A scalable tool architecture for diagnosing wait states in massively parallel applications

Markus Geimer[a], Felix Wolf[a,b,*], Brian J. N. Wylie[a], Bernd Mohr[a]

[a] *Jülich Supercomputing Centre, Forschungszentrum Jülich, 52425 Jülich, Germany*

[b] *Department of Computer Science, RWTH Aachen University, 52056 Aachen, Germany*

Abstract

When scaling message-passing applications to thousands of processors, their performance is often affected by wait states that occur when processes fail to reach synchronization points simultaneously. As a first step in reducing the performance impact, we have shown in our earlier work that wait states can be diagnosed by searching event traces for characteristic patterns. However, our initial sequential search method did not scale beyond several hundred processes. Here, we present a scalable approach, based on a *parallel replay* of the target application's communication behavior, that can efficiently identify wait states at the previously inaccessible scale of 65,536 processes and that has potential for even larger configurations. We explain how our new approach has been integrated into a comprehensive parallel tool architecture, which we use to demonstrate that wait states may consume a major fraction of the execution time at larger scales.

Key words: performance analysis, scalability, event tracing, pattern search

1 Introduction

Faced with increasing power dissipation and with little instruction-level parallelism left to exploit, computer architects are realizing performance gains by using larger numbers of moderately fast processor cores rather than by further increasing the speed of uniprocessors. As a consequence, numerical simulations

* Corresponding author.
Email address: `f.wolf@fz-juelich.de` (Felix Wolf).

are being required to harness much higher degrees of parallelism in order to satisfy their growing demand for computing power. However, writing code that runs efficiently on large numbers of processors and cores is extraordinarily challenging and requires adequate tool support for performance analysis. Increased concurrency levels impose higher scalability demands not only on applications but also on software tools. When applied to larger numbers of processors, familiar tools often cease to work in a satisfactory manner (e.g., due to escalating memory requirements, limited I/O bandwidth, or failing displays).

In message-passing applications, which still constitute the major portion of large-scale applications running on systems such as IBM Blue Gene or Cray XT, processes often require access to data provided by remote processes, making the progress of a receiving process dependent upon the progress of a sending process. If a rendezvous protocol is used, this relationship also applies in the opposite direction. Collective synchronization is similar in that its completion requires a certain degree of progress for each participating process. As a consequence, a significant fraction of the time spent in communication and synchronization routines can often be attributed to wait states that occur when processes fail to reach implicit or explicit synchronization points in a timely manner, for example, as a result of an unevenly distributed workload. Especially when trying to scale communication-intensive applications to large processor counts, such wait states can present severe challenges to achieving good performance.

As a first step in reducing the impact of wait states, application developers need a diagnostic method that allows their localization, classification, and quantification especially at larger scales. Because wait states cause temporal displacements between program events occurring on different processes, their identification can be accomplished by logging those events along with a timestamp in event traces. Actions typically stored in such traces include entering/leaving a function or sending/receiving a message. While, in principle, wait states are visible in time-line diagrams generated from event traces by graphical display tools such as Vampir [21], we have shown in our earlier work on the KOJAK trace analyzer [28] that wait states can be identified more effectively by automatically searching the trace data for execution patterns indicating their occurrence. In addition to usually being faster than a manual analysis performed using a trace browser, this approach is also guaranteed to cover the entire event trace and not to miss any instances.

However, as the number of processors used by individual applications rises to thousands, our original approach of sequentially analyzing a single global trace, as used by KOJAK, becomes increasingly constrained due to the large number of events. The biggest impediment to achieving scalability is the lack of processing power and memory available to a sequential program. Merging large numbers of process-local trace files into a single global trace file creates another bottleneck.

In this article, we describe how the pattern search can be accomplished in a more scalable way by exploiting both distributed memory and parallel processing capabilities available on the target system. Instead of sequentially analyzing a single global trace file, we analyze separate process-local trace files in parallel by *replaying* the original communication on as many processors 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 previously intractable scales.

The general concept of the parallel analysis and our prototypical implementation, which we call Scalasca, has been outlined in [9]. Here, we focus on Scalasca's parallel trace-analysis architecture, describe recent enhancements resulting in significantly reduced overall analysis time and substantially increased scalability, and provide evidence that the wait states we are looking for are manifested as critical performance problems at larger scales.

We start our discussion in Section 2 with a review of related work including our previous sequential design, followed by a description of our trace analyzer's new parallel architecture in Section 3, where we provide some background on trace generation, discuss the current parallel pattern-analysis scheme, and compare it to our initial prototype. In Section 4, we give experimental results that show scalability for up to 65,536 processes and substantial improvements in comparison to the initial parallel version. Afterwards, we demonstrate the relevance of our performance-analysis method using a real-world fluid dynamics application in Section 5. Finally, in Section 6 we discuss current limitations, outline potential remedies, and describe our plans for more advanced analyses.

2 Related and Prior Work

Event tracing is a well-accepted technique for postmortem performance analysis of parallel applications. Time-stamped events, such as entering a function or sending a message, are recorded in a memory buffer at runtime, later written to one or more files, and analyzed afterwards with the help of software tools. For example, graphical trace browsers, such as Jumpshot [34], Paraver [16], or Vampir [21], allow the fine-grained investigation of execution behavior using a zoomable time-line display. Whereas profiling summarizes performance metrics such as execution time or hardware event counts, tracing preserves both the spatial and temporal relationships between individual runtime events. Hence, tracing is the performance method of choice when the interaction between different processes or the temporal evolution of a performance metric needs to be examined.

In spite of its strength, the use of tracing is often limited by the large amount of data being generated and the size of the resulting files [27]. Users are typically confronted by problems such as the following:

- Excessive storage requirements, especially during trace generation and in-memory analysis
- Intrusion when flushing event data buffers to disk at runtime
- Costly file I/O when merging many potentially large process-local trace files into a global file (if the analysis tool requires a single global trace file)
- Long processing times during analysis
- Failure, extended response times, and insufficient size of graphical displays

In most cases, tracing can only be applied to relatively short execution intervals. Even a few minutes of program runtime can easily result in several megabytes of trace data per process. In general, the size of an event trace is influenced by two dimensions: the number of processes (width) and the number of events per process (length). Although we are also investigating solutions to handle longer traces, the work presented here primarily addresses wide traces.

So far, a number of approaches have been developed to either avoid or improve the handling of large traces. Freitag et al. [7] reduce the trace length of OpenMP applications by avoiding the recording of repetitive behavior in the first place. The underlying scheme is based on a dynamic periodicity detection algorithm, which is applied locally and therefore assumes structural similarities between concurrent control flows. The scalable visualization of inevitably long traces can be facilitated by dividing the trace file into frames, as in the SLOG format by Wu et al. [32], to represent intervals that can be separately loaded and analyzed. Moreover, unlike common linear storage schemes for event data, a tree-based main memory data structure called *compressed Complete Call Graphs* (cCCGs) developed by Knüpfer et al. [14] allows potentially lossy compression of long traces while observing previously specified deviation bounds. Furthermore, Noeth et al. [22] apply section descriptors to perform both intra-node and inter-node compression, thus addressing both long and also wide traces. Although their scheme is able to reduce traces from applications employing more regular communication patterns to near constant size independent of the number of nodes, it is not suitable for our analysis because it ignores the interval length between individual events. Whether a more recent extension that aims at retaining an approximation of the relative event distance by introducing delta-time histograms [23] would be more suitable remains to be evaluated. Finally, Labarta et al. [15] describe a variety of scalability-enhancing techniques applied in the Paraver project, covering all steps of the trace analysis from instrumentation through post-processing to

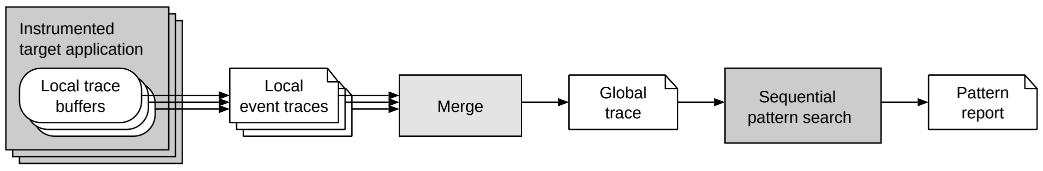


Fig. 1. KOJAK's sequential analysis workflow.

visualization. One key method is a system of filters that performs a stepwise transformation of larger traces into smaller ones both by eliminating dispensable features as well as by summarizing unnecessary details using a mechanism they call soft counters.

In the KOJAK project [28], we developed the idea of searching event traces of parallel applications for characteristic execution patterns indicating wait states, such as a receiver being blocked while waiting for a message to arrive. The KOJAK toolset offers a rich collection of patterns related to various programming models including MPI 1 + 2, OpenMP, and SHMEM. Figure 1 shows the sequential trace-analysis workflow enacted by KOJAK. Traces are written from the collection buffer of each process into process-local files, which are then merged according to event timestamp order to produce a chronologically ordered global trace that the sequential trace analyzer can handle. This is a notable bottleneck in the combined measurement and analysis process, which is extremely sensitive to file-system performance. During the sequential pattern search, the global trace file is traversed from beginning to end. To allow an efficient search despite the available memory usually being too small to hold the entire trace data, the search component exploits the locality of event accesses, following a sliding window approach [26]. While sequential access to a global event trace simplifies the detection of patterns involving concurrent events, the relative size of the search window shrinks in comparison to the overall trace size as the number of target-application processes is increased, making it more likely that locality of event accesses cannot be preserved. Our experiences show that the sequential model becomes impractical for analyzing traces from more than a few hundred processes [9]. In comparison, the parallel model proposed in this article offers two major advantages. First, since the process-local traces are accessed in parallel, there is no need for merged trace files. Second, the memory and processing power available to the parallel analysis scales linearly with the number of application processes, allowing analyses at much larger scales.

The parallel pattern search was inspired by the distributed trace analysis and visualization tool Vampir Server by Brunst et al. [6], which provides parallel trace access to non-merged traces, albeit targeting a 'serial' human client in front of a graphical time-line browser as opposed to fully automatic and parallel trace analysis. In earlier work, Miller et al. [18] already used a distributed algorithm operating on multiple local trace data sets in the parallel performance system IPS-2 to determine the critical path responsible for the length of application execution.

As an alternative to detecting wait states by comparing timestamps of concurrent events, as advocated here, Vetter [25] extracts the occurrence of wait states from event traces using machine-learning techniques. He traces individual message-passing operations and then classifies each instance using a decision tree, which has been previously trained for a particular hardware/software configuration. Our approach, in contrast, draws conclusions exclusively from runtime information and does not require any platform-specific training prior to analysis. Alternative methods of identifying wait states in MPI applications that do not require event tracing at all include intercepting MPI-internal events (e.g., the actual begin of message receipt) exposed by a call-back interface such as MPI PERUSE [20], replacing non-blocking communication operations with their blocking counterparts and actively polling in between [3], or piggybacking timestamps [19]. While relieving the user from the burden of collecting space-intensive event traces, they may introduce other tradeoffs, such as requiring a non-standard MPI implementation, incurring additional measurement intrusion including expensive runtime bookkeeping when trying to detect more complex patterns, or demanding a global clock. Also, they do not share the potential of traces for more advanced analyses such as performance prediction via trace-based simulation [11].

3 Parallel Trace-Analysis Architecture

Scalasca is a parallel trace-analysis tool specifically designed for large-scale systems. It searches event traces of parallel applications for characteristic execution patterns indicating various types of wait states. A distinctive feature of Scalasca is that it achieves a high degree of scalability by analyzing the trace data in parallel. Although Scalasca also offers call-path profiling as an alternative performance-analysis mechanism, we concentrate our discussion exclusively on its trace-analysis capabilities.

The current tracing infrastructure allows the measurement and analysis of MPI 1 applications written in C/C++ and Fortran on a wide range of HPC platforms [31]. Support for OpenMP and hybrid codes is already in progress. Figure 2 illustrates the basic trace-analysis workflow and the role of the different components in transforming raw measurement data into knowledge of application execution behavior. First, the instrumented application is linked to the measurement library so that, when running it on the parallel machine, every application process generates its local portion of the trace data in EPILOG format [29]. After program termination, Scalasca loads the trace files into main memory and analyzes them in parallel using as many processors as

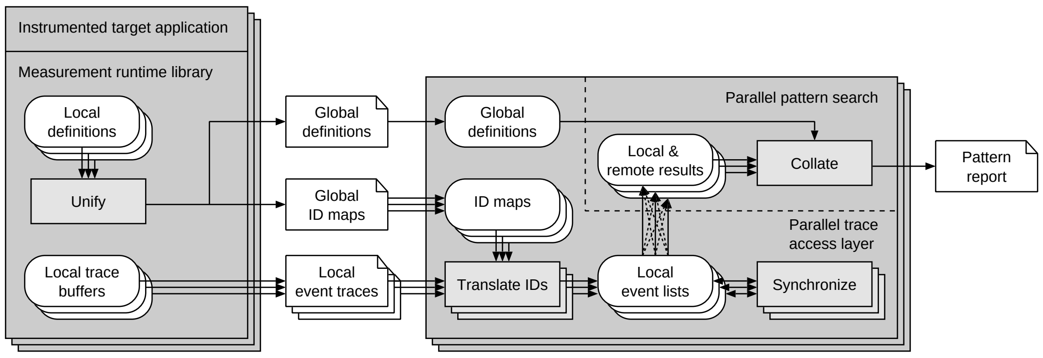


Fig. 2. Schematic overview of Scalasca's parallel analysis workflow. Gray rectangles denote programs, white rectangles with the upper right corner turned down denote files, and bubbles denote data objects residing in memory. Stacked symbols denote multiple instances of programs, files or data objects running or being processed in parallel.

used for the target application itself. In a preprocessing step, references to objects such as code regions or communicators that appear in event records are made consistent across all processes, after unified object definitions and their global identifiers have been computed at measurement finalization. Moreover, on systems without a global clock, the timestamps must be retroactively synchronized to allow precise measurements of temporal displacements between concurrent events. During the actual analysis, Scalasca searches for characteristic patterns indicating wait states, classifies detected instances by category, and quantifies their significance. The results are collated into a single pattern-analysis report similar in structure to an ordinary call-path profile but enriched with higher-level communication and synchronization inefficiency metrics. In the remainder of this section, we first provide background on the generation of trace files, before we review our parallel trace-analysis approach in more detail, emphasizing (i) the parallel event access during analysis, (ii) the parallel analysis algorithm, and (iii) the collation of local analysis results.

3.1 Generation of Trace Files

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

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, this step is performed in an automated fashion during compilation and linkage. In view of the I/O bandwidth and storage demands of tracing on large-scale systems, and specifically the perturbation caused by processes flushing their trace data to disk in an unsynchronized way while the application is still running, it is generally desirable to limit the amount of trace data per application process so that the size of the available memory is not exceeded. This can be achieved via selective tracing, for example, by recording events only for code regions of particular interest or by limiting the number of timesteps during which measurements take place.

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. We will nonetheless present overhead numbers for the real-world example given in Section 5. Moreover, if the communication frequency and with it the event frequency rises with increasing numbers of processes, then the observable dilation might also become a matter of scale.

Definition unification. Measured event data refers to objects such as source-code regions, call paths, or communicators. Motivated by the desire to minimize storage requirements and avoid redundancy in traces, events reference these objects using identifiers, while the objects themselves are defined separately. To avoid extra communication between application processes during measurement acquisition, each process may use a different local identifier to denote the same object. However, to establish a global view of the program behavior during analysis, a global set of unique object definitions must be created and local identifiers replaced with global identifiers that are consistent across all processes. This procedure is called *unification*. Separate collection buffers on each process are used for definition and event records, avoiding the need to extract the definitions from a combined trace later. At measurement finalization, each rank in turn sends its definition buffers to rank zero for unification into a set of global definitions and an associated identifier mapping. Although our current unification algorithm is predominantly sequential, the distributed design takes advantage of message communication to facilitate the exchange of object definitions and the generation of mapping information while reducing expensive file I/O that would be otherwise prohibitive. Tree-based unification of definitions might be beneficial, and such an approach using MR-Net [24] is currently under investigation. The global definitions and mappings are then written to two files, alongside the files for the dumped contents of each trace buffer. The two auxiliary files are subsequently needed by the post-mortem trace analyzer to translate local object identifiers in event records to global identifiers used during analysis. While the sizes of process-local definition buffers, unified global definitions, and associated mappings are highly dependent on application instrumentation and measurement configuration, for the SMG2000 experiments considered later in Section 4 the size of each process-local definition buffer was around 20 kB, and for 65,536 processes the resulting global definitions and mappings consumed almost 5 MB and 82 MB, respectively. Although the mapping data grows linearly with the number of processes, the mappings for each process can be written immediately to file after generation such that capacity is required for only one set.

3.2 Parallel Event Access

The low-level interface available for reading EPILOG traces offers only rudimentary support for complex analysis tasks. First, the interface follows a sequential access model. In addition, the correct interpretation of certain events may require knowledge of related events. For example, to save space, records of exit events may not specify the region that was left. Instead, this information must be obtained from the matching enter event, assuming that process-local enter and exit events form a correct parenthesis expression. While necessary for the correct understanding of individual events, finding matching events is also an important prerequisite for the pattern analysis itself because patterns are typically identified along a path of related events. To simplify the analysis logic required for the pattern search, the trace analyzer accesses the trace file through a high-level interface, which is provided as a separate abstraction layer [8] between the parallel pattern search and the raw trace data stored on disk (Figure 2, bottom right). Implemented as a C++ class library called PEARL, this layer offers random access to individual events as well as abstractions that help identify matching events.

To maintain efficiency of the trace analysis process as the number of application processes rises, PEARL follows a parallel trace access model, operating on multiple process-local trace files instead of a single global trace file. The main advantage of the parallel model is that it allows the size of the target system to be exploited by scaling the amount of memory and the number of processors available for the analysis with the number of application processes to be analyzed. As a consequence, the analysis algorithms and tools based on PEARL are usually parallel applications in their own right with expected scalability improvements compared to serial versions. Here we present a specific application of PEARL to automated trace file analysis, although its utility is far more general [4,11].

Usage model. Establishing truly parallel abstractions, such as complete messages or collective operation instances, that is, gathering all events belonging to these structures, requires the matching of corresponding events across several processes, which may incur costly communication. To minimize the communication overhead, the primary usage model of PEARL is that of a *replay-based* analysis. The central idea behind a replay-based analysis is to reenact the target application's communication based on the trace information so that each communication operation can be analyzed using an operation of similar type. For example, to analyze a message transfer in point-to-point mode, the required event data is exchanged using a single point-to-point operation. Exploiting MPI messaging semantics, this model offers the advantage that matching related communication events occurs automatically while replaying the corresponding MPI operations.

Event storage. The usage model of PEARL currently assumes a one-to-one mapping between analysis and target-application processes. That is, for every process of the target application, one analysis process responsible for the trace data of this application process is created. Data exchange in PEARL is accomplished via MPI. We currently require that the amount of trace data per application process does not exceed the size of the memory available to a single process on the target system. This offers the advantage that during analysis PEARL can keep the entire event trace in main memory, which allows performance-transparent access to individual events, but also limits the trace size per process and may necessitate measures such as coarsening the instrumentation or restricting the measurement to selected intervals.

Loading traces. To the programmer, PEARL offers classes to represent process-local traces, events, and objects referenced by those events (e.g., regions, communicators). When instantiating a trace object, PEARL reads the trace data into memory while unifying all object references appearing in event records based on the local-to-global identifier mapping mentioned in Section 3.1. This ensures that event instances created from event records point to the correct objects. After unification, all objects carry a unique numerical identifier that is consistent across all processes and that facilitates the exchange of events and their object references between different analysis processes.

Timestamp synchronization. To allow accurate trace analyses on systems without globally synchronized timers, we also provide the ability to synchronize inaccurate timestamps postmortem. Linear interpolation based on clock offset measurements during initialization and finalization of the target program already accounts for differences in offset and drift, assuming that the drift of an individual processor is not time-dependent. This step is mandatory on all systems without a global clock, such as Cray XT and most PC or compute blade clusters. However, inaccuracies and drifts that vary over time can still cause violations of the logical event order that are harmful to the accuracy of our analysis. For this reason, such violations can be compensated by shifting communication events in time as much as needed to restore the logical event order while trying to preserve the length of intervals between local events [4]. This logical synchronization is currently optional and should be performed if the trace analysis reports (too many) violations of the logical event order.

Event access. Event access occurs through event objects that allow access to all possible event attributes and that expose iterator semantics for navigating through the local trace. In addition to the iterator functionality, event objects also provide links between related events, which are called pointer attributes. As a consequence of the parallel in-memory event storage, pointer attributes can both point forward and backward, but not to remote events. Currently, there are pointer attributes to identify the enter and exit events of the enclosing region instance (i.e., *enterptr* and *exitptr*). Whereas *enterptr* points backward, *exitptr* points forward in time. These two pointer attributes can be used, for example, to determine the duration of the communication operation (i.e., region instance) for a given communication event. Another special event attribute identifies the call path of an event by providing a pointer into the global call tree. In this way, PEARL applications can easily determine whether events have equivalent call paths, a feature used to automatically associate patterns with the call paths causing them.

Exchanging event data between processes. To facilitate inter-process analysis of communication patterns, PEARL provides means to conveniently exchange one or more events between processes. Remote events received from other processes are represented by remote event objects, which are similar to local event objects, but without iterator semantics and attributes pointing to other remote events, since we do not have full access to the remote event trace. There are generally two modes of exchanging events: point-to-point and collective. Point-to-point exchange allows a remote event object to be created at the destination process with arguments specifying the source process, a communicator, and a message tag. To complete the exchange, the corresponding source process invokes a send method on the local event object to be transferred. Collective exchange works in a similar fashion. Moreover, the exchange of multiple events can be accomplished in one batch by first collecting local events in an event set on the sender's side and then instantiating a remote event set on the receiver's side by supplying message parameters to the constructor. Each event stored in an event set is identified by a numeric identifier that can be used to assign a role, for example, to distinguish a particular constituent of a pattern. To reduce the amount of data being transferred, event sets can transparently assign multiple roles to a single event.

3.3 Parallel Analysis Algorithm

The task of the parallel trace analysis is to locate patterns of inefficient behavior and to quantify associated waiting times separately for every call path and process. A pattern is typically composed of multiple potentially concurrent constituent events with certain constraints regarding their relative order. The associated waiting time, which we call the *severity* of the pattern, is usually calculated as the temporal difference between selected constituents. To accomplish the search, the parallel analyzer component traverses the local traces in parallel from beginning to end while exchanging information at synchronization points of the target application using the infrastructure described in the previous subsection. That is, whenever an analysis process sees events related to communication or barrier synchronization, it engages in an operation of a similar type with those analysis processes that see the corresponding source or destination events. Currently, the parallel analysis employs one analysis process per local trace file, which results in an equal number of application and analysis processes. This allows the user to run it immediately after the target application within the same batch job, thereby avoiding a second wait in the batch queue. In future versions, we plan to also allow a smaller number of analysis processes, which can be useful if the analysis is carried out on a different machine.

In this subsection, we describe how the parallel replay can be used to search for complex patterns of inefficient behavior at larger scales. A full list of the patterns the parallel analyzer supports can be found online [13], illustrated with explanatory diagrams. This set includes the full range of MPI 1 patterns provided by KOJAK's sequential analyzer [28], with the exception of *Late Receiver / Wrong Order*, which, however, is rarely significant in practice. Below, we illustrate the parallel analysis mechanism on a representative subset of these patterns, which is shown in Figure 3.

3.3.1 Point-to-Point Communication

As an example of inefficient point-to-point communication, we first consider the so-called *Late Sender* pattern (Figure 3(a)), where a receiver is blocked while waiting for a message to arrive, that is, the receive operation is entered by the destination process before the corresponding send operation has been entered by the source process. The waiting time lost as a consequence is the time difference between the enter events of the two MPI function instances that enclose the corresponding send and receive events.

Figure 4(a) illustrates the parallel detection algorithm for this pattern. During the parallel replay, the detection is triggered by the communication events on both sides (i.e., the send and receive event). Whenever an analysis process finds a send event, a message containing this event as well as the associated enter event is created, after the latter has been located via the backward-pointing *enterptr* attribute. This message is then sent to the process representing the receiver using a point-to-point message. When the destination process reaches the receive event, the above-mentioned message containing the *remote constituents* of the pattern is received. Together with the locally available constituents (i.e., the receive and the enter event), a Late Sender situation can be detected by comparing the timestamps of the two enter events and calculating the time spent waiting for the sender. The waiting time is ascribed to the

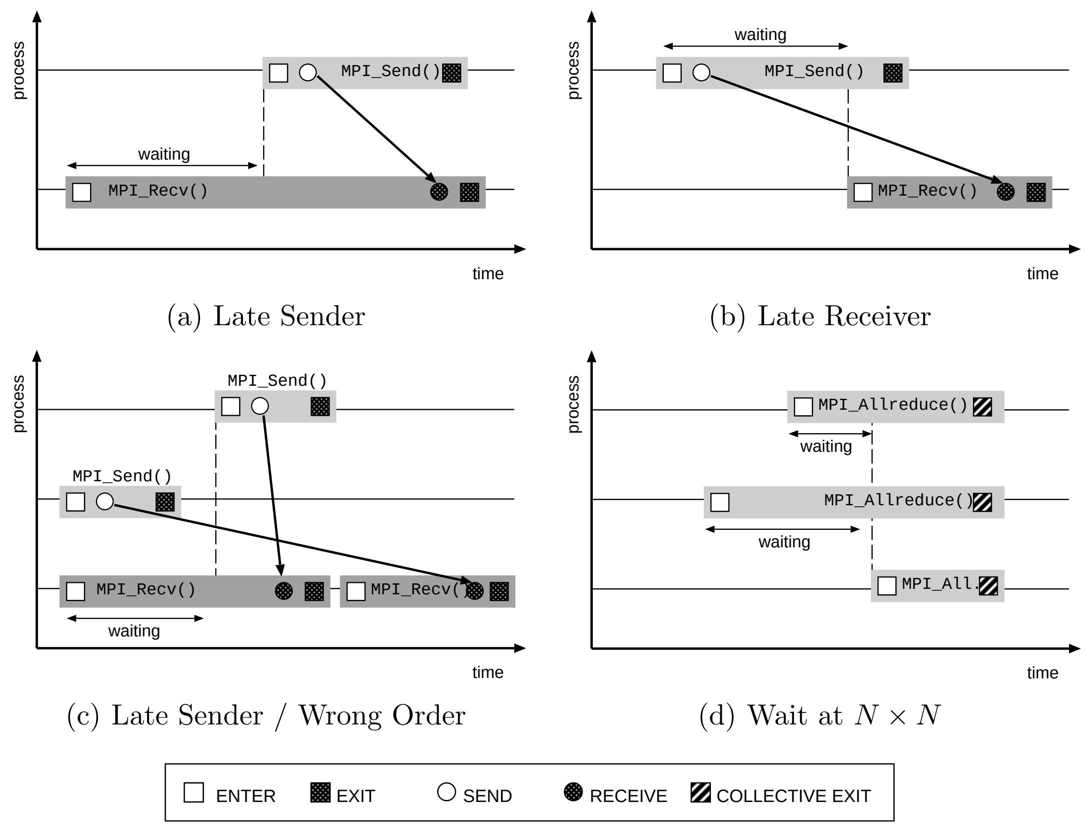
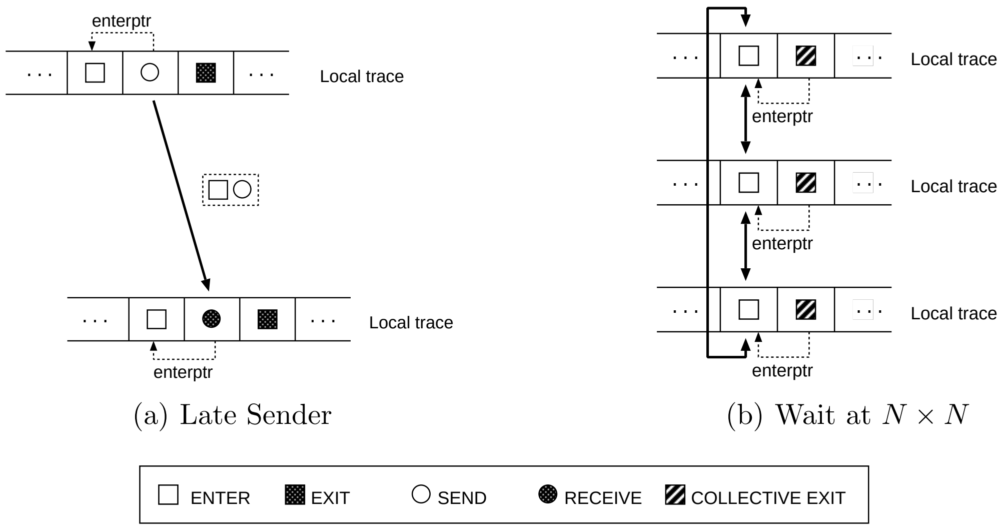


Fig. 3. Patterns of inefficient behavior. The combination of MPI functions used in each of these examples represents just one possible case. For instance, the synchronous receive operation in pattern (a) can be replaced by an immediate receive followed by a wait operation. In this case, the waiting time would occur during the wait operation.



(a) Late Sender (b) Wait at $N \times N$

Fig. 4. Parallel pattern detection.

call path of the send operation using the above-mentioned call-path attribute. The correct matching of send and receive events is accomplished by replaying the message traffic with equivalent tag and communicator information available in the event trace, ensuring that the messages are received in the original order. In addition to being more scalable, the replay-based approach is also resilient with respect to insufficiently synchronized clocks because it is still able to match send and receive events correctly even if the relative chronological order between the two events is reversed.

Conversely, the *Late Receiver* pattern (Figure 3(b)) describes a sender that is blocked while waiting for the receiver when a rendezvous protocol is used. This can happen for several reasons. Either the MPI implementation is working in synchronous mode by default, or the size of the message to be sent exceeds the available MPI-internal buffer space and the operation is blocked until the data is transferred to the receiver. The behavior is similar to an `MPI_Ssend()` waiting for message delivery. Valid instances of this pattern must satisfy the following condition: the receive operation must have started after the send operation has begun but before it has ended. Although this criterion does not reliably prove that the sender was blocked, it is nevertheless a necessary condition and it is the strongest that can be proved on the basis of our current event model. A detailed discussion of the performance problem related to this pattern can be found in [10]. The parallel detection of the Late Receiver pattern follows a strategy similar to *Late Sender.* In this case, however, both the enter and the exit event enclosing the send event are transferred to the destination process. The exit event is located via the forward-pointing *exitptr* attribute emanating from the send event. After the arrival of the two remote constituents, the destination process compares their timestamps to the timestamp of the local enter event that marks the begin of the receive operation.

As shown in the figure, the severity of the pattern is the difference between the two enter events. Although the severity is calculated by the receiver, it is attributed to the sender. To avoid the additional overhead of transferring the calculated waiting time back to the sender every time this pattern occurs, it is temporarily stored as a *remote result* by the receiver. The distinction between local and remote results is resolved at the end of the analysis, as explained in Section 3.4.

By contrast, detecting the *Late Sender / Wrong Order* pattern (Figure 3(c)) is more difficult. This pattern describes a receiver waiting for a message, although an earlier message is ready to be received by the same destination process (i.e., message received in wrong order). The detection requires a global view of the messages currently in transit (i.e., a global message queue), which is not available in our parallel implementation. As a substitute, each analysis process keeps track of the n last local occurrences of the Late Sender pattern using a ring buffer. If a receive event is encountered during the replay, we compare the timestamps of the matching send event with those of the buffered Late Sender occurrences. If the Late Sender's send operation starts after the send event associated with the current receive, the Late Sender instance is classified as a wrong-order situation and removed from the buffer. Although this approach does not guarantee that all occurrences of this pattern will be found, empirical results suggest that the coverage of our method is sufficient in practice.

To avoid sending redundant messages while executing the detection algorithms for the different performance problems related to point-to-point communication, we exploit specialization relationships between patterns and reuse results obtained on higher levels of the hierarchy. This is implemented using an event notification and call-back mechanism similar to the publish-and-subscribe approach [30] used in KOJAK and requires every point-to-point message to be replayed only once, even if it contributes to multiple patterns.

3.3.2 Collective Communication

The second type of communication operation we consider is MPI collective communication. As an example of a related performance problem, we discuss the *Wait at N×N* pattern (Figure 3(d)), which quantifies the waiting time due to the inherent synchronization in n-to-n operations, such as `MPI_Allreduce()`. A schematic view of how this pattern can be handled in parallel is given in Figure 4(b). While traversing their local trace data, all processes involved in a collective operation instance will eventually reach their collective exit event marking the end of this instance. After verifying that it belongs to an n-to-n operation by examining the associated region identifier, the analyzer determines the latest of the corresponding enter events via a maximum reduction using an `MPI_Allreduce()` operation. After that, each process calculates the local waiting time by subtracting the timestamp of the local enter event from the maximum timestamp obtained through the reduction operation. The group of ranks involved in the analysis of the collective operation is easily determined by reusing the communicator of the original collective operation.

Similar algorithms can be used to implement patterns related to 1-to-n, n-to-1, barrier and scan operations. For n-to-n and barrier operations, the analyzer also calculates asymmetries that occur when leaving the operation. As with point-to-point operations, a single MPI call per collective operation instance is used to determine all the associated waiting times, even if they belong to different patterns.

3.4 Collation of Analysis Results

Whenever an analysis process finds a pattern instance, it measures the waiting time incurred as a result of this pattern (i.e., its severity) and accumulates this value in a [pattern, call path] matrix. At the end of the trace traversal, the local results are merged into a three-dimensional [pattern, call path, process] structure characterizing the whole experiment, which is then written to an XML file.

To make this most efficient, we first have to separate truly local waiting times from remote waiting times that have only been locally accumulated. As an example of the latter, let us revisit the Late Receiver pattern. The waiting time occurs at the sending process, but it is calculated and stored at the receiving process. However, differentiating between local and remote waiting times complicates the collation because the rank to which the time belongs would then have to be made explicit. Hence, remote times are distributed to their original locations using point-to-point messages before initiating the merge procedure. Once all processes store only their truly local waiting times, a designated writer process prepares the report header before it gathers the aggregated metrics for each call path from each process and appends these to the report body. Since the size of the report may exceed the memory capacity of the writer process, the report is created incrementally, alternating between gathering and writing smaller subsets of the overall data. The MPI gather operation used for this purpose allows this procedure to take advantage of efficient tree-based algorithms employed in most MPI implementations.

Since implementing an initial prototype [9] of the parallel trace-analysis scheme, we have applied a number of optimizations and extensions, substantially enhancing performance and scalability in comparison to the initial version. Below we give a brief overview of the most important changes, while deferring a quantitative evaluation to Section 4.

First, in the initial version of our parallel design, each process wrote its locally collected results including remote results to a separate XML file, again incurring costly I/O when thousands of files had to be created simultaneously. The local results were then merged into a single global result file using a sequential postprocessing tool, significantly prolonging the total analysis time. The present version employs the much more efficient collation scheme described in Section 3.4. Taking advantage of MPI collective communication while directly writing the gathered results to a single report file, the new collation mechanism is mainly responsible for the performance improvements in comparison to our initial parallel prototype. Moreover, after substituting non-blocking sends for blocking ones and reducing the number of collective replays required for every collective operation recorded in the trace to one, the replay engine now runs significantly faster. To increase the engine's processing capacity, the memory footprint of event objects has been scaled down by calculating certain event attributes on the fly and customizing the memory management. Furthermore, logical timestamp correction capabilities have been added to deal with cases where non-constant clock drifts render the linear method insufficient when applied in isolation [4].

Finally, the optimized trace analyzer is now supported by a more efficient measurement system. Eliminating the need for temporary local definition files and combining rank-local mappings into a single file has reduced the number of auxiliary files created by the measurement system in addition to the actual trace files from $2 * |ranks| + 1$ to just 2 extra files, offering improved trace generation performance. Additionally, serial optimizations have cut the time required for identifier unification and map creation at measurement finalization by a factor of up to 25 (in addition to savings from creating only two global files).

4 Scalability

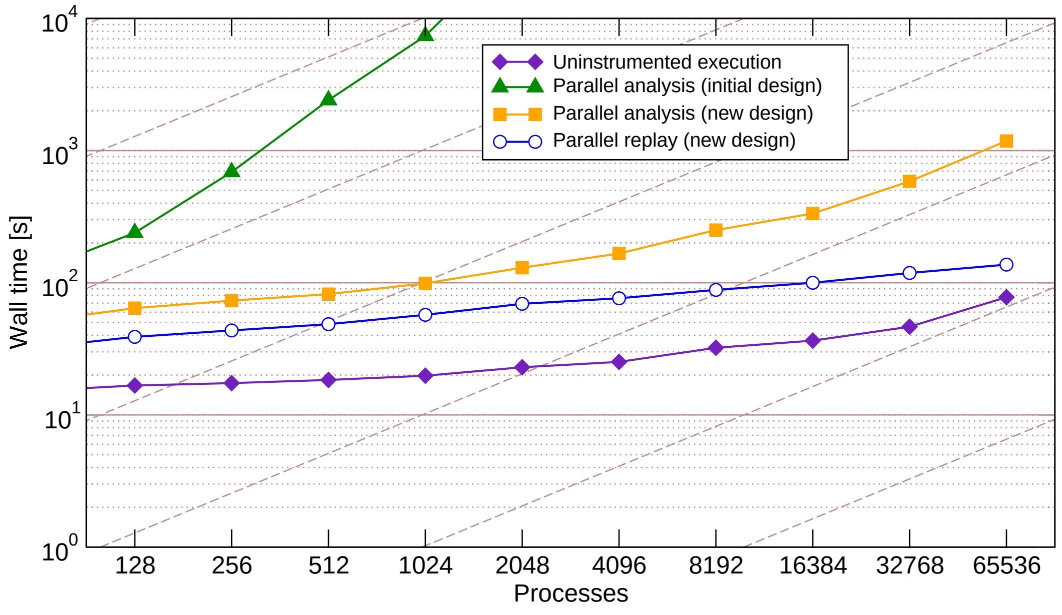
To evaluate the efficiency of the parallel trace analysis, a number of experiments with our current prototype implementation have been performed at a range of scales and compared with the initial parallel version. Measurements were taken on the IBM Blue Gene/P system (JUGENE) at the Jülich Supercomputing Centre, Germany, which consists of 16,384 quad-core 850 MHz PowerPC 450 compute nodes (each with 2 GB of memory), 152 I/O nodes, and p55A service and login nodes each with eight dual-core 1.6 GHz Power5+ processors [12]. The system was running the V1R2 software release with the GPFS parallel file system. A dedicated partition consisting of all of the compute nodes in virtual-node mode was used for the parallel analyses, whereas serial components ran on the lightly loaded login node. Two applications with quite different execution and performance characteristics were selected for detailed comparison.

The ASC benchmark SMG2000 [2] is a parallel semi-coarsening multigrid solver that uses a complex communication pattern. The MPI version performs a large number of non-nearest-neighbor point-to-point communication operations (but only a negligible number of collective communication operations) and can be considered to be a stress test for the memory and network subsystems of a machine. Applying a weak scaling strategy, a fixed $64\times64\times32$ problem size per process with five solver iterations was configured, resulting in a nearly constant application runtime as additional processors were used. Because the number of events traced for each process increases with the total number of processes, the aggregate trace volume increases faster than linearly.

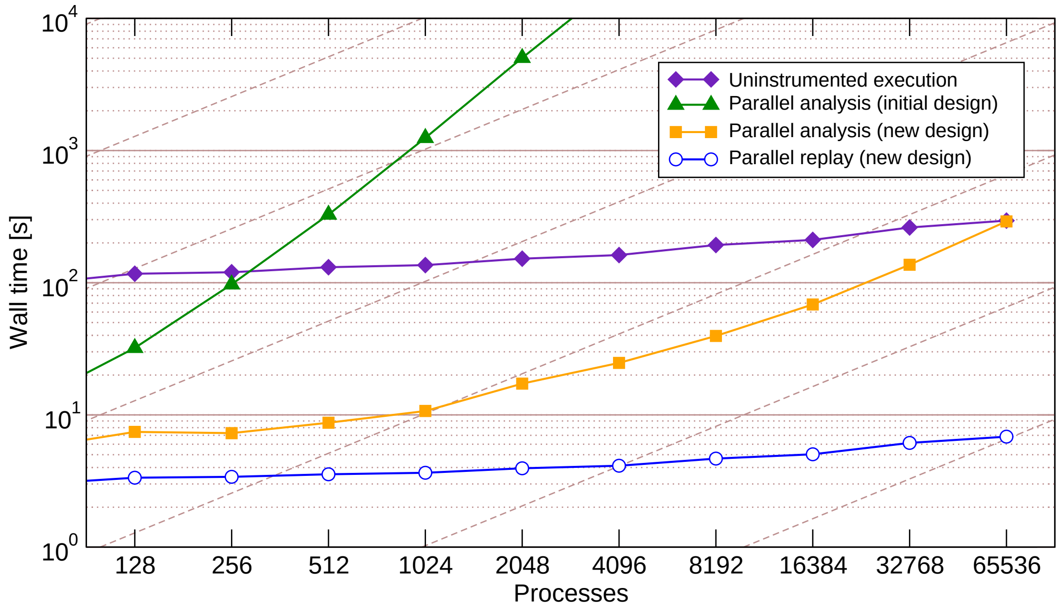
The second example, the benchmark code SWEEP3D [1], performs the core computation of a real ASCI application and solves a 1-group time-independent discrete ordinates (Sn) 3D Cartesian geometry neutron transport problem. To exploit parallelism, SWEEP3D maps the three-dimensional simulation domain onto a two-dimensional grid of processes. The parallel computation implements a pipelined wavefront process that propagates data along diagonal lines through the grid using MPI point-to-point messages. Again, we used a constant $32 \times 32 \times 512$ problem size per process. In contrast to SMG2000, the amount of trace data per process is independent of the total number of processes.

Figure 5 charts wall-clock execution times for the uninstrumented applications and the analyses of trace files generated by instrumented versions with a range of process numbers on JUGENE. The 9-fold doubling of the number of processes necessitates a log–log scale to show the corresponding range of times, to the extent that they remain practical. The figure shows the total time needed for the parallel analysis when using both the initial and the new design. The total analysis time includes loading the traces, performing the replay, and collating the results. It can be seen that the total analysis time has been reduced to below 20 minutes in both cases. The substantial improvement in comparison to our initial parallel prototype is largely due to the new collation scheme. Because file I/O can be subject to considerable variation depending on overall system load, the time needed for the replay itself without file I/O is also shown separately for the new version.

While the set of compressed execution traces from 65,536 SWEEP3D processes reached 79 GB aggregate size (13E9 events in total), the corresponding execution traces from 65,536 SMG2000 processes were 1060 GB (over 178E9 events in total). File I/O commands increasing proportions of the analysis



(a) SMG2000.



(b) SWEEP3D.

Fig. 5. Application execution times and trace analysis times using the the initial parallel version and the improved parallel version for up to 65,536 processes on Blue Gene/P. Linear scaling is represented by dashed lines.

time. However, it is worth noting that with only one I/O node per 512 processor cores, our test system does not offer the most generous I/O configuration possible on Blue Gene/P. A comparison with data from 16,384 processes obtained on a Cray XT3/4 attached to a Lustre file system showed that a more favorable replay-to-file I/O ratio is possible.

By contrast, the actual procedure of replaying and analyzing the event traces exhibits smooth scaling behavior up to very large configurations. Although the original prototype implementation demonstrated the same scalability, the current version is twice as fast and requires 25% less memory. Because of its replay-based nature, the time needed for this part of the analysis depends on the communication behavior of the target application. Since communication is a key factor in the scaling behavior of the target application as well, similarities can be seen in the way both curves evolve as the number of processes increases. Depending on the computation and communication characteristics of the target application, the replay can be either faster (SWEEP3D) or slower (SMG2000) than the application itself. Notably, the total time for the new analysis approach scales much better and is orders of magnitude faster than the original parallel analysis prototype even at modest scales, making it possible to examine traces at previously intractable scales in a reasonable time.

5 Example

In this section, we show that the problems Scalasca diagnoses do indeed appear in real applications and that they can have significant performance impact especially at larger scales. As an example, we consider the XNS computational fluid dynamics code being developed at the Chair for Computational Analysis of Technical Systems at RWTH Aachen University. This engineering application can be used for effective simulations of unsteady fluid flows, including microstructured liquids, in situations involving significant deformations of the computational domain. The algorithm is based on finite-element techniques on irregular three-dimensional meshes using stabilized formulations and iterative solution strategies [5]. The XNS code consists of more than 32,000 lines of Fortran90 in 66 files. It uses the EWD substrate library, which fully encapsulates the use of BLAS and communication libraries, adding another 12,000 lines of mixed Fortran and C within 39 files. Here, we present results obtained from an already tuned version of the code [33].

Performance was studied on the IBM Blue Gene/L system JUBL at the Jülich Supercomputing Centre, consisting of 8,192 dual-core 700 MHz PowerPC 440 compute nodes (each with 512 MB of memory) and 288 I/O nodes. The system was running the V1R2 software release with the GPFS parallel file system configured with 4 servers. The test case consisted of a three-dimensional space-time simulation of the MicroMed DeBakey axial blood pump, which was executed in virtual node mode on a dedicated partition. This data set requires very high resolution simulation to accurately predict shear-stress levels and flow stagnation areas in an unsteady flow in such a complex geometry. From a comparison of simulation rates (and later analyses) for various numbers of timesteps, it was determined that the first timestep's performance was representative of that of larger numbers of timesteps, allowing the analysis to concentrate on simulations consisting of a single timestep.

Our observations using the Scalasca analysis are summarized in Figure 6 for a range of scales between 256 and 4096 processors. The problem size was kept constant across all configurations so that we could witness strong scaling. For each configuration, we provide the simulation timestep rate as an overall performance indicator as well as a breakdown of the total execution time into percentages corresponding to various communication modes and wait states. The columns to the left show the fraction of time spent in MPI function calls and the inherent waiting time in each configuration. The columns to the right split the MPI time further into different modes of communication (point-to-point and collective) and synchronization. For each mode, we give the overall percentage followed by relevant wait states caused by patterns discussed in Section 3.3: Late Sender for point-to-point communication, Wait at $N \times N$ for collective communication, and Wait at Barrier for collective synchronization,

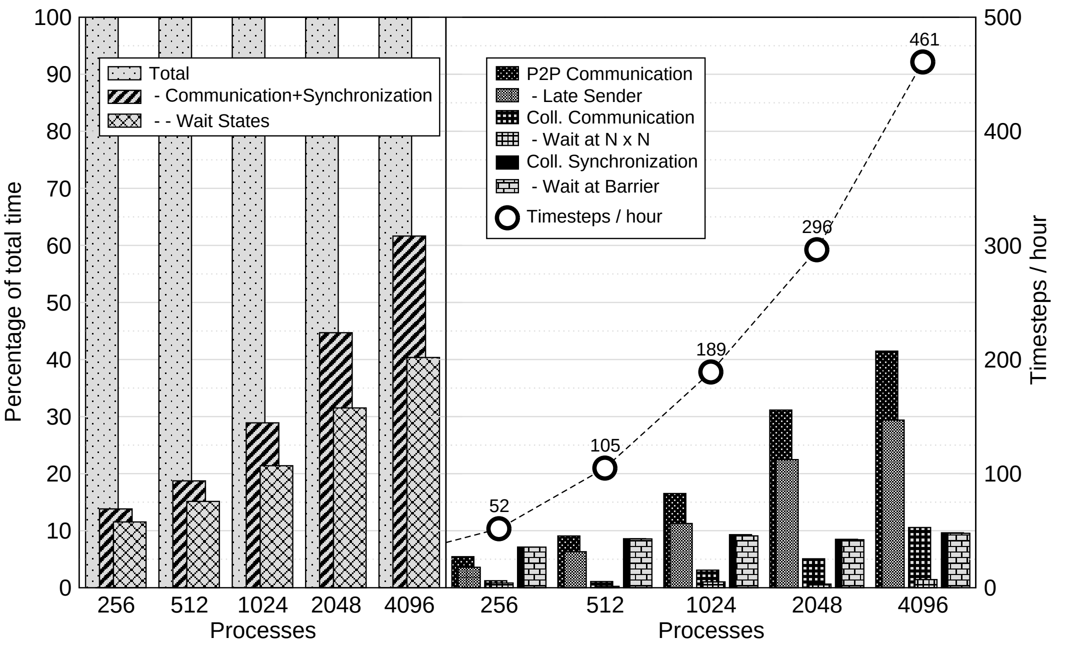
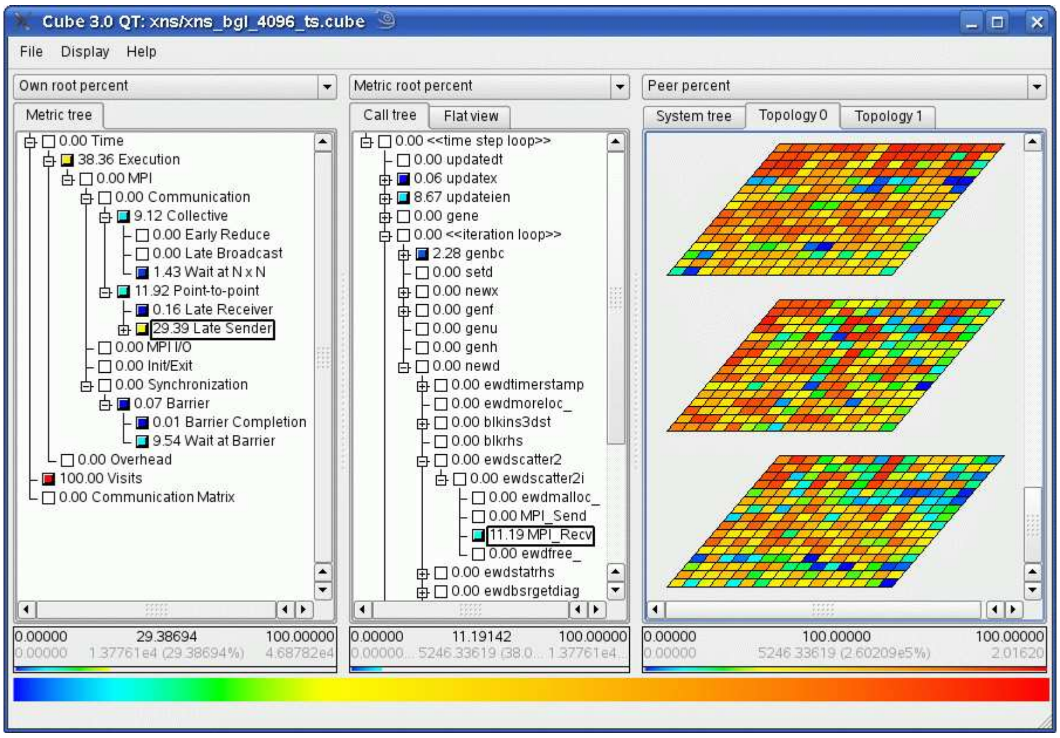


Fig. 6. Performance behavior of XNS at a range of scales from 256 to 4096 processors. The circles indicate the simulation timestep rate as an overall performance indicator (higher is better). The bars show the percentage the application spent in various communication modes including associated wait states.

which is essentially a barrier version of Wait at $N \times N$.

The application exhibits almost perfect scaling behavior with up to 512 processes. However, as the number of processes is raised further, the parallel efficiency is continuously degraded, although even at the largest configuration of 4096 processes a noticeable speedup can still be observed. Beyond 512 processes, as we can see, the communication and synchronization overhead grows steeply, with point-to-point communication being the dominating factor. Yet the primary result of our analysis is that the biggest fraction of this overhead is actually waiting time, in the case of 4096 processes amounting to roughly 40% of the total time and 65% of the time spent in MPI, illustrating that the wait states we are targeting can constitute principal performance properties at larger scales. That is, the saturation of speedup observed for XNS with higher numbers of processes is not only the result of communication demand growing with the number of processes, but also to a larger extent the consequence of the untimely arrival of processes at synchronization points, which can now be classified and quantified using our analysis technique.

With 4,096 processes, the execution time for an uninstrumented XNS single



```
0.00 Time
  38.36 Execution
    0.00 MPI
      0.00 Communication
        9.12 Collective
          0.00 Early Reduce
          0.00 Late Broadcast
          1.43 Wait at N x N
        11.92 Point-to-point
          0.16 Late Receiver
          29.39 Late Sender
      0.00 MPI I/O
      0.00 Init/Exit
      0.00 Synchronization
        0.07 Barrier
          0.01 Barrier Completion
          9.54 Wait at Barrier
  0.00 Overhead
100.00 Visits
0.00 Communication Matrix
```

```
0.00 <<time step loop>>
  0.00 updatedt
  0.06 updatex
  8.67 updateien
  0.00 gene
  0.00 <<iteration loop>>
    2.28 genbc
    0.00 setd
    0.00 newx
    0.00 genf
    0.00 genu
    0.00 genh
    0.00 newd
      0.00 ewdtimerstamp
      0.00 ewdmoreloc_
      0.00 blkins3dst
      0.00 blkrhs
      0.00 ewdscatter2
        0.00 ewdscatter2i
          0.00 ewdmalloc_
          0.00 MPI_Send
          11.19 MPI_Recv
          0.00 ewdfree_
      0.00 ewdstatrhs
      0.00 ewdbsrgetdiag
```

```
0.00000       29.38694              100.00000
0.00000   1.37761e4 (29.38694%)    4.68782e4

0.00000       11.19142              100.00000
0.00000...5246.33619 (38.0... 1.37761e4...

0.00000       100.00000             100.00000
0.00000   5246.33619 (2.60209e5%)   2.01620
```

Fig. 7. XNS Late Sender bottleneck with 4096 processes in `ewdscatter2i()`. The middle pane shows the distribution of Late Sender times across the call tree as percentage of the time spent in the timestep loop. The right pane visualizes how the time incurred by the selected call path is spread across the physical Blue Gene/L torus topology.

timestep simulation was 115 seconds, of which the timestep loop itself was 28 seconds. It took 692 seconds to collect an 82 GB trace (using a filter to eliminate unimportant small functions), and parallel analysis of the 2.9E9 traced events took a further 216 seconds (including 51 seconds for the replay).

The measurement dilation in the timestep loop caused by our instrumentation increases from a very small amount within measurement errors to about 14%, as the number of processes is raised from 256 to 4096. The increase can be explained with a growing fraction of communication, which represents the primary source of overhead, along with decreasing message sizes. Although in the 4096 case the 14% can already be regarded as significant, it is still reasonable in proportion to the percentage of waiting time identified. Even if the entire dilation was subtracted from the overall waiting time, the application would still spend one quarter of the time in idle mode.

The largest contributor to the overall waiting time is Late Sender. The screen shot of the Scalasca GUI in Figure 7 shows the analysis results for 4096 processes and locates a major source of this pattern in a call path leading to the function `ewdscatter2i()` (middle pane). The call tree has been re-rooted to display only the timestep loop. As the mapping of waiting times onto the physical Blue Gene/L torus topology illustrates (right pane), the distribution of wait states across different processes is rather irregular and is presumably an artifact of communication imbalance caused by the irregular mesh. Furthermore, it appears that a major fraction of the Late Sender time coincides with message receipt in the wrong order (not shown in the screen shot). Work to optimize the application based on the insights presented above is in progress.

6 Conclusion and Outlook

We have presented a parallel tool architecture for automatically detecting wait states in event traces of massively parallel applications. Instead of sequentially analyzing a single and potentially large global trace file, we analyze separate local trace files in parallel by performing a replay of the target application's communication behavior. By exploiting the distributed memory capacity and the parallel processing capabilities of the target system, the new trace-analyzer architecture has delivered satisfactory performance for message-passing applications with up to 65,536 processes, offering wait-state diagnoses at previously impractical scales. Our current analyzer prototype, which is a parallel application in its own right, is capable of detecting a variety of wait states related to the use of the MPI 1 parallel programming interface, which, as we have shown using a full-size engineering application, can present severe performance problems at larger scales. We plan to add support for additional parallel programming interfaces, such as OpenMP and MPI 2, in the future.

Although we were able to significantly improve the performance of our analysis in comparison to our initial prototype by employing a much more efficient collation scheme, file I/O remains the dominating factor. On systems such as Blue Gene/P that support the definition of memory segments persistent across jobs, the transfer of trace data from the application to the analysis could be accomplished at basically no cost, expanding the potential of our method to much larger scales. Alternatively, the analysis could occur immediately during measurement finalization, although then the additional data structures needed to collect analysis results would have to compete with the application for the available memory, reducing the space left to store event data. Finally, the collation, which is still serialized through a single writer, could be parallelized by having all analysis processes write their local results directly to a single or at least a small number of physical files. Not using a separate file per process avoids directory contention during file creation, an approach we are currently testing to speed up the generation of trace files.

However, the most significant limitation of our trace-analysis approach is the restriction imposed on the number of events per process: if too large, the trace does not fit into the main memory available to a single analysis process. To become more flexible in this respect, we are considering supporting selective tracing in a more automated way, identifying classes of behaviorally equivalent execution phases based on more space-efficient phase profiles, tracing only representatives of each class, and extrapolating the analysis results back to the full length of execution. A more far-reaching solution would be to consume the trace data intermittently at runtime. In this context, it appears worthwhile to evaluate the extent to which global synchronization operations can be exploited to process trace buffers on-line and how the inevitable perturbation introduced by temporary suspensions of the execution can be compensated for.

On a final note, the diagnostic method presented in this article will still require substantial resources, even if all the optimizations outlined above become effective. Although it is not uncommon in science for the study of a phenomenon to take as long or even longer than the phenomenon itself, as the evaluations of particle collisions in high-energy physics or the simulation of protein folding demonstrate, the yardstick should be whether the costs of obtaining an insight outweigh the benefits. So far, our method has primarily been able to diagnose only symptoms, which makes it hard to translate the results into measurable profits. Simply knowing the locations of wait states in the program is often insufficient to understand the reason for their occurrence. We therefore hope to expand the potential of our approach in such a way that eventually it will be possible to more easily turn the insights into improvements of application performance. Building on earlier ideas by Meira, Jr., et al. [17], we plan to extend the current Scalasca trace analysis such that it can distinguish between primary and secondary waiting times, with secondary waiting times being caused by primary ones. In addition, we are investigating the notion of a delay, which would be the counterpart of a primary waiting time, allowing us to determine why waiting occurred. A major challenge will be to make this distinction at much larger scales than was possible before. Once reasonable candidate optimization hypotheses have been derived from identified delays, they could be efficiently evaluated using trace-based real-time simulation, a scalable approach which we have already successfully demonstrated with a limited set of examples [11].

Acknowledgments. This work was supported by the Helmholtz Association under Grants No. VH-NG-118 and No. VH-VI-228. Also, we would like to thank Marek Behr, Mike Nicolai, and Markus Probst from the Chair for Computational Analysis of Technical Systems at RWTH Aachen University for giving us access to their code.

References

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

[2] Advanced Simulation and Computing Program, The ASC SMG2000 benchmark code, `https://asc.llnl.gov/computing_resources/purple/archive/benchmarks/smg/` (2001).

[3] Argonne National Laboratory, The FPMPI-2 MPI profiling library, `http://www-unix.mcs.anl.gov/fpmpi/`.

[4] D. Becker, R. Rabenseifner, F. Wolf, Timestamp synchronization for event traces of large-scale message-passing applications, in: Proc. of the 14th European PVM/MPI Users' Group Meeting, Paris, France, vol. 4757 of Lecture Notes in Computer Science, Springer, 2007.

[5] M. Behr, D. Arora, O. Coronado, M. Pasquali, Models and finite element techniques for blood flow simulation, International Journal of Computational Fluid Dynamics 20 (2006) 175–181.

[6] H. Brunst, W. E. Nagel, Scalable performance analysis of parallel systems: Concepts and experiences, in: Proc. of the International Conference on Parallel Computing (ParCo), Dresden, Germany, 2003.

[7] F. Freitag, J. Caubet, J. Labarta, On the scalability of tracing mechanisms, in: Proc. of the 8th Euro-Par Conference, Paderborn, Germany, vol. 2400 of Lecture Notes in Computer Science, Springer, 2002.

[8] M. Geimer, F. Wolf, A. Knüpfer, B. Mohr, B. J. N. Wylie, A parallel trace-data interface for scalable performance analysis, in: Proc. of the Workshop on State-of-the-Art in Scientific and Parallel Computing (PARA), Umeå, Sweden, vol. 4699 of Lecture Notes in Computer Science, Springer, 2006.

[9] M. Geimer, F. Wolf, B. J. N. Wylie, B. Mohr, Scalable parallel trace-based performance analysis, in: Proc. 13th European PVM/MPI Users' Group Meeting, Bonn, Germany, vol. 4192 of Lecture Notes in Computer Science, Springer, 2006.

[10] W. Gropp, E. Lusk, A. Skjellum, Using MPI - Portable Parallel Programming with the Message-Passing Interface, chap. 4, 2nd ed., MIT Press, 1999, pp. 81–93.

[11] M.-A. Hermanns, M. Geimer, F. Wolf, B. J. N. 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), IEEE Computer Society, Weimar, Germany, 2009.

[12] IBM Blue Gene/P Team, Overview of the IBM Blue Gene/P project, IBM J. Res. & Dev. 52 (1/2) (2008) 199–220.

[13] Jülich Supercomputing Centre, Scalasca parallel performance analysis toolset documentation, `http://www.scalasca.org/software/documentation`.

[14] A. Knüpfer, W. E. Nagel, Construction and compression of complete call graphs for post-mortem program trace analysis, in: Proc. of the International Conference on Parallel Processing (ICPP), Oslo, Norway, IEEE Computer Society, 2005.

[15] J. Labarta, J. Gimenez, E. Martinez, P. Gonzalez, H. Servat, G. Llort, X. Aguilar, Scalability of visualization and tracing tools, in: Proc. of the International Conference on Parallel Computing (ParCo), Málaga, Spain, 2005.

[16] J. Labarta, S. Girona, V. Pillet, T. Cortes, L. Gregoris, DiP : A parallel program development environment, in: Proc. of the 2nd International Euro-Par Conference, Lyon, France, Springer, 1996.

[17] W. Meira, Jr., T. L. LeBlanc, A. Poulos, Waiting time analysis and performance visualization in Carnival, in: Proc. of the SIGMETRICS Symposium on Parallel and Distributed Tools, Philadelphia, PA, 1996.

[18] B. P. Miller, M. Clark, J. K. Hollingsworth, S. Kierstead, S.-S. Lim, T. Torzewski, IPS-2: The second generation of a parallel program measurement system, IEEE Trans. on Parallel and Distributed Systems 1 (2) (1990) 206–217.

[19] O. Morajko, A. Morajko, T. Margalef, E. Luque, On-line performance modeling for MPI applications, in: Proc. of the 14th Euro-Par Conference, Las Palmas de Gran Canaria, Spain, vol. 5168 of Lecture Notes in Computer Science, Springer, 2008.

[20] MPI PERUSE, `http://www.mpi-peruse.org/`.

[21] W. Nagel, M. Weber, H.-C. Hoppe, K. Solchenbach, VAMPIR: Visualization and analysis of MPI resources, Supercomputer 63, XII (1) (1996) 69–80.

[22] M. Noeth, F. Müller, M. Schulz, B. R. de Supinski, Scalable compression and replay of communication traces in massively parallel environments, in: Proc. of the IEEE International Parallel and Distributed Processing Symposium (IPDPS), Long Beach, CA, 2007.

[23] P. Ratn, F. Müller, B. R. de Supinski, M. Schulz, Preserving time in large-scale communication traces, in: Proc. of the 22nd International Conference on Supercomputing (ICS), Kos, Greece, 2008.

[24] P. C. Roth, D. C. Arnold, B. P. Miller, MRNet: A software-based multicast/reduction network for scalable tools, in: Proc. of the Supercomputing Conference (SC), Phoenix, AZ, 2003.

[25] J. Vetter, Performance analysis of distributed applications using automatic classification of communication inefficencies, in: Proc. of the ACM International Conference on Supercomputing (ICS), Santa Fe, NM, 2000.

[26] F. Wolf, EARL – API documentation, Tech. Rep. ICL-UT-04-03, University of Tennessee, Innovative Computing Laboratory (Oct. 2004).

[27] F. Wolf, F. Freitag, B. Mohr, S. Moore, B. J. N. Wylie, Large event traces in parallel performance analysis, in: Proc. of the 8th Workshop on Parallel Systems and Algorithms (PASA), Frankfurt am Main, Germany, Lecture Notes in Informatics, Gesellschaft für Informatik, 2006.

[28] F. Wolf, B. Mohr, Automatic performance analysis of hybrid MPI/OpenMP applications, Journal of Systems Architecture 49 (10-11) (2003) 421–439.

[29] F. Wolf, B. Mohr, EPILOG Binary Trace-Data Format, Tech. Rep. FZJ-ZAM-IB-2004-06, Forschungszentrum Jülich (May 2004).

[30] F. Wolf, B. Mohr, J. Dongarra, S. Moore, Efficient pattern search in large traces through successive refinement, in: Proc. of the 10th Euro-Par Conference, Pisa, Italy, vol. 3149 of Lecture Notes in Computer Science, Springer, 2004.

[31] F. Wolf, B. J. N. Wylie, E. Ábrahám, D. Becker, W. Frings, K. Fürlinger, M. Geimer, M.-A. Hermanns, B. Mohr, S. Moore, M. Pfeifer, Z. Szebenyi, Usage of the Scalasca toolset for scalable performance analysis of large-scale parallel applications, in: Tools for High Performance Computing: Proc. of the 2nd HLRS Parallel Tools Workshop, Stuttgart, Germany, Springer, 2008.

[32] C. E. Wu, A. Bolmarcich, M. Snir, D. Wootton, F. Parpia, A. Chan, E. Lusk, W. Gropp, From trace generation to visualization: A performance framework for distributed parallel systems, in: Proc. of the Supercomputing Conference (SC), Dallas, TX, 2000.

[33] B. J. N. Wylie, M. Geimer, M. Nicolai, M. Probst, Performance analysis and tuning of the XNS CFD solver on BlueGene/L, in: Proc. 14th European PVM/MPI Users' Group Meeting, Paris, France, vol. 4757 of Lecture Notes in Computer Science, Springer, 2007.

[34] O. Zaki, E. Lusk, W. Gropp, D. Swider, Toward scalable performance visualization with Jumpshot, High Performance Computing Applications 13 (2) (1999) 277–288.