Aachen Institute for Advanced Study in Computational Engineering Science

Preprint: AICES-2010/01-1

12/January/2010

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

D. Böhme, M. Geimer, M.-A. Hermanns and F. Wolf

RWTH AACHEN UNIVERSITY

AICES

Financial support from the Deutsche Forschungsgemeinschaft (German Research Association) through grant GSC 111 is gratefully acknowledged.

List of AICES technical reports: http://www.aices.rwth-aachen.de/preprints

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

David Böhme*†, Markus Geimer*, Marc-André Hermanns*, and Felix Wolf*†

* Jülich Supercomputing Centre,
Forschungszentrum Jülich, 52425 Jülich, Germany
† Aachen Institute for Advanced Study in Computational Engineering Science,
RWTH Aachen University, 52056 Aachen, Germany
{d.boehme, m.geimer, m.a.hermanns, f.wolf}@fz-juelich.de

Abstract

Driven by growing application requirements and accelerated by current trends in microprocessor design, the number of processor cores on modern supercomputers increases 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 trace back and that complicate assessing the actual cost 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 cost in terms of resource waste to their original cause. By replaying event traces in parallel, we can identify the processes and call paths responsible for the most severe imbalances even for runs with very large numbers of processes. In addition, we give an outlook on how our method can be extended to automatically determine the benefits of rectifying a previously identified imbalance.

I. Introduction

Driven by growing application requirements and accelerated by current trends in microprocessor design, the number of processor cores on modern supercomputers increases 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. Occurring often during simulations of irregular and dynamic domains, which are typical for many engineering codes, load and communication imbalance presents a key challenge to achieving satisfactory scalability. As "the petascale manifestation of Amdahl's Law" [1], already delays of single processes may spread wait states across the entire machine, whose 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 employing complex point-to-point communication patterns, wait states may propagate across process boundaries along far-reaching cause-effect chains that are hard to trace back manually and that complicate assessing the actual cost 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 a process performs some additional activity not performed by its peers. 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 cost of a delay manifests itself 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 on 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 cost 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 their remediation.

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. 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. [3], [4], we describe how Scalasca's scalable wait-state analysis was extended to also identify the responsible delays and their cost. Now, users can see at the first glance where load balancing should be improved to reduce the waiting time in their programs most effectively. To this end, our paper makes the following specific contributions:

- A terminology that can be used 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 their cost

To underline the full potential of our approach, we also describe how to determine the potential execution time savings from rectifying a previously identified delay via a trace-based real-time simulation. This is a non-trivial extension of the pure delay analysis, as in most cases a delaying operation cannot be simply removed but must be redistributed to other processes instead – potentially with complex side effects that are hard to quantify analytically. With this extension, however, it would become possible to automatically evaluate larger numbers of candidate optimization hypotheses before investing time in expensive code changes.

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 evaluate the value of our method using two examples, demonstrating its scalability and illustrating the additional insights it offers into the performance behavior. As a logical extension of the delay analysis, Section VI describes how the effects of rectifying a previously identified delay can be automatically determined. Finally, in Section VII, we conclude the paper 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 [3], [4]. 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. Besides minor distinctions with respect to the cost model and the way of presenting delays to the user, we perform a very similar analysis – but in a much more scalable manner. Based on a parallel replay of the underlying traces, our method is suitable for application runs with several thousand processes. Moreover, by simulating the effects of rectifying a given delay our approach comes much closer to deriving promising optimization hypotheses. Again leveraging the postmortem analysis of event traces, Jafri [5] 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 ones via propagation. However, neither does his analysis identify the responsible delays, as ours does, nor does his sequential implementation address scalability on large systems.

Also influenced by Meira Jr. et al., Morajko et al. [6] 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 in 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 [7]. Furthermore, the statistical inference process may prove inaccurate for applications with highly time-dependent performance behavior [8]. Finally, the lack of traces precludes the later simulation of imbalance smoothing to narrow the space of potential optimizations, as proposed in this paper.

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. [9] rank code regions based on their dispersion across the process space to identify the most promising optimization target. Phase profiling [10] 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. [11] 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. [12] 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 cost 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, not requiring detailed event traces that are too costly to generate for longer runs, the abovementioned profiling techniques may serve as a basis to identify suitable candidate execution intervals for our delay analysis.

The principle of trace-driven performance prediction underlying our method to evaluate the effects of evening out a given imbalance has already been extensively studied. Several approaches [13], [14], [15] address questions about performance implications when varying architectural parameters, such as the number or speed of the processors or the latency and bandwidth of the network, and to a lesser extent also when introducing synthetic perturbations [16] that reflect modified application-level behavior. Our work clearly concentrates on the effects of fine-grained alterations of application-level behavior, such as the balancing of individual functions, with respect to the performance under an identical execution configuration. The most important methodological difference is the use of a parallel real-time replay of the simulated communication at the original scale, which offers scalability advantages and eliminates the need to model the extremely complex communication infrastructures found on today's large-scale machines. Compared to more lightweight statistical approaches to predict the performance under larger processor configurations such as [17], our method offers very fine-grained insights at the level of individual call paths into scalability limitations under the current configuration.

III. Tracing Methodology

As the foundation for our subsequent elaborations, 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 challenges to achieving good performance, especially when trying 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 their localization and quantification 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

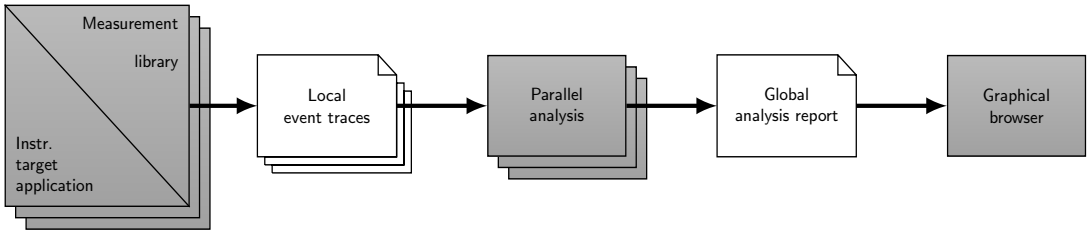


Fig. 1. Scalasca's parallel trace-analysis workflow. Gray rectangles denote programs and white rectangles with the upper right corner turned down denote files. Stacked symbols denote multiple instances of programs or files running or being processed in parallel.

in parallel by *replaying* the original communication on as many processor cores as have been 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 can achieve good scalability at very large scales. So far, we were able to complete trace analyses of runs with up to 294,912 cores on a full IBM Blue Gene/P system.

A. Trace-Analysis Workflow

The Scalasca trace-analysis workflow is illustrated in Figure 1. Before any events can be collected, the target application is instrumented, that is, extra code is inserted to intercept the 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, avoiding that the size of the available trace buffer is 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 timesteps 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 timestep loop) that have been previously identified based on 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 have 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 system 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 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, 8, 9, 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 [18] 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 and the trace-based simulation described in Section VI. In the next section, we will see typical event sequences produced by our event model.

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 cost 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 the 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 marked in red. In both cases, the waiting occurs because they try to receive a message that has not been sent yet, a situation commonly referred to as *late sender*. 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 of process B is a direct wait state because it was caused by excess computation of process A in function foo(). However, by inducing a wait state in process B, this excess computation is indirectly responsible for a wait state of process C, which is why we call this 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 message receipt at B delays the dispatch of the message to C.

b) Delay: A delay is the counterpart 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 analogy to our taxonomy of wait states, we can distinguish between direct and indirect delays. Again, both types appear in Figure 2: First, there is a *direct* delay caused by process A in function foo() (hatched area) that is responsible for the wait state of process B. Second, the wait state of process B is at the same

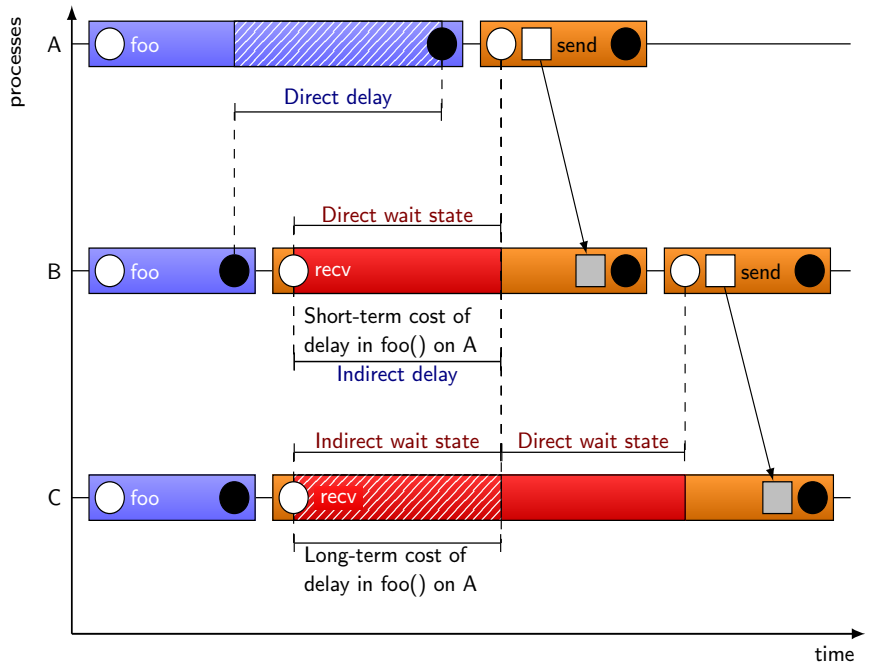


Fig. 2. Time-line diagram showing the activities of three processes and their interactions. The execution of a certain code region is displayed as a colored rectangle and the exchange of a message as an arrow pointing in the direction of the transfer. Events recorded in the trace are symbolized as small circles (enter and exit) or squares (send and receive). Process A delays process B due to an imbalance in function foo(), inducing a wait state in the receive operation of B. The wait state in B subsequently delays process C. Thus, the total cost of the delay corresponds to the total amount of wait states caused by it directly (short-term costs) or indirectly (long-term costs).

time the *indirect* delay partially responsible for the wait state of process C. Note that a delay must not necessarily be of computational nature. For example, the decomposition of irregular domains can easily lead to excess communication when processes have to talk to strongly varying numbers of neighbors.

c) Cost: The cost of a delay is the amount of wait states it causes. The short-term costs cover the direct wait states, whereas the long-term costs cover the indirect wait states. Due to their established meaning in business administration, we deliberately avoid the terms direct or indirect cost.

Of course, trying to determine the exact delineation of the delaying intervals does not necessarily yield a unique solution. For example, it is not clear why the delay of process A should be positioned at the end of function foo() and not at its beginning. On the other hand, such precise knowledge of the delay interval is rarely needed to form a useful optimization hypothesis. The key is to find a suitable characterization of the execution differences between the delaying and the waiting process from the moment the wait state materializes back to their last synchronization point. Since in our result representation all waiting times will be accumulated on a per-call-path basis anyway, we ignore the exact distribution over time and form two *time vectors* (one for each process) with each element representing the accumulated time spent in a given call path between the two synchronization points. Then, we subtract the time vector of the waiting process from the one of the delaying process, yielding the *delay vector*. The delay vector represents an approximation of the call paths of the delaying process that influence the formation of wait states elsewhere, with the times indicating their likely contribution. Although the element sum of the delay vector must be positive, under certain circumstances some of its elements may still be negative. This can happen if a small excess load of the waiting process is overridden by a larger excess load of the delaying process. In this case, we apply the following heuristic: Since a delay cannot be negative, we set the negative elements of the delay vector to zero and scale the remaining positive ones proportionally so that the element sum, which corresponds to the total delay length, remains unchanged. This is sufficient, since we are mainly interested in identifying the call paths primarily responsible for the delay.

To identify the delay whose remediation will yield the highest execution-time benefit, we need to know the amount of wait states it causes, including both the direct and indirect ones. In Section V, we will

see that the long-term costs can be much higher than the short-term costs, a distinction that our delay analysis facilitates. 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, our delay analysis is based on the parallel trace-replay technique described in Section III, extending the simple parallel wait-state search. Every analysis process traverses its local event trace while exchanging information with its peers at synchronization points according to the communication pattern to be replayed. However, whereas the simple wait-state search requires only one replay in forward direction from beginning to end, the delay analysis requires an additional backward replay run. The backward replay runs from the end to the beginning of the trace and reverses the roles of senders and receivers, so that data pertaining to a receive event can be transferred to the corresponding send event.

During the forward phase, each analysis process annotates its local trace with detected wait states, that is, for every event that marks the beginning of a wait state we store the corresponding amount of waiting time. During the backward phase, which starts at the endmost wait states, we pass waiting times and accumulated delay costs backward along the communication chain to the locations of direct delays, where they are finally accounted as the costs of these delays. The direct waiting time induced by a delay is booked as its short-term cost, while the accumulated delay cost without the short-term cost is booked as its long-term cost. The total cost is simply the sum of the two.

Note that the current version ignores wait states induced at the sender by the receiver, such as late receiver, which can occur if a rendezvous protocol is employed. To keep the memory footprint of the trace annotations small, the trace is divided into phases delimited by globally synchronizing collective communication or barrier operations if available. The analysis is then carried out separately for each phase. Once a phase has been finished the annotations are cleared. In the following, we outline the backward cost-accounting algorithm in greater detail, distinguishing between point-to-point and collective communication:

a) Point-to-point communication: Whenever an analysis process reaches a send or receive event during the backward replay, it creates the time vector mentioned earlier, which contains the accumulated time per call path since the last synchronization point with its peer. During the reverse communication reenactment, we transfer the following pieces of information back from the receiver (effect) to the sender (cause):

- The amount of the wait state induced at the receiver during the communication operation being replayed
- The accumulated delay cost of this wait state (i.e., the amount of waiting time subsequently induced by this wait state). Note that this is zero for the endmost wait states
- The time vector $\vec{t_r}$ of the receiver, which is needed by the sender to calculate the delay vector

In the meantime, the sender calculates its own time vector $\vec{t_s}$ plus a wait-state vector $\vec{w_s}$, which contains the accumulated waiting time per call path since the last synchronization point. The wait-state vector is necessary to isolate the direct delay, which we are ultimately interested in. The (direct) delay vector $\vec{d}$ is then initially calculated as:

$$\vec{d} = \vec{t_s} - \vec{w_s} - \vec{t_r}$$

If the delay vector contains negative elements, we set them to zero and apply the scaling factor according to the heuristic described earlier. Both cost metrics being transferred, the amount of the remote wait state and the cost of this wait state, are then proportionally divided into two portions depending on how much of this cost was caused by direct or indirect delay. The portion corresponding to the direct delay is then shared among the call paths the delay vector has non-zero elements for according to the size of these elements. The portion corresponding to the indirect delay is shared among the wait state instances expressed in the event annotations, again according to their size.

b) Collective communication: The cost accounting for collective communication works in a similar way. In broadcasts and globally synchronizing operations such as a barrier or an n-to-n data exchange, the delay is attributed to the root process or the last process arriving at the collective call, respectively. In contrast, waiting times are attributed to all remaining processes. For n-to-1 operations, waiting time is only attributed to the root process. When a collective communication event is reached, the delay-inducing process forwards the individual local call-path time vectors as well as the accumulated wait time since the previous synchronization to each of the waiting processes. There, the short- and long-term delay costs are calculated as in the point-to-point case. The results are aggregated using a reduction operation and stored at the delay-inducing process. Another reduction operation aggregates the scaling factors of each process, which are necessary to update the delay costs attributed to the individual wait-state instances of the delaying process.

V. Evaluation

To evaluate our delay-analysis approach, we conducted a number of experiments using two different MPI codes – the ASCI benchmark Sweep3D [19] and the astrophysics simulation Zeus-MP/2 [20]. The scalability of the delay analysis was demonstrated using Sweep3D, whereas its functional capabilities were studied using both codes. All measurements were taken on the 72-rack IBM Blue Gene/P supercomputer Jugene at the Jülich Supercomputing Centre.

The measurement dilation was minimal. In fact, the instrumented version was even slightly faster than the uninstrumented one, an effect we are still investigating. All experiments were taken using a fixed-size trace buffer of 10 MB per process, which was sufficient to capture trace data for the entire run.

A. Sweep3D

Sweep3D is an MPI benchmark code 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. For domain decomposition, Sweep3D maps the (i, j) planes of the three-dimensional domain onto a two-dimensional grid of processes.

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. Figure 3 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.

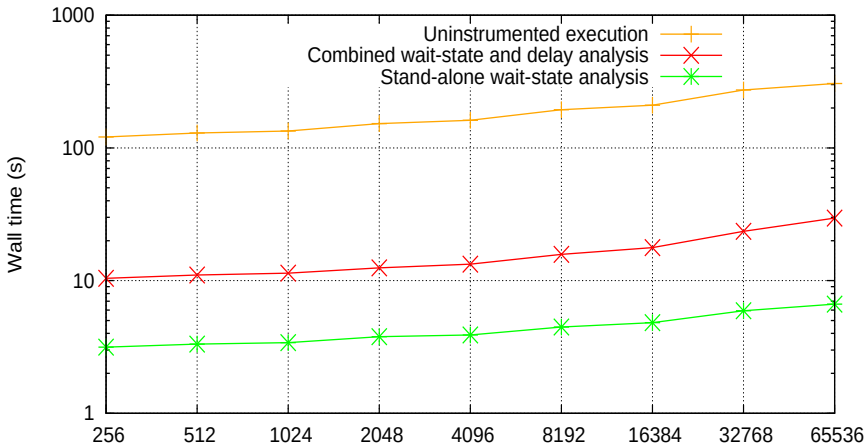


Fig. 3. Comparison of Sweep3D application execution time, combined wait-state and delay analysis time, and pure wait-state analysis time at various scales.

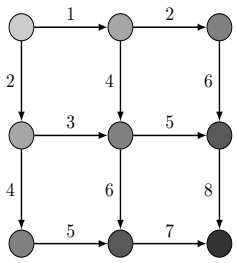


Fig. 4. Example of a 2D wavefront propagation.

```
do octants
  do angles in octant
    do k planes
      if neighbor(E/W) MPI_RECV(E/W)
      if neighbor(N/S) MPI_RECV(N/S)
        ... compute grid cell ...
      if neighbor(E/W) MPI_SEND(E/W)
      if neighbor(N/S) MPI_SEND(N/S)
    end do k planes
  end do angles in octant
end do octants
```

Listing 1. Code structure of the sweep() subroutine.

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 noticeably more time than the pure wait-state analysis, it scales equally well. As expected when replaying the original communication, both curves run in parallel to the unistrumented execution. We believe that the increase in trace-analysis time can be tolerated even for configurations larger than 65,536 – justified by the insight into the wait-state formation it provides, as we show below.

Before we explain the findings of our delay analysis, we briefly review the communication pattern of Sweep3D. The parallel computation follows a pipelined wavefront algorithm, propagating data along diagonal lines through the process grid. An example of such a wavefront starting at the upper-left corner of a 3×3 grid is shown in Figure 4 as a data-dependency graph. The arrows represent data dependencies, the numbers next to the arrows indicate the order of communications. In the source code, a subroutine called sweep() is responsible for the wavefront propagation. Wavefronts are initiated from all four corners of the process grid and pipelined to enable multiple wavefronts to follow each other along the same direction simultaneously. Wavefronts from a new direction already start while the pipeline from the previous direction is still drained. Listing 1 shows the basic code structure of sweep().

The performance of parallel wavefront algorithms in general and Sweep3D in particular has already been studied in great detail [21], [22]. In this article, we look at the performance of Sweep3D from an experimental point of view with a clear focus on identifying the origins of wait states in MPI communication. In the following, we concentrate our analysis on late-sender waiting time, as it is responsible for the largest part of overall waiting time. As can be seen in Table I, the amount of late-sender time as part of total execution time increases with the number of processes. With 16,384 processes, almost 40% of the aggregated runtime is spent in late-sender wait states. Examining the late sender delay costs, we find long-term delay costs to be much higher than short-term delay costs. Moreover, the percentage of long-term delay costs vs. the total delay costs grows from 89% to 98% as the number of processes is increased. Hence, the major performance issue in Sweep3D is not so much the computational imbalance itself, but rather the propagation of wait states across the process grid. Larger process counts exacerbate the problem.

Processes	256	1 024	4 096	16 384
Total execution time	30 945 s	137 210 s	665 170 s	3 454 600 s
Late sender time	2 460 s	20 333 s	168 750 s	1 317 600 s
Percent	7.95 %	14.82 %	25.37 %	38.14 %
Delay costs				
Short-term	257 s	1 204 s	5 615 s	24 448 s
Long-term	2 202 s	19 129 s	163 140 s	1 292 200 s

TABLE I
SWEEP3D LATE-SENDER WAITING TIME AND DELAY COSTS, ACCUMULATED ACROSS ALL PROCESSES.

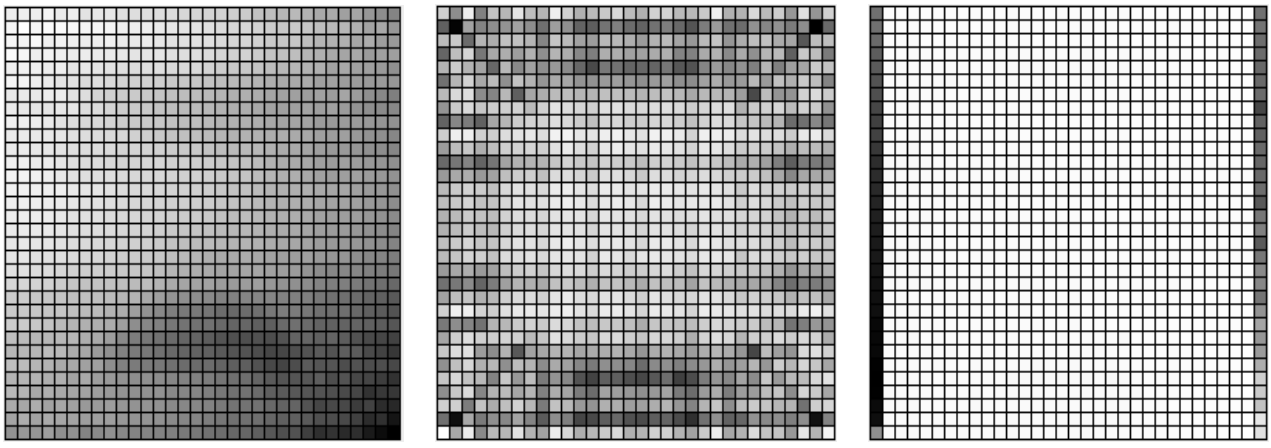
Figure 5(a) shows the distribution of late-sender waiting time across the process grid for a Sweep3D

run on 1024 processors. Waiting time varies between 16.6 and 24.2 s per process and increases from the upper-left to the lower-right corner of the process grid. Using our delay analysis, we can now determine the origin of wait-inducing load imbalance by analyzing the short-term delay costs. In Figure 5(b), the short-term costs are mapped onto the processes which cause them and not onto those where they materialize. Obviously, short-term delay costs originate from a relatively wide ring along the border of the process grid. There is almost no computational delay in the central region. However, short-term delay costs cover only a small fraction of the overall waiting time. As we know from Table I, the dominant part of waiting time are long-term delay costs. Interestingly, the origin of long-term delay costs (Figure 5(c)) is almost entirely confined to the vertical borders of the process grid. Waiting time caused by computational delay of these processes propagates through the system and induces the large majority of overall waiting time.

To explain these findings, recall the Sweep3D communication pattern. Processes receive messages first from east or west direction and then from north or south. Then they update their grid cells and subsequently send messages first to their eastern or western and then to their northern or southern neighbors, depending on the direction of the wavefront. Communication is done using blocking sends and receives. By waiting for their eastern or western neighbors first, processes remain unaffected by the delays at their northern or southern neighbors. As a consequence, wait states propagate primarily in horizontal direction. The only exception of this rule are the processes on the left and right borders: They do not have a western or eastern neighbor, respectively, which is why here any delay first propagates vertically, from where it then can spread horizontally. For this reason, computational delay of the border processes can propagate through almost the entire grid, whereas the delays of other processes usually affect only a single row. The left and right border processes therefore cause significantly higher long-term delay costs. While in this example merely localizing the wait states cannot adequately explain their formation, our delay analysis precisely pinpoints the propagation of wait states caused by computational delay on the vertical borders as the major cause of waiting time in Sweep3D – a non-obvious finding given the fairly complex communication pattern.

B. Zeus-MP/2

The Zeus-MP/2 code performs hydrodynamic, radiation-hydrodynamic (RHD), and magnetohydrodynamic (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.



(a) Late sender waiting time (b) Short-term delay costs (c) Long-term delay costs

Fig. 5. Distribution of late-sender waiting time, origin of short-term delay costs, and origin of long-term delay costs across the two-dimensional process grid with 1024 (32×32) processes. A darker color indicates a higher value.

We analyzed the 2.1.2 version on 512 processes simulating a three-dimensional magnetohydrodynamic wave blast, based on the "mhdblast_XYZ" example configuration provided with the distribution available for download. The number of simulated timesteps was limited to 100 in order to constrain the size of the recorded event trace. As with Sweep3D, we focus our analysis on the identification of 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 6 shows, the dominant part of 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 phase, respectively. Figure 7 illustrates the mapping of short- and long-term delay costs onto their responsible call paths. Especially the lorentz() and i-loop call-path locations engender a large fraction of long-term delay costs, indicating that delays in those call paths still manifest themselves in the endmost 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 8(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 picture 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 region 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 8(b)).

Examining the delay costs reveals that almost all direct delay originates from the border processes of the central, overloaded region (Figure 8(c)). The distribution of workload explains this observation: Within the central and outer regions, workload is relatively well balanced. Therefore, communication within the same region is not significantly delayed. In contrast, the large difference in computation time between 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 9 shows how delay in the lorentz() subroutine at the border of the

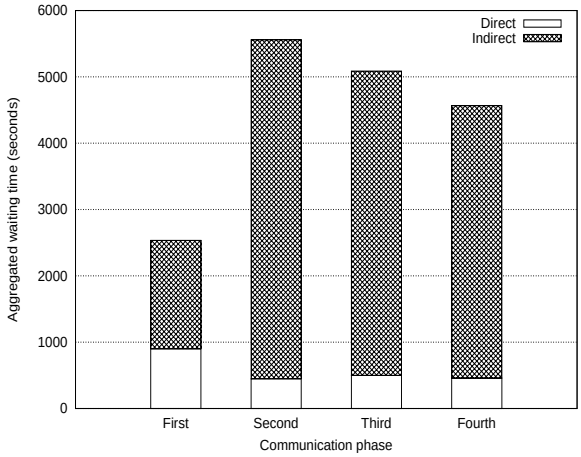


Fig. 6. Waiting time in communication phases in Zeus-MP/2, broken down in direct and indirect waiting time.

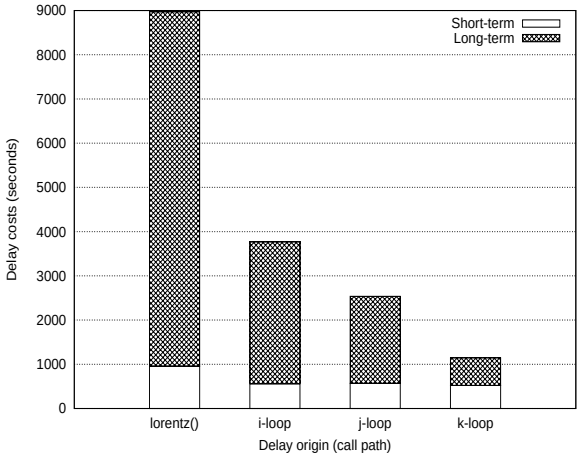
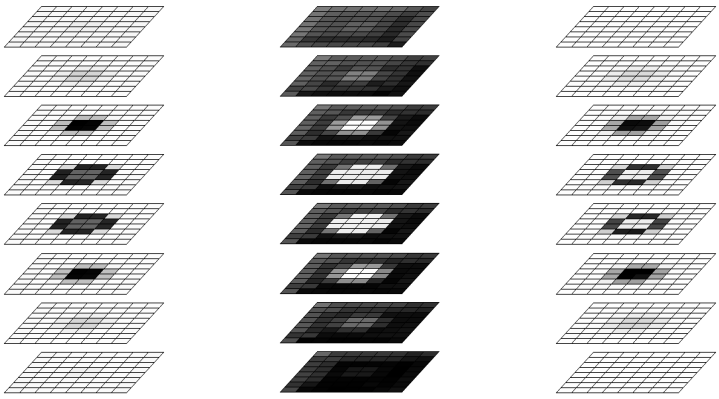
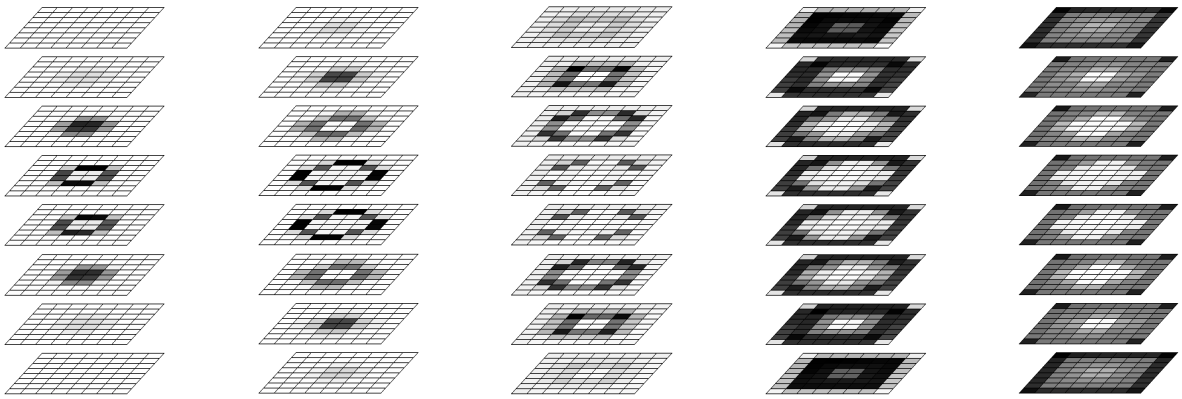


Fig. 7. Delay costs caused by four critical call paths of Zeus-MP/2, broken down in short- and long-term costs.



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

Fig. 8. Distribution of computation time, waiting time, and origin of delay costs in Zeus-MP/2 across the three-dimensional computational domain.



(a) Delay (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. 9. Propagation of delay caused by the lorentz() subroutine. Delay in lorentz() causes direct wait states in the first communication phase, which induce indirect wait states at the surrounding layer of processes, and travel further through the system during the second and third communication phase.

central region causes direct wait states at the surrounding processes during the first communication phase, which in turn cause indirect waiting time within the next layer of processes, and further propagate to the outermost processes during the second and third communication phase.

VI. Evaluation of Optimization Hypotheses

In general, the excess workload identified as a delay 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. Our goal is therefore to automatically predict the effects of redistributing a given delay without altering the application itself and to determine the savings we can realistically hope for. Since the effects of such changes are hard to quantify analytically, we combined our delay analysis with a framework developed earlier by the authors [23] that can simulate these changes via a real-time replay of event traces after they have been modified to reflect the redistributed load. To demonstrate our ideas using the Zeus-MP/2 example, we extended the initial prototype of the simulator to also support non-blocking communication [24].

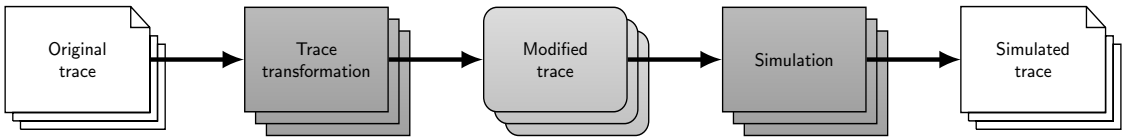


Fig. 10. Simulation workflow: The original trace is modified according to an optimization hypothesis and the modified trace is subsequently replayed in real time to rectify the distorted timestamps.

As illustrated in Figure 10, the simulator operates on the trace data already gathered for the delay analysis, first applying trace transformations according to the optimization hypothesis and subsequently performing a parallel real-time reenactment of the actions recorded in the trace using the original execution platform and configuration (i.e., number of cores, etc.). In the course of the simulation, computation intervals are simulated by elapsing the time in between using busy wait, whereas communication intervals are simulated by replaying the communication operations recorded in the trace using identical message sizes. Event timestamps are successively updated to reflect the simulated time as it progresses. By performing a parallel real-time replay, we can achieve high scalability and good predictive accuracy. Not relying on a potentially complex model of the message-passing subsystem, our method is also platform independent. After the simulation run is finished, the simulated trace is subject to a wait-state/delay analysis, allowing a detailed per call-path comparison of the original and the modified execution as well as a quantification of the overall execution time saving (or degradation) including the changes in waiting time. In the context of this paper, optimization hypotheses aim at the redistribution of direct delays, and thus, the elimination of their short- and long-term costs.

First, we validated the accuracy of the simulator for the Zeus-MP/2 example by performing an identity simulation (i.e., without any trace transformation applied). Compared to the original execution, the simulator reproduced the execution time as well as the wait states with a deviation of less than 1.5% and 1.0%, respectively, rendering the simulation accurate enough for our purposes. Then, we investigated the effect of globally balancing the lorentz() subroutine, which had been identified by our analysis in Section V as the root cause of a significant amount of direct and indirect waiting time. While keeping the overall computation time constant, the workload of the lorentz() subroutine was equally distributed across all processes in our simulation, effectively reducing the load imbalance seen in Figure 8(a). As can be seen

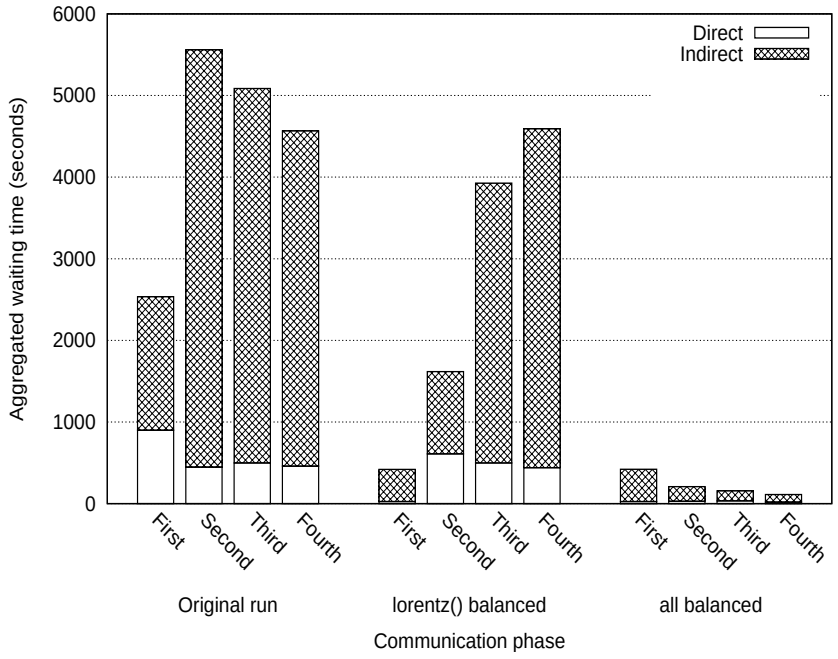
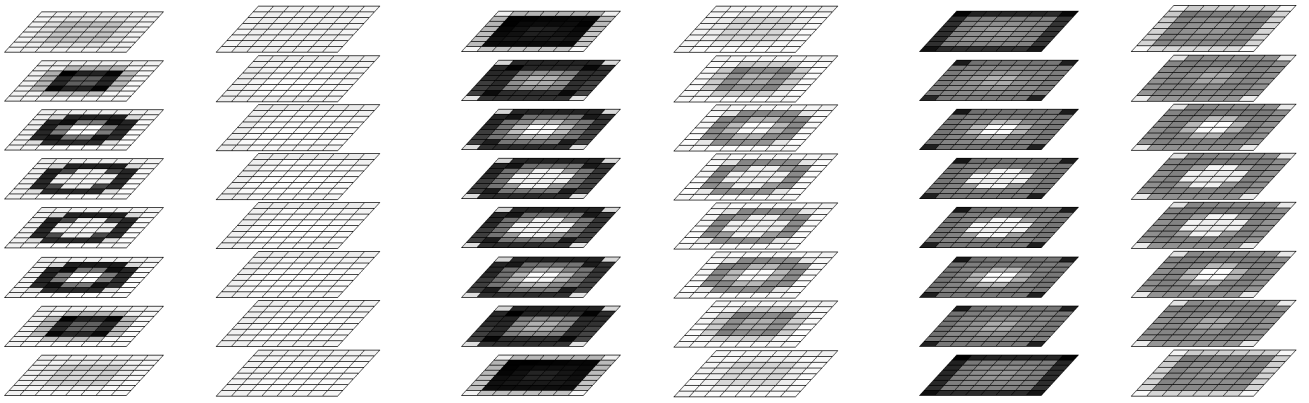


Fig. 11. Simulated reduction of waiting time in different scenarios during all four communication phases.



(a) 1st phase (orig.) (b) 1st phase (sim.) (c) 2nd phase (orig.) (d) 2nd phase (sim.) (e) 3rd phase (orig.) (f) 3rd phase (sim.)

Fig. 12. Simulated reduction of waiting time after globally balancing subroutine lorentz() during the first three communication phases.

in Figure 11, the simulation revealed that the wait states during the first communication phase after the lorentz() subroutine would be significantly reduced, almost completely eliminating the direct wait states and leaving only a small fraction of the long-term costs, which originated from delays outside lorentz(). In addition, the indirect wait states (i.e., the long-term costs) during the second and third communication phases were reduced, too, leading to an overall reduction of the waiting time by 36% and an overall runtime improvement of 4%. To better understand the effects of balancing lorentz(), Figure 12 visualizes the distribution of waiting times across the virtual process topology before and after the change for the first three communication phases. The direct wait states in the first communication phase at the processes neighboring the overloaded processes disappeared almost completely (Figure 12(b)), while the propagated wait states showing up during the second and third communication phase at processes farther outside appeared substantially reduced (Figures 12(d) and 12(f)).

In a next step, we additionally balanced the three computational loops inside the hsmoc() subroutine, almost entirely eliminating the load imbalance seen in Figure 8(a) and the majority of direct delays detected by our analysis. Thus, the simulation plausibly suggests that the removal of these load imbalances would remediate almost all of the remaining wait states during the four communication phases within the main iteration loop (Figure 11, right), reducing the overall runtime by approximately 11%. Unfortunately, since the publicly-available code was developed outside our organization, we neither had the expertise to reproduce these results using the real code nor could we find out whether such changes would have been possible at all.

VII. 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 us to gain non-trivial insights into the wait-state propagation occurring in two example codes running on up to 65,536 cores. Because wait states usually cannot be removed without redistributing work, we also showed how in principle the effects of complex transformations such as balancing the load inside a function can be simulated without changing the code itself. As illustrated, both facets of our delay analysis (i.e., the accounting of delay costs and the simulation of delay remediation) can be combined into a powerful diagnostic framework for identifying promising candidates for load balancing. In our

future work, we plan to conduct further application studies to demonstrate that the insights enabled by our method can be turned into real performance improvements.

ACKNOWLEDGMENT

Financial support from the Deutsche Forschungsgemeinschaft (German Research Association) 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] 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.

[4] 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.

[5] 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, November 2007, pp. 237–243.

[6] 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.

[7] 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, 2008, pp. 194–201.

[8] 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.

[9] 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.

[10] 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.

[11] 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.

[12] 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, (to appear).

[13] C. Mendes, "Performance prediction by trace transformation," in *Proc. of the 5th Brazilian Symposium on Computer Architecture*, Florianopolis, September 1993.

[14] G. Rodriguez, R. Badia, and J. Labarta, "Generation of simple analytical models for message passing applications," in *Proc. of the European Conference on Parallel Computing (Euro-Par)*, ser. Lecture Notes in Computer Science, vol. 3149. Pisa, Italy: Springer, August - September 2004, pp. 183–188.

[15] G. Zheng, T. Wilmarth, P. Jagadishprasad, and L. V. Kalé, "Simulation-based performance prediction for large parallel machines," *International Journal of Parallel Programming*, vol. 33, no. 2-3, June 2005.

[16] G. Lyon, R. Snelick, and R. Kacker, "Synthetic-perturbation tuning of MIMD programs," *The Journal of Supercomputing*, vol. 8, pp. 5–28, 1994.

[17] B. Barnes, B. Rountree, D. Lowenthal, J. Reeves, B. de Supinski, and M. Schulz, "A regression-based approach to scalability prediction," in *22nd ACM International Conference on Supercomputing*, Kos, Greece, June 2008, pp. 368–377.

[18] D. Becker, R. Rabenseifner, F. Wolf, and J. C. Linford, "Scalable timestamp synchronization for event traces of message-passing applications," *Parallel Computing, Special Issue EuroPVM/MPI 2007*, (to appear).

[19] Accelerated Strategic Computing Initiative, "The ASCI SWEEP3D benchmark code," http://www.c3.llnl.gov/pal/software/sweep3d/, 1995.

[20] 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.

[21] D. Sundaram-Stukel and M. K. Vernon, "Predictive analysis of a wavefront application using LogGP," in *Proceedings of the 7th ACM SIGPLAN Symposium on Priciples and Practices of Parallel Programming*, vol. 34, no. 8, Aug. 1999, pp. 141–150.

[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] 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.

[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 2009)*, Aug. 2009, (to appear).