Verifying Causality between Distant Performance Phenomena in Large-Scale MPI Applications

Marc-André Hermanns*, Markus Geimer*, Felix Wolf*† and Brian J. N. Wylie*
*Jülich Supercomputing Centre, Forschungszentrum Jülich, 52425 Jülich, Germany
†Department of Computer Science, RWTH Aachen University, 52056 Aachen, Germany
email: {m.a.hermanns, m.geimer, f.wolf, b.wylie}@fz-juelich.de

***Abstract*—In message-passing applications, the temporal or spatial distance between cause and symptom of a performance problem constitutes a major difficulty in deriving helpful conclusions from performance data. Just knowing the locations of wait states in the program is often insufficient to understand the reason for their occurrence. We present a method for verifying hypotheses on causality between temporally or spatially distant performance phenomena in message-passing applications without altering the application itself. The verification is accomplished by modifying MPI event traces and using them to simulate the hypothetical message-passing behavior. By performing a parallel real-time reenactment of the communication to be simulated using the original execution configuration, we can achieve high scalability and good predictive accuracy in relation to the measured behavior. Not relying on a potentially complex model of the message-passing subsystem, our method is also platform independent.**

I. Introduction

As a prerequisite for the productive use of state-of-the-art supercomputers, the HPC community needs powerful and scalable performance-diagnosis tools that make the optimization of parallel applications both more effective and more efficient. One major difficulty application developers are confronting with traditional performance tools is that the tools often diagnose only the symptoms of performance problems but not necessarily their causes. In fact, the symptoms may appear (i) much later than the event causing it, (ii) on a different processor, or (iii) both. The temporal or spatial distance between cause and symptom constitutes a substantial challenge in deriving helpful conclusions from a set of performance data. Especially in message-passing applications, load imbalance may create wait states at the next synchronization point following the imbalance. When trying to scale communication-intensive applications to large processor counts, such wait states can present severe challenges to achieving good performance. Of course, these effects are not necessarily confined to load imbalance and may be initiated by a large variety of behaviors, e.g., disparate communication requirements or coordination activities that are performed only by designated processes. Additionally, messages may propagate wait states from one process to the next, creating potentially complex and far-reaching cause-effect chains. Finally, the individual contribution of a performance phenomenon to a given wait state is hard to quantify because wait states may occur as a superposition of several influences.

In our previous work on the Scalasca toolset [1], we have shown that wait states in MPI message passing programs can be identified by searching event traces for characteristic patterns – even at very large scales. Here, we present a complementary approach aimed at better understanding their causes. Drawing from earlier ideas on trace-based performance prediction [2]–[5], we have designed and implemented a simulator called SILAS (SImulation of LArge-Scale parallel applications) that can be used to verify hypotheses on causal connections between different performance phenomena at very large scales. The verification is accomplished by modifying event traces according to a hypothesis and using them to simulate the hypothetical message-passing behavior. The predicted behavior can then be scanned for wait states to investigate how the modification would influence (and hopefully reduce) their occurrence in various parts of the program. Typical questions the simulation can answer encompass how the performance behavior changes if a specific computation is more evenly distributed across the machine or if a specific communication operation is replaced or eliminated.

As a distinctive property, the simulator performs a parallel real-time reenactment of the communication to be simulated using the original execution configuration. Supporting conclusions with respect to the same hardware and an identical number of processes, our approach offers the following advantages:

- Scalability – Since the simulation is carried out at the original scale, that is, on as many CPUs as have been used to generate the traces, processing capabilities (i.e., both processors and memory) grow proportionally with the number of application processes, allowing us to simulate execution configurations with thousands of processes.
- Accuracy and platform independence – The real-time replay eliminates the need for *modeling* communication and, thus, removes a major source of prediction inaccuracy. At the same time, using the communication substrate of the target system automatically integrates the most important platform-specific parameters at basically no additional per-platform design cost. Porting the simulator to a new system is therefore straightforward.

The simulator has been designed to enhance the trace-analysis functionality of the Scalasca toolset by adding accurate and scalable predictive capabilities. Our ultimate objective is to go beyond Scalasca's present localized wait state diagno-

sis by automatically identifying and evaluating hypotheses on how the detected wait states can be most effectively removed. The current prototype of the simulator has been tested and evaluated on an IBM Blue Gene/L system.

In this article, we give an overview of the simulator and show how it can be used to accurately predict the effects of very fine-grained changes in the application behavior. We start with a review of related work in Section II. In Section III, we describe the basic workflow of verifying optimization hypotheses, outlining the usage of the simulator in the context of the Scalasca toolset. In Section IV, we illuminate fundamental design principles, explain key mechanisms, and discuss limitations. Experimental evidence of accurate predictions at larger scales using both synthetic benchmarks and real applications is presented in Section V. Finally, in Section VI, we conclude the paper and outline future directions.

II. Related Work

The principle of trace-driven performance prediction has already been intensively studied. Several approaches have addressed questions about performance implications when varying architectural parameters, such as CPU speed and network latency and bandwidth, and to a lesser extent also when introducing *synthetic perturbations* [6] that reflect modified application-level behavior.

Mendes transformed event traces of message-passing applications according to a prediction model based on relative processor speed, optionally differentiated by code section, and message transfer times previously obtained from benchmark measurements as a function of the message size [2].

An early performance-analysis toolkit offering trace-based simulation capabilities as one element of a comprehensive feature catalog is AIMS [3], which estimates the scalability of parallel applications by extrapolating previously generated execution traces to higher numbers of processors and larger problem sizes. The extrapolated traces can be subsequently analyzed using standard trace-analysis modules provided by the toolkit.

Originally motivated by the need to study the sharing of multiprocessors among multiple applications, DIMEMAS [4] provides the ability to simulate the execution behavior of parallel programs based on previously generated event traces. The underlying prediction model allows the adjustment of relative processor speeds, network bandwidth and latency within and across nodes, the number of input and output links, and the processor scheduling policy. Besides simple architectural parameter studies, DIMEMAS has been used to investigate the effects of scaling individual program states and to develop analytical models as a functions of various architectural parameters by extrapolating simulations from multiple traces. While DIMEMAS itself is a sequential tool, traces used as input for DIMEMAS stem from message-passing or multithreaded programs.

Predicting application performance for emerging architectures larger than those at one's disposal is the focus of BigSim [7]. Based on Charm++, an object-based and message-driven parallel programming language, BigSim combines an emulator that is capable of running larger numbers of virtual processes on a smaller number of physical processors with a postmortem simulator that uses traces generated during an emulated run.

Compared to the approaches described above, our work clearly concentrates on the effects of fine-grained alterations of application-level behavior with respect to the performance under an identical execution configuration. Typical use cases include the balancing of individual functions or the elimination or replacement of communication operations. 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 relieves us of the burden of modeling the extremely complex communication infrastructures found on today's large-scale machines.

III. Hypothesis Verification

In this section, we describe the typical usage scenario of our simulator in the context of the Scalasca toolset. Scalasca has been specifically designed for use on large-scale systems including IBM Blue Gene and Cray XT, but is also well-suited for small- and medium-scale HPC platforms. A distinctive feature of Scalasca is the ability to identify wait states in event traces of MPI applications with very large numbers of processes using a parallel replay of the target application's communication behavior [1].

Looking for ways to extend our trace analysis toward a better understanding of the relationship between imbalanced execution and wait states led to the idea of designing a trace-based simulator, capable of operating at very large scales and accurate enough to predict the long-range effects of potential optimizations on the formation of wait states later in the program. Since no source-code modification is required, it should become possible to automatically test a larger number of optimization hypotheses derived from the original trace data and rank them according to the expected performance benefit to identify the most promising ones.

Figure 1 illustrates the role of the simulator in the procedure of verifying hypotheses on causality between temporally or spatially distant performance phenomena. The general objective of the process is to generate wait-state analyses from both the measured and the predicted behavior and compare the results to allow conclusions on the effects of hypothetical program modifications with respect to wait states and other performance metrics. The workflow starts with running the instrumented target application in the execution configuration we want to make predictions for and generating an event trace consisting of one trace file per application process. During all subsequent steps, access to the event trace occurs through a parallel object-oriented high-level API [8]. The primary usage model of the API assumes a one-to-one mapping between application and tool processes, that is, for every process of the target application, one tool process is created that loads the corresponding trace data into main memory and offers

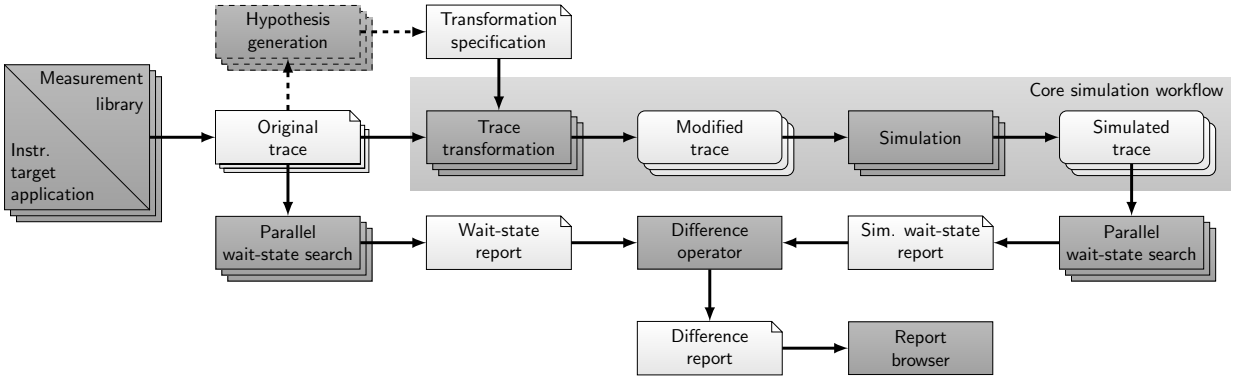


Fig. 1. Workflow for verifying optimization hypotheses. Dark rectangles denote programs, light rectangles with the upper right corner turned down denote files, and light rectangles with rounded corners denote data objects residing in memory. Stacked symbols indicate multiple instances of programs, files, or data objects running or being processed in parallel. The target application generating the event trace is the entry stage of the workflow. Judging the difference between normal execution and the predicted outcome of the optimization displayed in the report explorer is the final stage.

random access to individual events. Data exchange among tool processes is accomplished via MPI communication.

A hypothesis includes the specification of a trace transformation, which may prescribe the adjustment of event timestamps, the deletion of existing events, or the insertion of new events to model changes in the application's source code. As already pointed out, our ultimate objective is the automatic derivation of suitable hypotheses from the original trace data, for example, after identifying local or global load imbalances or other disparities among application processes (shown in Figure 1 using dashed lines). Currently, a set of parameterized standard transformations including the scaling of functions or the elimination of messages can be specified via a configuration file and provided as input to the trace-transformation stage. Arbitrary transformations can be implemented as hand-written programs utilizing the aforementioned trace-access API, which has been extended for this purpose by adding an interface to modify the trace data.

After the transformation has been applied, the simulator performs a parallel real-time replay of the events stored in the trace. Computation intervals are simulated simply by elapsing the time in between using busy wait, whereas communications are simulated by reenacting the communication operations recorded in the trace. Thus, the time of a communication is determined by the time needed to execute the corresponding MPI call under modified conditions. As the simulation progresses, event timestamps are adjusted to reflect the time elapsed since simulation start. Obviously, keeping all the trace data in memory is an essential prerequisite for performing the simulation in real time because reading the trace data from file in the course of the replay can severely compromise simulation accuracy unless such interruptions are appropriately accounted for.

Finally, a wait-state search is performed on both the original and the simulated event trace, classifying and quantifying all instances broken down by call path and process. The results of the two analyses are subtracted using a difference operator [9] defined over the set of potential search outputs. For every type of wait state, the operator essentially calculates the element-wise difference between corresponding (call path, process) matrices, taking into account that the simulated run may exhibit call paths not present in the original run and vice versa. The difference data set can be visually explored to assess the changes the modified behavior has brought about, in particular with respect to the reduction or migration of wait states. Propagating the effects of changes starting from the point of their injection onwards through the entire execution and also carrying influences over to remote processes, our simulator allows the verification of causal connections between temporally or spatially distant performance phenomena within the confidence limits our simulator offers.

IV. Replay-based Simulation

In this section, we examine the core simulation workflow (shaded area in Figure 1) in more detail. Using the simple example depicted in Figure 2, we illustrate the two elementary steps of trace transformation and simulation. We explain fundamental design principles of the simulator and discuss techniques applied to ensure satisfactory simulation accuracy.

A. Trace Transformation

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 and the call path are specified as event attributes.
- Sending and receiving messages. Message tag, communicator, and size are specified as event attributes.
- Exiting collective communication operations. This special exit event carries event attributes specifying the communicator, the amount of data 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 begin

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). Our event model currently ignores other types of communication, such as RMA, and file I/O.

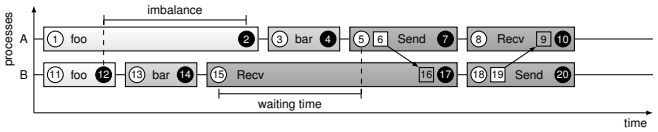
At a lower level, the event trace can be modified by altering event timings, deleting existing events, inserting new events, and otherwise changing arbitrary event attributes relevant to the replay. Since all events carry absolute timestamps, the modification of a timestamp may necessitate modifying the timestamps of subsequent events. Modifying the end times of communication operations is not necessary because these times will be "measured" during the simulation, as we will see in Section IV-B. As a preliminary model of a higher-level mechanism, we have implemented a few sample transformations, such as scaling regions or balancing regions among processes both globally and on a per-instance basis. Further transformations, such as substituting communication operations or modifying message parameters, will be added as we gain more experience with application test cases. The use of a higher-level mechanism, which is currently accessible via a configuration file supplied as input to the simulator, has the advantage that consistency constraints ensuring the logical integrity of the trace (e.g., avoiding dangling messages sent without matching receive event) can be more easily enforced.

Figure 2a shows an event trace generated from two MPI processes. After executing the functions `foo()` and `bar()` in a row, both processes engage in two message communications via matching pairs of `MPI_Send()` and `MPI_Recv()`. Whereas the first time the message is sent from A to B, the second time the message is sent in the opposite direction. Apparently, function `foo()` exhibits an imbalance because process B spends less time in `foo()` than process A does. Function `bar()`, in contrast, is entirely balanced. The imbalance in `foo()` indirectly causes process B to wait for the message sent by A during the first communication, a situation also known as *Late Sender*. No wait state can be observed during the the second communication.

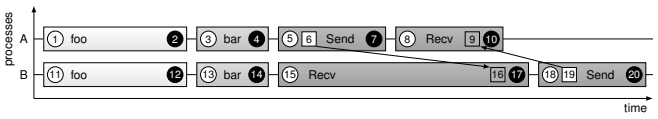
Our obvious hypothesis is that the wait state in the first `MPI_Recv()` can be removed by balancing function `foo()` with expected benefits for the overall performance. Balancing `foo()` during trace transformation yields the trace shown in Figure 2b, with the timestamps of events e_2 and e_{12} being modified and the timestamps of all subsequent events adjusted accordingly. Of course, the lengths of the communication intervals now seem distorted because the MPI calls are simply shifted to the left or to the right without accounting for changes that would occur if the MPI calls were carried out under these new conditions. Note that the receive event of process A (e_9) now happens before the matching send event (e_{19}), violating the causal event order. The task of rectifying this distortion is left to the actual simulation.

B. Simulation

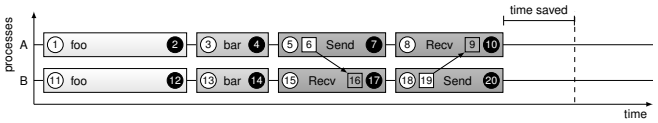
As event traces model only a very restricted view of the application behavior, the simulator faces the challenge of having to approximate both computation and communication



(a) Original trace where an unbalanced execution of `foo()` causes a Late Sender situation (receiver has to wait for the corresponding sender) during the first message exchange.



(b) Balancing region `foo()` during trace transformation modifies all subsequent timestamps to preserve temporal offsets.



(c) The replay-based simulation measures all timestamps while reenacting the application behavior, thus, adjusting the communication operations to their predicted length.

Fig. 2. Original event trace (a), event trace after trace transformation (b), and simulated event trace (c). Circles denote enter and exit events, squares denote send and receive events.

accurately enough to produce realistic event timings in the output trace. Because input and output of the simulator are on the same abstraction level, our primary focus is the length of intervals between events but not necessarily what happens inside.

The general principle of the simulation is to traverse the event trace in parallel, each simulation process being responsible for a different application process, whose trace data resides in the memory of the simulation process. During the traversal, each simulation process examines the events assigned to it in chronological order and takes different actions depending on the type of the event and its associated interval. The traversal is performed in real time, that is, an event is reached at the time it is supposed to occur during the simulated run. For the purpose of the simulation, we regard everything that occurs outside a communication operation as computation. As a general rule, computation intervals are simply elapsed, whereas communication intervals are filled by reenacting the corresponding communication operation. In the course of the simulation, timestamps are successively changed to simulated time.

a) Computation: A computation interval is simulated by elapsing the corresponding time span, whether it is still the original one or whether it has been modified during trace transformation. This is accomplished by calling a wait function, supplying the requested time interval as an argument to a simple busy wait, implemented using highly-accurate timers available on the target system. We have found this to be a portable solution, as the timer functionality is already provided by the Scalasca infrastructure in a platform-independent way.

b) Communication: To accurately replay the communication, we use the communication operations specified in the

modified event trace with identical send and receive buffer sizes. Since the data type is not recorded in the trace, we always transfer arrays of type MPI_BYTE. The current event model used by Scalasca already provides enough information to simulate most blocking MPI point-to-point and collective operations. Extensions to cover a wider range of operations including non-blocking communications that will be sufficient to support most of today's MPI codes are straightforward and already in progress. The clock value after performing a communication operation determines the operation's exit timestamp, whereas the length of the preceding computation interval determines the operation's enter timestamp. Emulating the way typical PMPI wrapper functions are implemented, send and receive timestamps correspond to the clock values before performing a send operation or after performing a receive operation, respectively.

Figure 2c shows the simulated trace based on the assumption that function foo() can be perfectly balanced. Since events e_5 and e_{15} now occur simultaneously, the waiting time inside the first receive operation disappears, moving events e_7 and e_{17} to the same position on the time axis. As a consequence, both processes enter the second communication at the same time (e_8 and e_{18}), correcting the causality violation still visible in 2b. As a net result, our simulation predicts that balancing function foo() reduces the overall execution time by the time indicated in the diagram. Note that the simultaneous completion of matching communication operations has only been chosen to keep the example simple and does not represent an inherent assumption of our simulator. Of course, the communication reenactment would account for potential completion offsets occurring under real conditions.

c) Small intervals: Because the simulation is performed in real time, one potential source of inaccuracy in our approach is the simulation of small intervals – especially of those that are smaller than the resolution of our wait function. Every call to this routine incurs a certain overhead, as it requires querying the system timer at least once. It is therefore preferable to reduce the granularity of the time measurements and make the time spans spent waiting as long as possible. For this reason, adjacent computation intervals are grouped together in a preprocessing step and later simulated in one chunk. After the replay, the timestamps of events delimiting individual parts of this super-interval are readjusted according to their relative distance. Small intervals between consecutive communication calls that cannot be grouped together are approximated to avoid the overhead of calling the timer.

C. Limitations

Currently, our simulator is not capable of correctly replaying non-blocking MPI point-to-point communication, as information on communication requests is not yet properly recorded in the trace data. Likewise, the non-determinism expressed in wildcard receives using MPI_ANY_SOURCE and/or MPI_ANY_TAG is not retained. Instead, the replay uses source and tag information identified during trace acquisition, thus, restricting the order in which messages are received during the simulation to the order previously observed. However, the required information can in principle be recorded in event traces to correctly model these two aspects. An appropriate extension of the event model is currently being pursued.

MPI collective operations transferring a different amount of data per process such as MPI_Gatherv(), can only be approximated using their less specific counterparts, as space requirements only allow us to record the aggregate amount of data sent and received for these routines. Also, our current approach is oblivious of MPI data types, which may misrepresent the computational overhead associated with reduction operations. Moreover, due to large variations in file system performance usually observed, we found replaying file I/O to offer little predictive value. Instead, file I/O is treated in the same manner as computation intervals are, that is, it is simulated using the busy wait function. Finally, we are aware that just spinning during computation intervals ignores potential interactions between processes through the memory subsystem. By shifting the relative time at which concurrent memory accesses of processes co-located on the same SMP node take place, the overall memory bandwidth requirements may change. It should be noted, however, that most of these issues reach far beyond the fidelity of analytical approaches our method can be compared to.

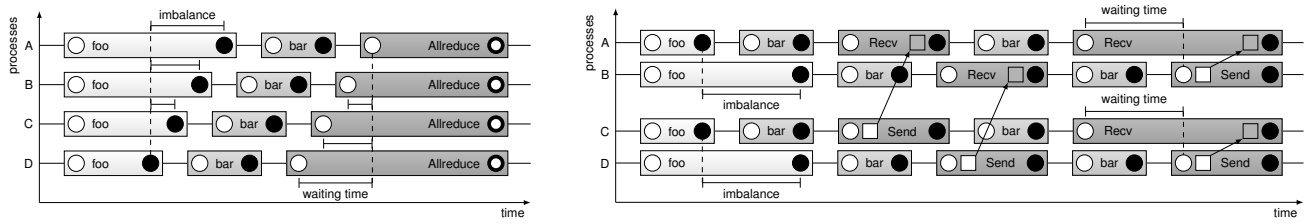
V. Examples

In this section, we report experiences gained so far with our simulator using both synthetic benchmarks, where the code can be more easily modified to reconstruct the hypothetical behavior in reality, and more complex real-world applications. After validating the overall accuracy of the simulation using unmodified trace data, we verified optimization hypotheses related to load balancing and improved communication behavior. All experiments were performed on the 8-rack IBM Blue Gene/L system JUBL at the Jülich Supercomputing Centre in coprocessor mode. The numbers reported always refer to the accumulated execution time across all processes.

A. Identity Simulation

One way of validating the overall simulation accuracy is to perform an *identity simulation*, that is, replaying a recorded event trace without applying any prior transformation, and comparing the predicted to the original behavior. For this purpose, we conducted measurements with the MPI version of the ASC SWEEP3D benchmark code [10] at a range of scales from 32 to 4,096 processes. The application was configured to run for a few minutes, with the problem size per process being roughly constant (i.e., weak scaling).

In our experiments, the relative deviation of the overall execution time predicted by the simulator from the execution time measured during an actual run was rather small and never exceeded 0.6%. As positive and negative errors occurring in different parts of the program may compensate each other, we added the absolute values of individual errors across all (call path, process) combinations. In relation to the total execution time, this sum was less than 0.8% in



(a) LB-COLL. Load imbalance in function `foo()` induces wait states at the next synchronizing collective communication. White circles with black borders denote collective exit events.

(b) LB-P2P. Load imbalance between even and odd ranks in function `foo()` induces wait states at the next point-to-point communication operation between pairs of even and odd ranks.

Fig. 3. One iteration of each of the two synthetic examples LB-COLL and LB-P2P, illustrating the long-range effects of load imbalance in function foo().

all configurations, demonstrating that a reasonable level of accuracy was sustained throughout the entire program. The instrumentation overhead created during trace acquisition was negligible for all configurations.

B. Load Balancing

Load imbalance is a common source of wait states in message-passing applications. Here, we present two synthetic benchmark programs with wait states being indirectly induced by load imbalance, propagating to the affected communication across a longer range of execution time through a phase of balanced behavior (Figure 3). Using these two examples, we demonstrate our simulator's ability to accurately predict the reduction of waiting time after removing the imbalance, thus verifying a causal connection between these two distant performance phenomena.

The first example is called LB-COLL and generates a *Wait at N×N* inefficiency pattern, where a load imbalance induces waiting times at the next synchronizing collective communication. Figure 3a shows one possible incarnation of this pattern, as it appears in our example. In this program, a sequence of three function calls is executed inside a loop of 100 iterations. The first routine is called `foo()`, emulating a load imbalance by making the execution time dependent on the rank number. The last function call in each iteration is `MPI_Allreduce()`, implicitly synchronizing all processes involved due to the all-to-all character of the communication. To show the long-range effects of the perturbation introduced by the imbalance, a routine `bar()` is executed in between, taking the same amount of time for each process.

The second example is called LB-P2P and generates a *Late Sender* inefficiency pattern, as depicted in Figure 3b. Load imbalance between processes with odd and even rank numbers causes processes A and C to wait in a later point-to-point receive operation. In this more complex case, not only computational phases (i.e., calls to `bar()`) appear between cause and symptom of the imbalance, but also additional communications involving other combinations of processes. Again, 100 iterations of the illustrated behavior were performed.

In both cases, the simulator was used to verify the hypothesis that the imbalance in function `foo()` is mainly responsible for the later formation of wait states and that balancing it would substantially contribute to their reduction. To validate the accuracy of our prediction, simulated executions were compared to measurements with a version of the program that had been previously modified according to our hypothesis. Like in the previous case, the experiments were performed on a range of scales from 32 to 4,096 processes. In relation to the results obtained for the identity simulation of SWEEP3D, the relative deviation of the predicted from the measured execution time was even smaller for both examples (in the order of 0.002%, i.e., showing only measurement noise). Contrasting the trace-analysis results of the original runs with the results of the simulated optimized runs using the difference operator introduced in Section III revealed that the simulated balancing of function `foo()` would eliminate the majority of the Late Sender pattern instances, as was expected. This result was confirmed by the measured optimized runs.

C. Altering Communication Behavior

XNS [11], a computational fluid dynamics application based on finite-element techniques on irregular three-dimensional meshes, serves as an example for a very substantial alteration of communication behavior. The code consists of roughly 45,000 lines of mixed Fortran and C in more than 100 files and has already been subject to performance analysis and subsequent optimization using the Scalasca toolset [12]. During this work, the unnecessary use of zero-sized point-to-point message transfers has been identified as a major scalability bottleneck. With respect to our simulation approach, this application example was especially interesting as it not only allowed us to show the contribution of a single performance problem to the formation of wait states in point-to-point communication, but also the accurate prediction of secondary effects, such as the migration of wait states after eliminating the cause of their initial materialization.

The basis of our investigation was an event trace acquired for one simulation time step during a run with 1,024 processes using a version of the program where the `MPI_Sendrecv()` calls responsible for the zero-sized messages had already been replaced with pairs of individual calls to `MPI_Send()` and `MPI_Recv()`. In future work, we plan to utilize the trace modification API outlined in Section IV-A to perform this step automatically during the trace-transformation stage without touching the source code itself. According to wait-state search results obtained for the original trace, the application suffered from a high fraction of time spent in MPI (59.9%) with roughly half of it attributable to Late Sender wait states.

Metric	Orig.	Hand-opt.	Δ	Simul.	Δ
Total	100.0	50.6	-49.4	53.1	-46.9
MPI	59.9	16.9	-43.0	19.4	-40.9
Point-to-Point	54.2	8.6	-45.6	11.2	-43.0
Late Sender	30.6	5.7	-24.9	8.0	-22.6
Wait at Barrier	5.1	7.7	+2.6	7.7	+2.6

TABLE I
COMPARISON OF THE ORIGINAL, THE HAND-OPTIMIZED, AND THE SIMULATED APPLICATION BEHAVIOR OF XNS. ALL NUMBERS REPRESENT PERCENTAGES OF THE ORIGINAL EXECUTION TIME.

Our transformation consisted of eliminating all transfers of zero-sized messages occurring inside two problematic routines identified during an earlier trace analysis to assess their contribution to the wait states observed. Although conceptually simple, applying the transformation meant eliminating more than 1.2 billion messages from the trace, which corresponds to more than 90% of the total number of message transfers.

Table I compares the execution behavior of the time-step loop of the original XNS version, a hand-optimized version, and the simulated optimization with respect to different time metrics. All numbers represent percentages of the original execution time. The predicted overall improvement was 46.9% compared to a measured improvement of 49.4%. In addition, the simulator predicts not only significant savings with respect to Late Sender wait states (22.6%), but also the migration of a smaller amount of waiting time to barrier synchronizations (2.6%) as a secondary effect. The obviously small deviations from the hand-optimized version mostly originated from the Late Sender metric, with the actual saving exceeding the prediction by about 2% of the original execution time. Thus, our simulator was able to establish a causal relationship between zero-sized messages and Late Sender wait states as well as to foresee a small amount of wait-state migration after their removal with reasonable accuracy. Note that the reduction in non-MPI time (difference between Total and MPI) observable in both the hand-optimized and the simulated run can be explained with the fact that a smaller number of MPI calls causes less instrumentation overhead.

VI. CONCLUSION

We have presented a novel approach to verifying hypotheses on causal connections between distant performance phenomena in MPI message-passing applications without altering their source code. Using trace-based simulation in the original execution configuration, we can accurately assess long-range effects of a variety of behaviors related to computation and communication. Since the simulation correctly propagates the influence expressed by an optimization transformation even across process boundaries via message communication, the initial cause and the final symptom may also be separated along the space dimension. The methodological key difference to earlier approaches is a parallel real-time reenactment of the simulated communication at the original scale, allowing the efficient simulation of MPI applications with thousands of processes. Moreover, since the reenactment eliminates the need to model the extremely complex communication infrastructures found on today's large-scale machines, our approach is also platform independent. Accurate predictions were shown for examples of increasing complexity with up to 4,096 processes.

As a next step, we plan to incorporate support for non-blocking communication and wildcard receive operations, as anticipated in Section IV-C, and evaluate our simulator with a broader range of realistic codes. As our ultimate goal is automatically identifying suitable optimization hypotheses, the simulator is intended to form the core component of a more universal tuning framework, where it will be used to verify optimization hypotheses derived from the original trace data. For this purpose, our future work will include the development of new trace-analysis algorithms with emphasis on the characterization of load and communication imbalance.

ACKNOWLEDGMENT

This work has been supported by the Helmholtz Association of German Research Centers under Grants No. VH-NG-118 and No. VH-VI-228 (Virtual Institute - High Productivity Supercomputing).

REFERENCES

[1] M. Geimer, F. Wolf, B. Wylie, and B. Mohr, "Scalable parallel trace-based performance analysis," in *Proc. of the 13th European Parallel Virtual Machine and Message Passing Interface Conference (EuroPVM/MPI)*, ser. Lecture Notes in Computer Science, vol. 4192. Bonn, Germany: Springer, September 2006, pp. 303–312.

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

[3] J. Yan, S. Sarukkai, and P. Mehra, "Performance Measurement, Visualization and Modeling of Parallel and Distributed Programs using the AIMS Toolkit," *Software – Practice and Experience*, vol. 25, no. 4, pp. 429–461, 1995.

[4] 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.

[5] G. Zheng, G. Kakulapati, and L. V. Kalé, "BigSim: A parallel simulator for performance prediction of extremely large parallel machines," in *Proc. of the 18th International Parallel and Distributed Processing Symposium (IPDPS)*. Santa Fe, New Mexico: IEEE Press, April 2004.

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

[7] 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.

[8] M. Geimer, F. Wolf, A. Knüpfer, B. Mohr, and B. Wylie, "A parallel trace-data interface for scalable performance analysis," in *Proc. Workshop on State-of-the-Art in Scientific and Parallel Computing (PARA, Umeå, Sweden)*, ser. Lecture Notes in Computer Science 4699. Springer, June 2006, pp. 398–408.

[9] F. Song, F. Wolf, N. Bhatia, J. Dongarra, and S. Moore, "An algebra for cross-experiment performance analysis," in *Proc. of the International Conference on Parallel Processing (ICPP)*. Montreal, Canada: IEEE Society, August 2004, pp. 63–72.

[10] *The ASC SWEEP3D Benchmark Code*, Advanced Simulation and Computing Program (ASC), https://asc.llnl.gov/.

[11] M. Behr, D. Arora, O. Coronado, and M. Pasquali, "Models and finite element techniques for blood flow simulation," *International Journal of Computational Fluid Dynamics*, vol. 20, pp. 175–181, 2006.

[12] B. Wylie, M. Geimer, M. Nicolai, and M. Probst, "Performance analysis and tuning of the XNS CFD solver on BlueGene/L," in *Proc. of the 14th European Parallel Virtual Machine and Message Passing Interface Conference (EuroPVM/MPI)*, ser. Lecture Notes in Computer Science, vol. 4757. Paris, France: Springer, September 2007, pp. 107–116.