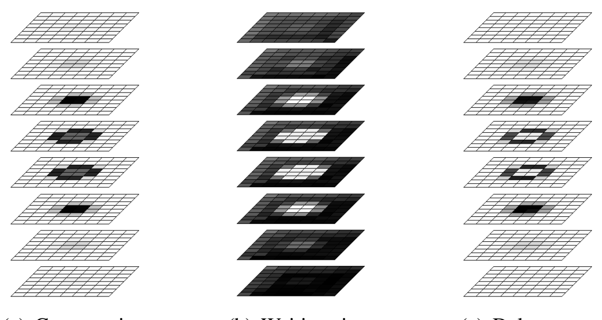
of Sweep3D, it also identified computational load imbalance and MPI waiting time growing with scale. By successively refining the analysis with additional source code annotations, the section of code applying 'fixup' corrections was pinpointed as the source of the load imbalance. To assess the influence 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.

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 routines as primary causes of wait states. Overall, these 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). These 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 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.

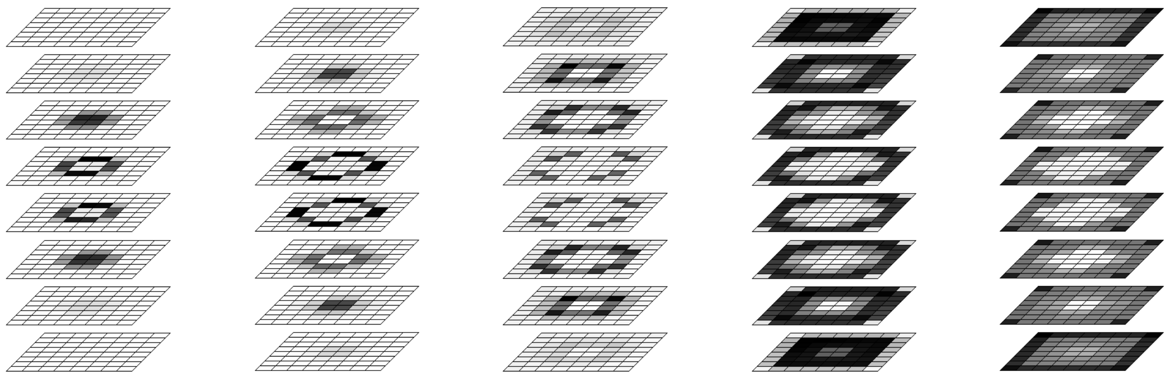
2) Zeus-MP/2: Our second case study is the astrophysical application Zeus-MP/2. The Zeus-MP/2 code performs hydrodynamic, radiation-hydrodynamic (RHD), and magnetohydrodynamics (MHD) simulations on 1, 2, or 3-dimensional grids. For parallelization, Zeus-MP/2 decomposes the computational domain regularly 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 computational domain. We analyzed the 2.1.2 version on 512 processes simulating a three-dimensional magnetohydrodynamics wave blast, based on the "mhdblast_XYZ" example configuration provided with the distribution. The number of simulated time steps was limited to 100 in order to constrain the size of the recorded event trace. As in the previous example, our analysis targets the origins of wait states.

The wave-blast simulation requires 188,000 seconds of CPU 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 within each iteration of the main loop, in the following denoted as first to fourth communication phase. As Figure 7 shows, the dominant 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,



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

Fig. 9. Distribution of computation time, waiting time, and total delay costs in Zeus-MP/2 across the three-dimensional computational domain.



(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.

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. 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 (Figure 9(b)).

Examining the delay costs reveals that almost all the delay originates from the border processes of the central, overloaded region (Figure 9(c)). The distribution of the workload explains this observation: Within the central and outer regions, the workload is relatively well balanced. Therefore, communication within the same region is not significantly delayed. In contrast, the large difference in computation time between the central and outer region causes wait states at synchronization points along the border.

Our findings 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 phase.

3) Illumination: Our last case study is Illumination, a 3D parallel relativistic particle-in-cell code for the simulation of laser-plasma interactions [20], [21], 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

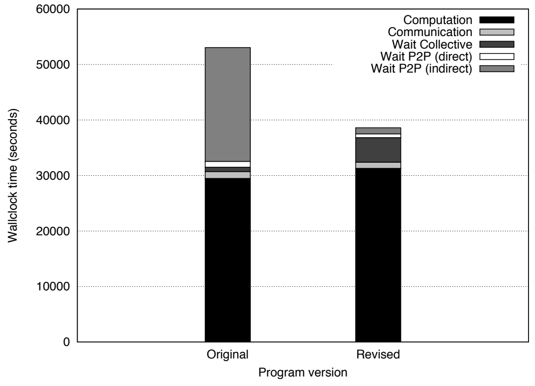
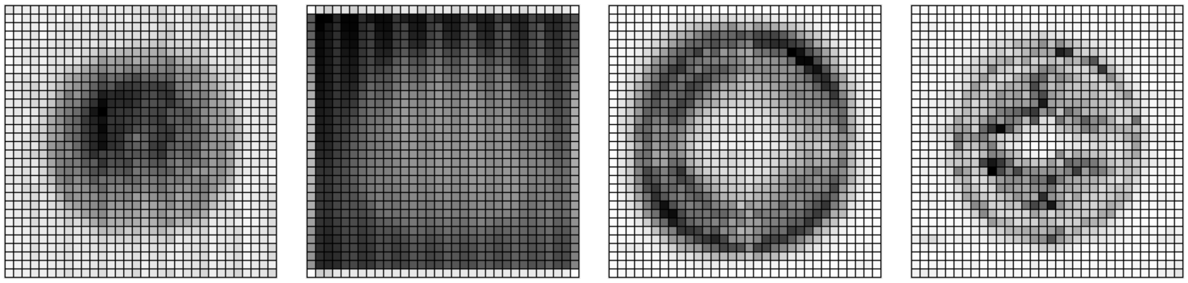


Fig. 11. Runtime composition and quantitative comparison of the original versus the revised version of Illumination. In the revised version, the indirect waiting time was significantly reduced and wait states were partially shifted from point-to-point to collective communication. A slight increase in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



(a) Computation (original) (b) Propagating wait states (original) (c) Propagating wait states (revised) (d) Total delay costs (revised)

Fig. 12. Comparing the original with the revised (i.e., optimized) version of Illumination by visually mapping computational load, propagating wait states, and delay costs onto the two-dimensional virtual process topology.

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 direct influence of the overloaded region, as indicated by its short-term delay costs, was negligible, whereas the dominant portion of waiting time was caused by propagation. Accordingly, a large number of the late-receiver wait states propagate themselves further. Figure 12(b) illustrates that with the exception of the border processes, a significant amount of propagating waiting time is present across all processes of the two-dimensional grid. Only a blurred resemblance of the inverted load-distribution pattern recalls the remote influence exerted by the computational imbalance. These findings suggest that the main problem was actually an inefficient communication pattern as such: in a sense, the communication impeded itself.

When exchanging 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 seems now 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, however, remains. Still, 62% of the waiting time is the result of propagation, but mapping the propagating wait states onto the responsible processes (Figure 12(c)) makes the load imbalance appear to be a less remote and more direct cause.

We can validate this assumption by clearly identifying delay within the computational part of the main loop as the major source of the waiting time. Also, the topological distribution of the delay costs across the process grid, as depicted in Figure 12(d), roughly delineates regions of equal load like level lines on a geographical map – similar to our previous case study. 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. This, however, would require a major redesign of the code, which is beyond the scope of this paper.

VI. 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 wait states they cause – even if those wait states materialize much later in the program. This cost attribution is essential, since the resulting wait states may consume much more resources than the delaying operation itself. Compared to earlier work, our approach is based on a parallel replay of event traces both in forward and in backward direction, which allowed non-trivial insights into the wait-state propagation occurring in three example codes running on up to 65,536 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. However, redistributing workloads in complex message-passing applications can have intricate side-effects that may compromise the expected reduction of waiting times. Given that balancing the load statically or even introducing a dynamic load-balancing scheme constitute major code changes, they should ideally be performed only if the prospective performance gain is likely to materialize. It would therefore be desirable if the effects of redistributing a given delay could be automatically predicted and the expected savings be determined without altering the application itself. Since the effects of such changes are hard to quantify analytically, we plan to combine our delay analysis with a framework developed earlier by the authors [10], [24] that can simulate these changes via a real-time replay of event traces after they have been modified to reflect the redistributed load. First results indicate both high accuracy and good scalability, further application studies are in progress.

Acknowledgment

Financial support from the Deutsche Forschungsgemeinschaft (German Research Foundation) through Grant GSC 111 and the Helmholtz Association of German Research Centers through Grant VH-NG-118 is gratefully acknowledged.

References

[1] J. Vetter (Ed.), "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.

[2] M. Geimer, F. Wolf, B. J. N. Wylie, and B. Mohr, "A scalable tool architecture for diagnosing wait states in massively-parallel applications," *Parallel Computing*, vol. 35, no. 7, pp. 375–388, 2009.

[3] Z. Szebenyi, F. Wolf, and B. J. N. Wylie, "Space-efficient time-series call-path profiling of parallel applications," in *Proc. of the ACM/IEEE Conference on Supercomputing (SC09, Portland, OR)*, November 2009.

[4] W. Meira,Jr., T. J. LeBlanc, and A. Poulos, "Waiting time analysis and performance visualization in Carnival," in *Proc. of the SIGMETRICS Symposium on Parallel and Distributed Tools (SPDT'96)*. New York, NY, USA: ACM, 1996, pp. 1–10.

[5] 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)*. New York, NY, USA: ACM, 1998, pp. 101–111.

[6] H. M. Jafri, "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, November 2007, pp. 237–243.

[7] O. Morajko, A. Morajko, T. Margalef, and E. Luque, "On-line performance modeling for MPI applications," in *Proc. of the 14th Euro-Par Conference (Las Palmas de Gran Canaria, Spain)*, ser. Lecture Notes in Computer Science, vol. 5168. Springer, August - September 2008, pp. 68–77.

[8] M. Schulz, G. Bronevetsky, and B. R. de Supinski, "On the performance of transparent MPI piggyback messages," in *Proc. 15th European PVM/MPI Users' Group Meeting (Dublin, Ireland)*, ser. Lecture Notes in Computer Science, vol. 5205. Springer, September 2008, pp. 194–201.

[9] Z. Szebenyi, B. J. N. Wylie, and F. Wolf, "Scalasca parallel performance analyses of SPEC MPI2007 applications," in *Proc. of the 1st SPEC Int'l Performance Evaluation Workshop (SIPEW, Darmstadt, Germany)*, ser. Lecture Notes in Computer Science, vol. 5119. Springer, June 2008, pp. 99–123.

[10] M.-A. Hermanns, M. Geimer, F. Wolf, and B. J. Wylie, "Verifying causality between distant performance phenomena in large-scale MPI applications," in *Proc. of the 17th Euromicro International Conference on Parallel, Distributed, and Network-Based Processing (PDP, Weimar, Germany)*. IEEE Computer Society, February 2009, pp. 78–84.

[11] J. K. Hollingsworth, "An online computation of critical path profiling," in *Proceedings of the 1st ACM SIGMETRICS Symposium on Parallel and Distributed Tools*, May 1996, pp. 11–20.

[12] M. Schulz, "Extracting critical path graphs from MPI applications," in *Proc. of the IEEE Cluster Conference*, Boston, MA, USA, September 2005.

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

[14] A. D. Malony, S. S. Shende, and A. Morris, "Phase-based parallel performance profiling," in *Proc. of the Conference on Parallel Computing (ParCo, Malaga, Spain)*, ser. NIC Series, vol. 33. John von Neumann Institute for Computing, September 2005, pp. 203–210.

[15] T. Gamblin, B. R. de Supinski, M. Schulz, R. Fowler, and D. A. Reed, "Scalable load-balance measurement for SPMD codes," in *Proc. of the ACM/IEEE Conference on Supercomputing (SC08, Austin, TX)*, November 2008.

[16] B. J. N. Wylie, D. Böhme, B. Mohr, Z. Szebenyi, and F. Wolf, "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, April 2010.

[17] D. Becker, R. Rabenseifner, F. Wolf, and J. C. Linford, "Scalable timestamp synchronization for event traces of message-passing applications," *Parallel Computing*, vol. 35, no. 12, pp. 595–607, 2009.

[18] Accelerated Strategic Computing Initiative, "The ASCI SWEEP3D benchmark code," http://www.ccs3.lanl.gov/pal/software/sweep3d/sweep3d_readme.html, 1995.

[19] J. C. Hayes, M. L. Norman, R. A. Fiedler, J. O. Bordner, P. S. Li, S. E. Clark, A. Ud-Doula, and M.-M. MacLow, "Simulating radiating and magnetized flows in multi-dimensions with ZEUS-MP," *Astrophysical Journal Supplement*, vol. 165, pp. 188–228, 2006.

[20] M. Geissler, J. Schreiber, and J. M. ter Vehn, "Bubble acceleration of electrons with few-cycle laser pulses," *New Journal of Physics*, vol. 8, no. 9, p. 186, 2006.

[21] M. Geissler, S. Rykovanov, J. Schreiber, J. M. ter Vehn, and G. D. Tsakiris, "3D simulations of surface harmonic generation with few-cycle laser pulses," *New Journal of Physics*, vol. 9, no. 7, p. 218, 2007.

[22] A. Hoisie, O. Lubeck, and H. Wasserman, "Performance analysis of wavefront algorithms on very-large scale distributed systems," in *Proceedings of the workshop on wide area networks and high performance computing*, ser. Lecture Notes in Control and Information Sciences, vol. 249. Springer Berlin / Heidelberg, 1999, pp. 171–187.

[23] D. Sundaram-Stukel and M. K. Vernon, "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, no. 8, August 1999, pp. 141–150.

[24] D. Böhme, M.-A. Hermanns, M. Geimer, and F. Wolf, "Performance simulation of non-blocking communication in message-passing applications," in *Proc. of the 2nd Workshop on Productivity and Performance (PROPER, in conjunction with Euro-Par 2009)*, August 2009, (to appear).