Trace-Based Detection of Lock Contention in MPI One-Sided Communication

Marc-André Hermanns, Markus Geimer, Bernd Mohr, and Felix Wolf

Abstract Performance analysis is an essential part of the development process of HPC applications. Thus, developers need adequate tools to evaluate design and implementation decisions to effectively develop efficient parallel applications. Therefore, it is crucial that tools provide an as complete support as possible for the available language and library features to ensure that design decisions are not negatively influenced by the level of available tool support. The message passing interface (MPI) supports three basic communication paradigms: point-to-point, collective, and one-sided. Each of these targets and excels at a specific application scenario. While current performance tools support the first two quite well, one-sided communication is often neglected. In our earlier work, we were able to reduce this gap by showing how wait states in MPI one-sided communication using active-target synchronization can be detected at large scale using our trace-based message replay technique. Further extending our work on the detection of progress-related wait states in ARMCI, this paper presents an improved infrastructure that is capable of not only detecting progress-related wait states, but also wait states due to lock contention in MPI passive-target synchronization. We present an event-based definition of lock contention, the trace-based algorithm to detect it, as well as initial results with a micro-benchmark and an application kernel scaling up to 65,536 processes.

M.-A. Hermanns (✉) • B. Mohr
JARA-HPC, Jülich Supercomputing Centre, Forschungszentrum Jülich GmbH, Jülich, Germany
e-mail: m.a.hermanns@fz-juelich.de; b.mohr@fz-juelich.de

M. Geimer
Jülich Supercomputing Centre, Forschungszentrum Jülich GmbH, Jülich, Germany
e-mail: m.geimer@fz-juelich.de

F. Wolf
Parallel Programming, TU Darmstadt, Darmstadt, Germany
e-mail: wolf@cs.tu-darmstadt.de

C. Niethammer et al. (eds.), *Tools for High Performance Computing 2016*,
DOI 10.1007/978-3-319-56702-0_6

1 Introduction

The Message Passing Interface (MPI) standard [11] supports three communication paradigms: point-to-point, collective, and one-sided communication. Together, they span the space of possible message-passing scenarios using MPI, each supporting distinct communication patterns. Although the functionality of either paradigm may be implemented using one of the others, the separate interfaces enable internal optimizations for a specific communication scenario. While point-to-point and collective communication are well supported by current performance analysis tools, one-sided communication is in comparison still lacking equal support. We believe that the level of available tool support for a language feature or library has a direct influence on the level of adoption by users. Considering that MPI 3.0 expanded its support for MPI one-sided communication, especially in the area of passive-target synchronization, it is therefore important to close this support gap, and open these new features to new users.

The Scalasca performance analysis toolset [5] provides a trace-based parallel performance analyzer, which automatically identifies wait states in communication and synchronization scenarios. Such wait states are situations in the parallel application execution where one process or thread waits for an activity on another process or thread to begin or end, before it can continue its own activities. A classic example of such a wait state is the *Late Sender* pattern, where a receiving process is waiting in a blocking receive operation for the sender to start the data transfer. To enable an efficient handling of large event traces, Scalasca uses a post-mortem parallel message replay technique, where performance relevant information is passed along the recorded communication paths of the measured application. We have shown in our earlier work how this replay technique can also be used to detect wait states in one-sided communication in the case of MPI active-target synchronization [6]. Passive-target synchronization, however, poses significant challenges to the replay technique. Information on communication and synchronization paths are largely implicit, thus the original replay does not have sufficient information to identify such wait states. For *Wait for Progress* wait states, we have shown—using the example of the one-sided communication interface ARMCI [7]—how the limitations of the original replay can be overcome by extending the communication infrastructure with an active-message-like communication interface, capable of sending asynchronous messages between arbitrary processes.

Progress-related wait states, however, are not the only wait states in passive-target synchronization. MPI provides passive-target synchronization using the concept of locks. As with all synchronization functions, using locks to ensure mutual exclusion during updates to remote memory bears the potential for wait states on processes with conflicting accesses. The detection of such wait states in MPI

passive-target synchronization, however, required a significant redesign of our initial implementation. The contributions of this work include

1. the extension and generalization of the communication infrastructure introduced in our earlier work [7], and
2. the detection of the *Lock Contention* wait state in lock-based synchronization.

The remainder of this paper is organized as follows. Section 2 discusses related work regarding the detection of lock contention in message-passing systems. Sections 3 and 4 first define the *Lock Contention* wait state and then discuss our implementation to detect it in MPI passive-target synchronization. Section 5 shows early results of measurements with two benchmark applications to demonstrate scalability and applicability in two common scenarios. Finally, Sect. 6 summarizes the contributions of this paper and provides an outlook on further optimizations of the detection as well as future integration of detected information in other parts of Scalasca's automatic analysis.

2 Related Work

Available performance analysis tools investigating lock contention, such as Intel VTune [8] or HPCToolkit [1], commonly focus on multi-threading scenarios. Locks in multi-threaded systems are similar in concept to locks in one-sided communication, however, their analysis can draw from different sources of information, such as accessing information already shared on the process-level. In this context, Tallent et al. even investigate root causes of shared-memory lock contention using blame shifting [13].

Tallent et al. also investigated the root causes of network contention in one-sided communication [14]. They focus on the message delivery and compare the actual time with the expected time, estimated through a model based on network and message parameters, and specifically exclude the investigation of synchronization time. Furthermore, they accurately estimate the total delay through network contention, yet, do not identify other processes or threads that are involved in the contention instance.

Zounmevo et al. describe the inefficiency pattern *Late Unlock* [15], which is a sub-pattern to the *Lock Contention* pattern described in this work, where lock contention occurs due to processes holding on to a lock longer than necessary. It is similar in nature to the *Late Complete* wait state defined in our earlier work on one-sided communication wait-state patterns [10]. The existence of such wait states in the use of MPI one-sided communication forms the motivation for the actual focus of their paper, the introduction of non-blocking epochs to prevent this kind of wait state. However, how to detect or quantify lock contention wait states is not discussed. With the infrastructure presented in this paper, the detection of their *Late Unlock* wait state pattern is a straight-forward part of our future work.

In our earlier work [7], we have introduced a scalable framework to identify wait states in passive-target synchronization in the Aggregate Remote Memory Copy Interface (ARMCI) [12]. The presented prototype used ARMCI one-sided communication with a collectively allocated fixed-size buffer per process to exchange data. While being suited for the fixed-size message data needed to identify progress-related wait states, the analysis of lock contention in MPI-based applications cannot guarantee a fixed upper bound to the buffer size, as it needs the full epoch information (including a dynamic number of RMA operations) to identify the point of lock acquisition within the lock epoch.

3 Lock Contention

Remote memory accesses need to use synchronization mechanisms to ensure consistency in the case of concurrent accesses. MPI defines two classes of synchronization schemes based on the explicit involvement of the target process: *active-target* and *passive-target*. Synchronization using the active-target class has both processes, the target and the origin of the one-sided communication operation, perform synchronization calls. In our earlier work [6], we have shown how wait states occurring in this synchronization class can be detected efficiently. Such synchronization can be employed effectively when the target process knows that its memory is being accessed during a specific period of time. In contrast, synchronization using the passive-target class has only the origin process actively involved in the synchronization, leaving the target process passive. Synchronization of this kind uses the concept of shared and exclusive locks to ensure mutual exclusion where necessary when accessing an MPI memory window. In lock-based synchronization schemes, critical code sections need to be guarded by calls to acquire and release a lock at the beginning and end of the code section, respectively. However, only the process performing the memory access (origin) has to call the synchronization explicitly. Target-side synchronization is performed implicitly by the runtime system. With exclusive locks, only a single process can hold a lock at any single moment; with shared locks, multiple origin processes can hold a lock concurrently. As shared locks only block other exclusive locks until their release but allow concurrent shared locks to be acquired, they present less chance of wait states and should be preferred in scenarios where the target memory is not modified.

The acquisition of any type of lock may naturally lead to a wait state, depending on the current state of the lock. To allow implementations to minimize such wait states, the MPI standard does not mandate the call to `MPI_Win_lock` to block until the lock is acquired for a window on a remote process, as long as the implementation also ensures that any accesses to the corresponding window are also postponed until the lock is finally acquired. Only the call to `MPI_Win_unlock` ensures that all pending accesses are completed once the call returns. The acquisition of the lock in MPI passive-target synchronization can therefore occur at any point in time between entering the `MPI_Win_lock` call and leaving the call to

`MPI_Win_unlock`. Local accesses to a given window also have to be guarded by the same synchronization calls. Unlike remote accesses, local stores to the window cannot be postponed by the runtime, as those updates are not performed through MPI functions. Therefore, the call to `MPI_Win_lock` has to block until the lock can actually be acquired by the process.

A lock is an access token and can be seen as a shared resource itself, with multiple processes competing for its ownership. The state when a process experiences wait states or delays due to other processes accessing the same shared resource is called *contention*. Wait states in the lock-based mutual exclusion mechanisms are therefore a special case of the general *resource contention* that can also be experienced with other shared resources, such as file systems or network devices. For their detection, information about all concurrent accesses needs to be gathered and analyzed.

In general, the *Lock Contention* wait state occurs when a process requests a lock of conflicting type to the one that is currently held by another process on the same resource. It then has to wait for the release of the lock by that process. If a process holds an exclusive lock for a given resource, no other process can acquire a lock—shared or exclusive—before the lock is released. If a process holds a shared lock, other shared locks can be obtained by other processes, while an exclusive lock can only be obtained again after all shared locks are released. This means that a specific process (if waiting) is only waiting for a specific process to release a lock, while multiple processes may be waiting for the same process to release it.

Figure 1 shows the different acquisition scenarios possible in MPI passive-target synchronization. The MPI prefixes to the respective calls have been omitted for clarity. The duration of each MPI call is modeled by an enter event (E) and a leave event (L) with corresponding time stamps. Remote memory access (RMA) operations as well as locking and unlocking events are modeled by corresponding event types in the respective function calls, but have been omitted in the figure

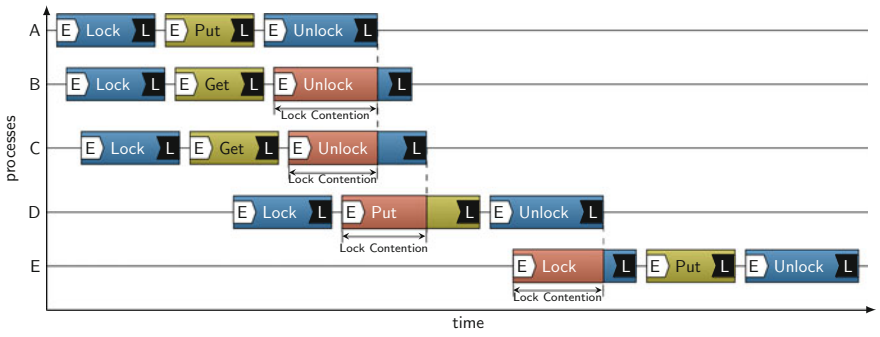

Fig. 1 Potential locations of wait states due to *Lock Contention*. When multiple processes access the same window on the same location, lock access chains build up. In this example, write accesses are protected by exclusive locks, whereas read accesses are protected by shared locks. In MPI passive-target synchronization, the moment of lock acquisition may not be known explicitly, but can only be inferred by checking the time of release of previous lock owners

for clarity as well. The locking behavior of a specific MPI implementation may depend on the available networking hardware or runtime parameters and thus may or may not be the same throughout the execution of the application. As already noted, similar to the relaxed blocking semantics of MPI general active-target synchronization, MPI passive-target synchronization only requires the unlock to guarantee completion of all pending RMA operations to the corresponding memory window, as long as the mutual exclusion requirements of the requested locking types are met. In the figure, we assume that all put operations are guarded by exclusive locks, while the get operations are guarded by shared locks. The target process is not shown, as it is not explicitly involved in the ordering of the concurrent accesses.

Process A requests an exclusive lock to the target window. As no other process is currently holding a lock, it can acquire it without waiting time. For this example, it is of no further interest which of the calls on process A actually acquired the lock. Processes B and C request shared locks to the target window, while process A still holds its exclusive lock. On either process, the lock acquisition and RMA operations are postponed until the unlock function call, where both processes wait for process A to release its lock. Process D requests an exclusive lock, while processes B and C still hold their shared locks. While the lock acquisition is postponed until after the return of the lock function call, the RMA operation call is blocked until the lock can be acquired after the last of the two processes (process C) releases its lock. Finally, process E requests the lock while process D is still holding on to its lock, and directly waits in the lock acquisition call until the lock is released by process D. The different scenarios shown in this figure depict all locations in the passive-target synchronization scenarios where a *Lock Contention* wait state can occur.

Generalizing from its potential locations in passive-target synchronization, the waiting time due to a *Lock Contention* wait state can formally be defined as the dependency between two activities on two distinct origin processes.

Definition 1 (Lock Contention) Let a_p and a_q be the activities of a passive-target synchronization or remote-memory access operation on origin processes p and q. Assume that a_p cannot complete before the acquisition of the corresponding lock held by process q. Assume further that q releases the lock at the end of activity a_q.

Then, the waiting time ω on process p is defined as the overlapping time of the two activities between the start of a_p and the end of a_q:

$$\omega = \begin{cases} \text{Leave}(a_q) - \text{Enter}(a_p) & \text{, if } \text{Enter}(a_p) < \text{Leave}(a_q) \leq \text{Leave}(a_p) \\ 0 & \text{, otherwise} \end{cases}$$

4 Wait-State Detection

The detection and quantification of *Lock Contention* wait states described in this paper is embedded into the message-replay algorithm of the Scalasca trace-based performance analysis toolset [5]. Scalasca assumes that wait states occur at points in the application execution where the execution of a thread or a process needs to communicate or synchronize with another thread or process, respectively. The detection of waiting time on either process needs information from all threads and processes involved. Using the communication and synchronization information encoded in the event trace, created during a measurement run, Scalasca transfers the information from one thread or process to another, for the latter to detect and quantify any waiting time. This approach has been employed successfully in the past for point-to-point and collective [5] as well as MPI one-sided communication using active-target synchronization [6]. For passive-target synchronization, two main challenges exist: (1) communication and synchronization information are only available in the event trace of the origin process and (2) only partial synchronization information is available during measurement. To address these challenges, the original replay infrastructure needed to be extended to allow communication along implicit communication paths.

4.1 The Active-Message Infrastructure

In our earlier work [7], we have introduced a framework that overcomes the original shortcoming of Scalasca's replay method for the case of detecting wait states due to insufficient target-side progress. While the overall concept as an active-message framework is also applicable to the detection of *Lock Contention* wait states, the information needed to detect which operation actually acquired the lock in MPI passive-target synchronization added the requirement of arbitrary-sized messages. This led to a complete re-design of the implementation. The overall requirements on the messaging infrastructure for the detection of lock contention in MPI passive-target synchronization are: the support of (1) inter-process communication not relying on specific target-side event records, (2) communication on paths not explicitly recorded, (3) asynchronous information exchange to enable runtime optimizations during event processing, (4) target-side execution of arbitrary tasks based on the communicated message, and (5) the exchange of messages of arbitrary size.

Our initial ARMCI prototype already fulfilled the first four requirements, however, the efficient exchange of arbitrary-sized messages through one-sided communication on collectively allocated fixed-sized memory windows posed a serious challenge. Furthermore, the initial implementation also used ARMCI constructs to perform the analysis of the ARMCI events in the trace. Although unproblematic for the general use case of Scalasca, where the measurement and analysis are performed

on the same machine, it does add complexity to use cases where the measurement and analysis are performed on different systems. The Scalasca analyzer, however, is a parallel application in its own right, independent of the measured application and is not required to re-use the same communication infrastructure. With the ubiquity of MPI on HPC platforms, a single implementation to serve the analysis of any one-sided communication interface, supporting both use cases, would benefit the user. With this in mind, the re-design of the active-message infrastructure was driven by the requirement for dynamic message sizes.

Two-sided and collective communication are often used as the data exchange layer in cooperative algorithms where the receiver receives a specific message. The receiver decides where the message data is stored and how to process it. The knowledge of how the data needs to be processed emerges from the context containing the explicit reception of the data. However, for unexpected messages on the application layer, the receiver cannot place the messages in the correct context and therefore does not know how to process them in the application. Any target-side processing of the data therefore needs to be part of the message. *Active messages* encode the context with the message or the message envelope, enabling target-side execution of specific code after the one-sided transfer succeeded. For specific message types, a *message handler* can be registered that will process a message ad hoc at the receiver. The sender, knowing for which context it provides data in the message, also sends the appropriate handler selection with the message. This effectively decouples the message from its receiving context, as the receiver can provide the appropriate message context by calling the handler selected by the sender.

To enable this, all processes need to agree on a specific set of message handlers to be used for communication and how they are encoded. The complexity of actions that can be encoded into a message largely depends on the communication interface and framework used. Some interfaces have a rather restricted set of message handlers that focus on the notification of the data arrival and sending an acknowledgment of transfer completion back to the sender. Others allow more complex message handlers, such as remote procedure calls.

Three classes form the cornerstones of Scalasca's active-message framework: (1) A *runtime* class, which defines the interface to message progress; (2) *request* classes, which define how data is transferred between processes; and (3) *handler* classes, which define packaging of data by the sender, and its processing by the receiver.

The runtime class is designed as a singleton object for each analysis process. It is agnostic to the concrete actions that need to be taken to transfer or process messages and delegates all these actions to other classes. Its interface enables users to enqueue requests that are then transferred to and executed on the target process asynchronously. To enable such asynchronous transfer and execution, the runtime class provides a call to advance communication independently of the current execution context. This enables the use of a variety of polling-based progress engines at the target. Scalasca explicitly calls into the runtime as part

of the event replay mechanism at least once per event. Additionally, it provides capabilities to continuously advance the communication while waiting at collective synchronization points.

Request classes define all concrete actions needed to transfer data between processes. For each communication interface used by the active-message framework, a distinct request class needs to be implemented. The current implementation provides an MPI-based request class, yet, support for further communication interfaces can easily be achieved by implementing further request classes. Note that the MPI-based requests can also be used to analyze applications that do not use MPI themselves, such as ARMCI-only and SHMEM-only applications, as the analysis is performed post mortem and the analyzer is a parallel application separate from the user application, potentially executed on a different HPC system. Additional request classes are therefore only necessary in cases where MPI is not available or a different implementation is desirable.

Handler classes define which data is packed at the origin and how it is unpacked and processed at the receiver. An application using the active-message framework, such as Scalasca's parallel analyzer, needs to derive specific handlers for each distinct task on the receiver side. Each handler provides an interface to `pack` all necessary data on the origin and `execute` data processing on the target.

Figure 2 shows an example of an active-message interaction between an origin and a target process. The origin initially creates a request that is passed to specific handler classes adding data to the request buffer. A single request may contain data from more than one handler (indicated by the opt keyword), enabling request aggregation. The origin enqueues the requests for sending and continues execution, while the runtime sends the message to the target as part of the `advance()` call. The target uses the same call to check for incoming requests. Upon incoming requests, the runtime automatically receives and decodes the message, and creates corresponding handler objects on-the-fly. The handler objects are immediately executed in packaging order. After all pending requests are processed, the progress call returns to the user. Using this flexible active-message framework, Scalasca's parallel analysis now supports the detection of two distinct wait-state patterns: (1) the *Wait for Progress* as presented in [7] and (2) the *Lock Contention* as described in more detail in the following section.

4.2 Detecting Lock Contention

To identify lock contention in one-sided communication, the analysis needs to process the lock acquisition and release times of all locks on a given window. The time between requesting or acquiring a lock and its release by a process is called a *lock epoch*. For one-sided communication interfaces with blocking lock semantics, such as ARMCI [12] and SHMEM [4], this is directly modeled by the events of the respective activities. For these interfaces, the only activities of the *lock epoch* that need to be evaluated during the analysis are the respective activities for acquiring

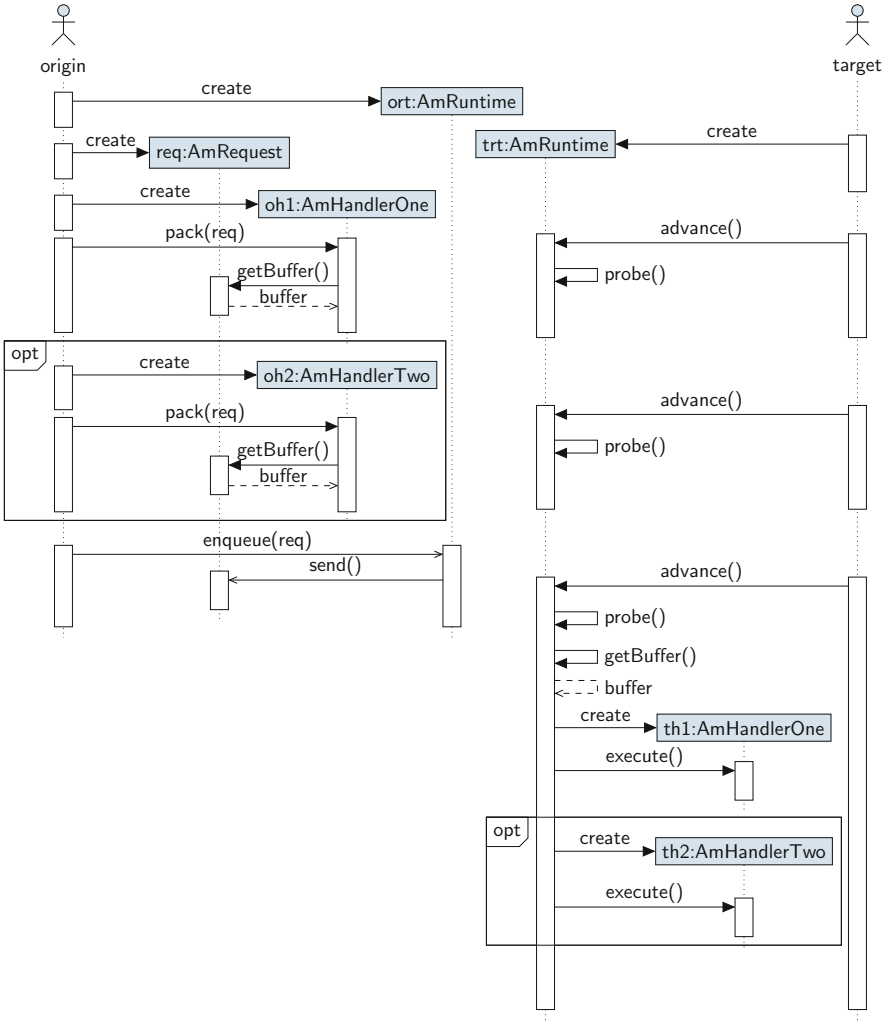

Fig. 2 Interactions between *AmRuntime*, *AmRequest*, and *AmHandler* classes in Scalasca's active-message infrastructure. Calls with *solid arrowheads* are synchronous and block until task completion. Calls with *line arrowheads* are asynchronous and return after initialization; the task will complete as part of the runtime progress. Fragments marked with opt are optional. The origin creates a request and passes it to one or more handlers, packing handler-specific data into its buffer. Once, the origin enqueues the request, the runtime transfers it to the target (not shown). The target has to probe regularly for incoming requests. Upon an incoming request, the handlers are created and executed with information from the request message buffer

and releasing the lock. For one-sided communication interfaces with non-blocking lock semantics, such as MPI, the lock acquisition time has to be computed during the contention analysis, as the event data does not directly encode the time of lock acquisition. For such interfaces, all remote-memory access activities of the lock epoch need to be available to the analysis process to determine the true time of lock acquisition. As a lock epoch can comprise an arbitrary number of RMA operations, the full information needed is of dynamic size.

Lock contention leads to so-called *contention chains*, where multiple processes wait for the successful acquisition of the lock, leading to partial access serialization. Moreover, two or more origin processes may compete for the access to a specific resource, but do not explicitly know of each other. To identify contention, however, the individual local information on the processes have to be compared to each other to (1) identify the order of accesses to the resource and (2) quantify potential waiting time due to a blocked resource. To enable contention analysis for one-sided communication interfaces, all origin processes need to gather the required information at a well-known location. It is important to note that any deterministic location will work, as long as all origin processes locking the same resource choose the same location and allow the contention chain to be determined. For our initial prototype, we chose the target process of a locked window. Further note that the current heuristic to determine the order of accesses does not have enough information to detect and correct skew in the timestamps of locking events. To correct such skew, ordering information would need to be gathered during measurement, where such information is currently not available. Therefore the analysis assumes the accuracy of the timestamps to suffice for ordering.

The analysis follows two phases: (1) gather epoch information; and (2) compute and distribute waiting time information. In the first phase, each origin process caches the relevant lock epoch data until it processes the lock-release event. Then, it creates an active-message request, packed with the lock epoch information, and sends it to the target process. On the target side, the request unpacks the data and stores it for later retrieval. As the active messages coming in from the individual origin processes do not generally arrive in the same order the lock was acquired and released by the application, the target needs to save incoming lock epochs until it reaches a point where it can safely assume to possess the full information on all lock epochs relevant for the contention analysis. Such points are reached at each collective or group-based synchronization point of the window or at collective synchronization points that synchronize at least all processes of the window's communicator. At these points the active-message runtime of Scalasca ensures that all requests are processed before continuing with the analysis. Independent of the locking semantics, all one-sided communication interfaces ensure completion of pending events with the release of the lock. Therefore, the release time of the lock is an indicator for the actual locking order during the application measurement. The target therefore stores

```
Input: Priority queue EpochQueue of lock epochs ordered by descending lock-release time
Output: Waiting time ω_p
if NumElements(EpochQueue) ≥ 2 then
    currentEpoch ← dequeue(EpochQueue);
    while NotEmpty(EpochQueue) do
        previousEpoch ← dequeue(EpochQueue);
        a_q ← GetReleaseActivity(previousEpoch);
        a_p ← FindBlockedActivity(currentEpoch, a_q);
        if Enter(a_p) < a_q ≤ Leave(a_p) then
            ω_p ← Leave(a_q) − Enter(a_p);
            SendContentionInfoTo(p);
        end
        currentEpoch ← previousEpoch;
    end
end
```

Algorithm 1: Compute *lock contention*

the individual lock epochs provided by the origin processes in a data structure sorted by the release time of the lock in the respective epoch.

Once the analysis system can assume that all distributed lock epochs have been collected and inserted into the queue, it can start its contention analysis as described by Algorithm 1. The pseudo-code given assumes a priority-queue data structure that sorts by the unlock timestamp of the corresponding epochs. Furthermore, process p denotes the waiting process, whereas process q denotes the process that p is waiting for. As the epochs are ordered in reverse-chronological lock-release order, the last lock epoch in the contention chain is processed first. The epoch information (currentEpoch) is taken from the queue to initialize the algorithm. Then, while more epoch information is available in the queue, another epoch (previousEpoch) is dequeued to compute the waiting time. For the previous epoch, we identify the activity a_q that released the lock, and the waiting activity a_p within the current epoch. This is done by finding overlap with one of the synchronization or remote-memory access operations within the current epoch with the lock-release activity of the previous epoch. If an overlapping activity is found, the waiting time is computed by the difference between the leave event of a_q and the enter event of a_p, and the respective information is sent to the waiting processes p. Then, the algorithm moves on to the next epoch available in the queue. On process p, the active message handler retrieves the message and adds the waiting time to the respective call path. The algorithm finishes when no further epochs are in the queue, which means the head of the contention chain is reached; the first epoch never suffers from lock contention itself.

5 Results

We tested our implementation of the lock contention detection algorithm using two benchmarks. The first is a verification benchmark that explicitly creates a lock contention to ensure the analysis works correctly. The second is an SOR benchmark, which we have used as a scalability test in earlier work, adapted to use MPI passive target synchronization for the data exchange.

5.1 Micro Benchmark

The lock-contention micro benchmark is used to verify the detection algorithm. It explicitly creates lock contention wait states in a controlled scenario. Processes are partitioned into process 0 acting as the target for all RMA operations, and the rest of the processes, scheduling RMA operations to update the window on the target process. After an initial barrier synchronization of all processes, all processes call the function `foo()` to simulate work with process-individual workloads. The simulated workload is the lowest on target rank 0 and increases with rank, thus the processes return from `foo()` in rank order. As the target has the lowest workload, it is the first to return from the call to `foo()` and is guaranteed to lock its local window before any of the other processes requested the lock. Locks on the local window are never postponed but block until the lock is successfully acquired, as a local lock epoch needs to ensure that local loads and stores to the window are appropriately protected. While the target holds the lock, it executes the function `bar()` for 2 s to simulate local updates to the window before releasing the lock again. The skew in the workload simulated by `foo()` ensures that the workers request the lock after it has been acquired by the target rank 0. They form a contention chain waiting for rank 0 to release the lock. Each process calls `foo()` again for a duration of 100 µs after its release of the lock. Finally, all processes are synchronized by another barrier operation.

The skew of the processes after completing the remote memory access leading to a subsequent *Wait at Barrier* wait state is independent of the initial skew induced by the calls to `foo` on the different processes; it only depends on the time needed to complete the RMA access and to pass the lock ownership to the next process.

The benchmark was executed on two nodes of a Linux Cluster with InfiniBand network using Open-MPI 1.10.0. Figure 3 shows screenshots of Vampir timeline views of selected regions of the measurement, as well as the corresponding Cube report as generated by Scalasca's trace analyzer. In the timeline views, user functions are shown in grey and MPI functions are shown in blue. Figure 3a shows the start of the lock contention, where each process initially calls function `foo()` for a rank-dependent duration. The following call to `MPI_Win_lock()` is too short for Vampir to place the name of the call in the respective timeline. The same applies for the RMA operations following the locks on processes 1 and higher.

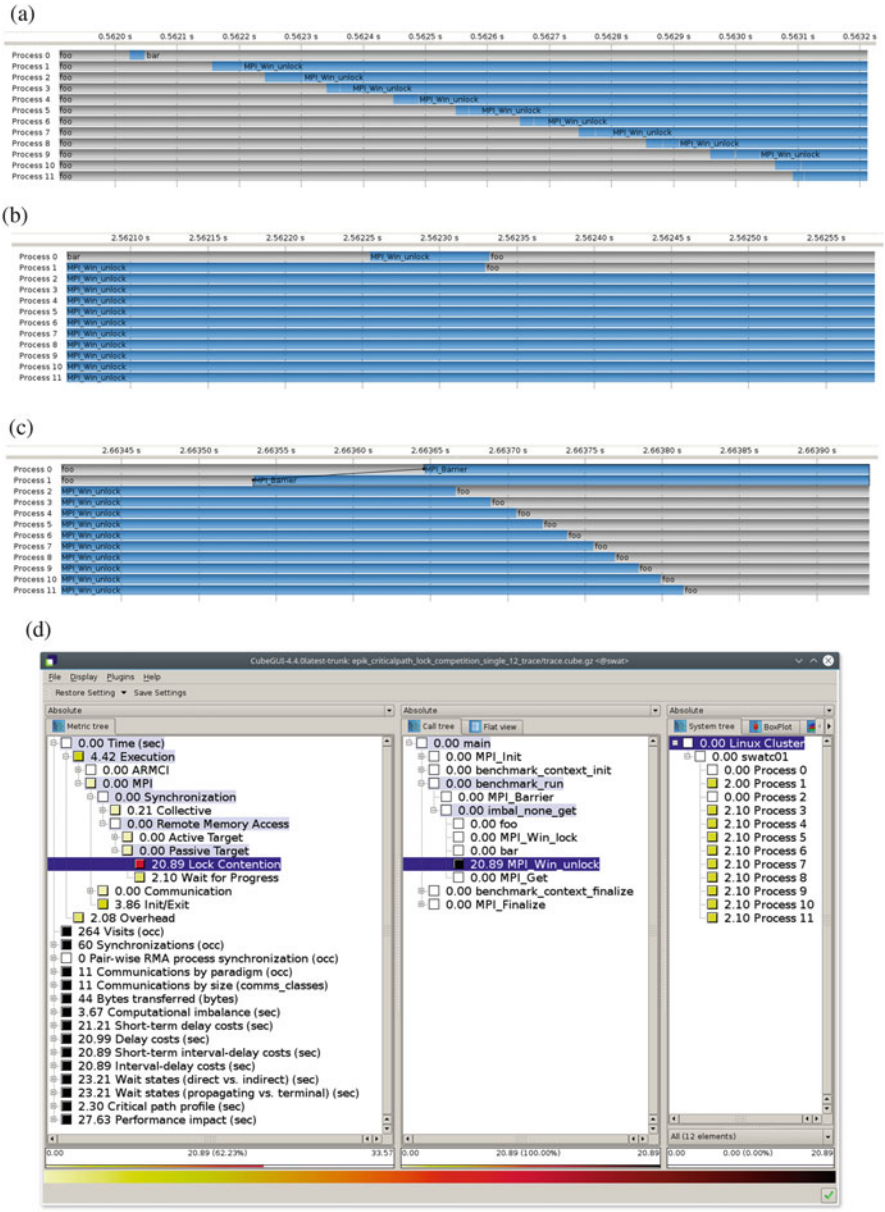

Fig. 3 Timeline views and Cube report of the execution of the lock-contention micro benchmark. (**a**) Process 0 acquires the lock and executes `bar()`, while remaining processes request the lock. (**b**) Process 0 releases the lock and process 1 completes access; process 2 cannot obtain the lock due to insufficient target-side progress. (**c**) Process 0 provides progress in barrier, enabling remaining processes to complete access. (**d**) Cube analysis report shows waiting time classified as *Lock contention* on all processes but process 2; waiting time on the latter is classified as *Wait for Progress*

Process 0, as the target, obtains an exclusive lock and executes the function `bar()` for 2 s. The remaining processes each block in the call to `MPI_Win_unlock()`, waiting for the target to release the lock. Figure 3b shows a detailed view of the time interval in which the target releases its lock on the window and passes the lock to process 1. Process 1 obtains the lock and performs its RMA operation, releasing the lock again. Process 2, however, is unable to obtain the lock directly from process 1, as the target (process 0) is busy with the execution of `foo()` after its release of the lock. Process 2 can obtain the lock only after process 0 provides progress within the barrier operation (Fig. 3c). As the barrier spans all processes, process 0 has to wait for the last process to join and continues to provide progress for all remaining processes. The call to `foo()` before the barrier is rank independent and lasts for 100 µs.

The Cube performance report shown in Fig. 3d reflects the observed behavior. The time spent in the *Lock Contention* wait state is about 2 s for process 1, which requested the lock right after process 0 and had to wait for the end of the 2 s execution of `bar()`. The waiting time on process 2 is not classified as *Lock Contention* but as *Wait for Progress* (not directly shown), as insufficient progress was the last factor extending the overall waiting time. However, for the remaining processes, progress was provided and the waiting time is classified as contention-based. The waiting time on processes 2 and higher is increased by about 100 µs compared to process 1 as further progress was only provided again after the execution of `foo()` on the target process.

5.2 SOR

The SOR benchmark is a computational kernel that iteratively solves the Poisson equation using a red-black successive over-relaxation method, distributing work on a two-dimensional Cartesian grid. It performs a nearest-neighbor halo exchange in each iteration. Originally implemented using point-to-point communication, we adapted the halo exchange to use one-sided communication in different synchronization schemes. After each iteration, a collective reduction is performed to test for convergence. Problem size and number of processes can easily be configured for a specific run. For the presented scaling measurements, the benchmark was configured for weak scaling, keeping the load per process constant. To prevent convergence, it was configured to perform a maximum of 500 iterations with a small error tolerance of 1×10^{-7}, to ensure the same number of iterations for each run. For the different execution scales, the processes were doubled in alternating dimensions, starting with a 32×16 process grid.

The Scalasca analyzer processes the event trace in different stages. The initial stage identifies the majority of the wait states, while further stages concentrate on the computation of higher-level metrics such as root causes and the critical path.

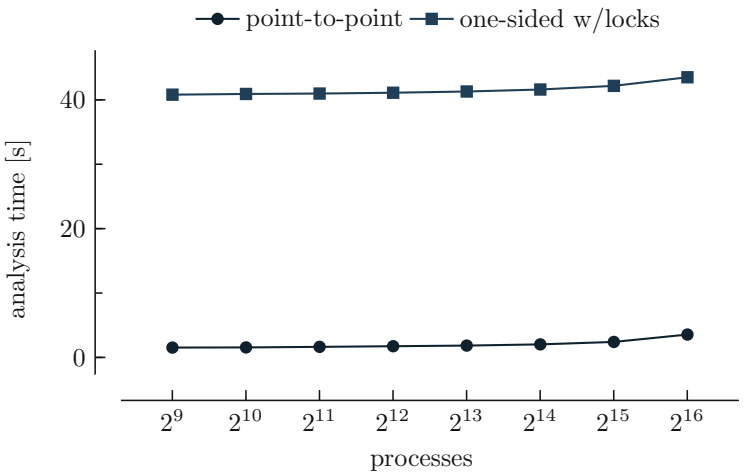

Fig. 4 Scaling results for the analysis of the SOR benchmark configured to run with point-to-point and one-sided communication using lock synchronization, respectively

As these extended analyses are outside the scope of this paper, Fig. 4 only shows the execution time of the initial stage of the analysis. Measurements were taken on the IBM Blue Gene/Q system JUQUEEN at the Jülich Supercomputing Centre of Forschungszentrum Jülich [9]. The two data series are named after the SOR implementation of the halo exchange measured. The analysis times shown for both SOR implementations also include the detection and quantification of collective communication wait states. While the analysis time for each scale is significantly higher for the analysis of one-sided communication compared to the point-to-point case, the study still demonstrates a similar scaling behavior in general. This indicates scale-independent overheads in the replay mechanism. Initial performance measurements indicate up to 10% of the runtime overhead due to the additional execution of the progress engine. Most of the overhead is therefore part of the message transfer itself (i.e., the active-message requests) and the execution of the handlers. Improved buffer reuse for the active-message requests may lower memory allocation overheads for the data transfer. For the handler execution, most handlers need to search the target-side trace for the corresponding event, incurring an $O(\log n)$ additional execution overhead per handler execution where n is number of events in the target-side trace, which may prove difficult to reduce. We plan to further investigate optimization targets to reduce the overall runtime overhead during the integration of the analysis prototype into the production version of the Scalasca analyzer, however, the out-of-order nature of the data handling during the analysis of passive-target synchronization constructs will likely remain more costly than the in-order processing of point-to-point and active-target synchronization constructs.

6 Conclusion and Outlook

In this paper we showcased our extended and generalized infrastructure for detecting and quantifying waiting time in passive-target one-sided communication constructs, at the example of lock contention. Using this infrastructure, we were able to re-construct process synchronization schemes not directly evident from the measurement data, and to demonstrate that waiting time is correctly detected and classified. The current analysis heuristic evaluates contention and progress-related wait states and classifies waiting time accordingly. While the implementation still provides room for optimization, the software prototype showed good scaling behavior up to 65,536 processes for the analysis of a common computational kernel using a halo exchange on a two-dimensional Cartesian grid.

The presented analysis prototype is handling MPI-2 one-sided communication. As part of our future work, we plan to extend the support to the additional synchronization calls of MPI-3 and beyond. Further optimization of the messaging infrastructure will be a high priority for the integration into the production version of the Scalasca analyzer. To provide a better load balancing during the analysis, we also plan to explore different epoch distribution schemes beyond the current target-centric approach, such as timeslice-based round robin distribution.

For the identification of the critical path [3] and root causes of wait states [2] it is critical to identify all wait states in the application. With contention-based wait states for one-sided communication being detected by the analyzer, we further plan to integrate their handling into our current critical-path and root-cause analysis. Furthermore, such an integration can then be used to also cover thread-based locking mechanisms as provided by POSIX threads or OpenMP.

Acknowledgements This work has been partly funded by the Excellence Initiative of the German federal and state governments. The authors gratefully acknowledge the computing time granted by the JARA-HPC Vergabegremium and VSR commission provided on the JARA-HPC Partition part of the supercomputer JUQUEEN [9] at Forschungszentrum Jülich.

References

1. Adhianto, L., Banerjee, S., Fagan, M.W., Krentel, M., Marin, G., Mellor-Crummey, J.M., Tallent, N.R.: HPCTOOLKIT: tools for performance analysis of optimized parallel programs. Concurr. Comput.: Pract. Exper. **22**(6), 685–701 (2010). doi:10.1002/cpe.1553. http://doi.wiley.com/10.1002/cpe.1553
2. Böhme, D., Geimer, M., Wolf, F., Arnold, L.: Identifying the root causes of wait states in large-scale parallel applications. In: Proceedings of the 39th International Conference on Parallel Processing (ICPP), San Diego, CA, pp. 90–100 (2010). doi:10.1109/ICPP.2010.18
3. Böhme, D., de Supinski, B.R., Geimer, M., Schulz, M., Wolf, F.: Scalable critical-path based performance analysis. In: Proceedings of the 26th IEEE International Parallel & Distributed Processing Symposium (IPDPS), Shanghai (2012)

4. Chapman, B.M., Curtis, A., Pophale, S., Poole, S.W., Kuehn, J.A., Koelbel, C., Smith, L., Curtis, T., Pophale, S., Poole, S.W., Kuehn, J.A., Koelbel, C., Smith, L., Curtis, A., Pophale, S., Poole, S.W., Kuehn, J.A., Koelbel, C., Smith, L.: Introducing OpenSHMEM: SHMEM for the PGAS community. In: Proceedings of the Fourth Conference on Partitioned Global Address Space Programming Model, no. c in PGAS '10, pp. 2:1–2:3. ACM, New York, NY (2010). doi:10.1145/2020373.2020375. http://doi.acm.org/10.1145/2020373.2020375
5. Geimer, M., Wolf, F., Wylie, B.J.N., Mohr, B.: A scalable tool architecture for diagnosing wait states in massively parallel applications. Parallel Comput. **35**(7), 375–388 (2009). doi:10.1016/j.parco.2009.02.003
6. Hermanns, M.A., Geimer, M., Mohr, B., Wolf, F.: Scalable detection of MPI-2 remote memory access inefficiency patterns. Int. J. High Perform. Comput. Appl. **26**(3), 227–236 (2012). doi:10.1177/1094342011406758
7. Hermanns, M.A., Krishnamoorthy, S., Wolf, F.: A scalable infrastructure for the performance analysis of passive target synchronization. Parallel Comput. **39**(3), 132–145 (2013). doi:10.1016/j.parco.2012.09.002. http://www.sciencedirect.com/science/article/pii/S0167819112000762
8. Intel Corp.: Intel VTune Amplifier XE (2012). http://software.intel.com/en-us/intel-vtune-amplifier-xe
9. Jülich Supercomputing Centre: JUQUEEN: IBM Blue Gene/Q Supercomputer System at the Jülich Supercomputing Centre. J. Large-Scale Res. Facil. **1**(A1) (2015). doi:10.17815/jlsrf-1-18. http://dx.doi.org/10.17815/jlsrf-1-18
10. Kühnal, A., Hermanns, M.A., Mohr, B., Wolf, F.: Specification of inefficiency patterns for MPI-2 one-sided communication. In: Proceedings of the 12th Euro-Par Conference, Dresden. Lecture Notes in Computer Science, vol. 4128, pp. 47–62. Springer, Berlin (2006)
11. MPI Forum (ed.): MPI: A Message-Passing Interface Standard. Version 3.1. MPI Forum (2015). http://www.mpi-forum.org/
12. Nieplocha, J., Carpenter, B.: ARMCI: a portable remote memory copy library for distributed array libraries and compiler run-time systems. In: Proceedings of the 11 IPPS/SPDP'99 Workshops Held in Conjunction with the 13th International Parallel Processing Symposium and 10th Symposium on Parallel and Distributed Processing, vol. 1586, pp. 533–546. Springer, London (1999). doi:10.1007/BFb0097937. http://dl.acm.org/citation.cfm?id=645611.662053
13. Tallent, N.R., Mellor-Crummey, J.M., Porterfield, A.: Analyzing lock contention in multithreaded applications. SIGPLAN Not. **45**(5), 269–280 (2010). doi:10.1145/1837853.1693489. http://doi.acm.org/10.1145/1837853.1693489
14. Tallent, N.R., Vishnu, A., Van Dam, H., Daily, J., Kerbyson, D.J., Hoisie, A.: Diagnosing the causes and severity of one-sided message contention. In: Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP 2015, pp. 130–139. ACM, New York, NY (2015). doi:10.1145/2688500.2688516. http://doi.acm.org/10.1145/2688500.2688516
15. Zounmevo, J.A., Zhao, X., Balaji, P., Gropp, W., Afsahi, A.: Nonblocking epochs in MPI one-sided communication. In: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC '14, pp. 475–486. IEEE Press, Piscataway, NJ (2014). doi:10.1109/SC.2014.44. http://dx.doi.org/10.1109/SC.2014.44