Specification of Inefficiency Patterns for MPI-2 One-Sided Communication

Andrej Kühnal, Marc-André Hermanns, Bernd Mohr, and Felix Wolf

Forschungszentrum Jülich,
Zentralinstitut für Angewandte Mathematik,
52425 Jülich, Germany
{a.kuehnal, m.a.hermanns, b.mohr, f.wolf}@fz-juelich.de

Abstract. Automatic performance analysis of parallel programs can be accomplished by scanning event traces of program execution for patterns representing inefficient behavior. The temporal and spatial relationships between individual runtime events recorded in the event trace allow the recognition of wait states as a result of suboptimal parallel interaction. In our earlier work [1], we have shown how patterns related to MPI point-to-point and collective communication can be easily specified using common abstractions that represent execution-state information and links between related events. In this article, we present new abstractions targeting remote memory access (also referred to as one-sided communication) as defined in the MPI-2 standard. We also describe how the general structure of these abstractions differs from our earlier work to accommodate the more complicated sequence of data-transfer and synchronization operations required for this type of communication. To demonstrate the benefits of our methodology, we specify typical performance properties related to one-sided communication.

1 Introduction

Remote memory access (RMA) describes the ability of a process to access a part of the memory of a remote process directly without explicit participation of the remote process in the data transfer. Since all parameters for the data transfer are determined by one process, it is also called *one-sided* or *single-sided* communication. One-sided communication is often made available to the programmer in the form of platform or vendor-specific libraries, such as SHMEM (Cray/SGI) or LAPI (IBM). In 1997, one-sided communication was added to the portable MPI standard version 2 [2].

On platforms with special hardware providing RMA support, one-sided communication can be used to improve the efficiency of parallel applications. For example, NASA researchers reported a 39% improvement in throughput after replacing MPI-1 non-blocking with MPI-2 one-sided communication in a global atmosphere simulation program [3]. As more and more scientists adopt this new paradigm to better utilize the underlying hardware, the demand for performance tools supporting RMA communication will increase. This is especially important in view of the complicated sequences of data transfer and synchronization operations involved and the fact that the MPI specification leaves a large degree of freedom to implementors regarding the blocking behavior of corresponding operations.

W.E. Nagel et al. (Eds.): Euro-Par 2006, LNCS 4128, pp. 47–62, 2006.

A performance-analysis technique successfully applied to traditional message-passing applications is event tracing. An event trace records performance-relevant runtime events, such as routine entries or exits or as sending and receiving point-to-point messages. The KOJAK tool [4] uses the temporal and spatial relationships between individual runtime events reflected in the event trace to recognize patterns that occur as a result of suboptimal parallel interaction. These patterns are specified as compound events to (i) allow the classification of inefficient behavior by describing the exact circumstances causing it and to (ii) enable the quantification of wait times incurred.

Compound events consist of multiple primitive events, as recorded in the trace file, and are connected by relationships, such as message transfers, that are often specific to a particular parallel programming model, such as MPI. They may be further characterized by constraints imposing, for example, a certain temporal order of events. To keep the pattern specifications as simple as possible and also to make the simultaneous search for different patterns more efficient, KOJAK includes a separate layer with common abstractions representing execution-state information and links between related events in terms of which the actual patterns are described.

In our earlier work [1], we have defined abstractions along with typical patterns describing performance properties in the context of traditional message passing (MPI-1) and shared-memory (OpenMP) programming. In [5], we provided informal descriptions of patterns suitable to diagnose inefficiencies related to one-sided communication. In this article, we outline the formal specification of these new abstractions and patterns. Compared to the previous ones related to MPI-1 and OpenMP, the abstractions presented here are more complicated to accommodate the complex sequences of data-transfer and synchronization operations involved in MPI-2 one-sided communication and to reflect the intricate inter-process relationships established by groups denoting potential origins or targets during communication epochs. The new patterns have been incorporated into the KOJAK tool, taking advantage of the recently added measurement infrastructure for one-sided communication events reported in [6]. As part of this effort, we have also specified abstractions and patterns related to SHMEM, which, however, are beyond the scope of this paper.

The outline of this article is as follows: We start with a short description of MPI-2 RMA communication and synchronization functions in Section 2. In Section 3, we give a brief overview of related work. In Section 4, we introduce the idea of creating suitable abstractions on top of which inefficiency patterns can be specified and explain their general structure. After that, we define abstractions for MPI-2 one-sided communication in Section 5. To demonstrate the usefulness of our methodology, Section 6 specifies several example MPI-2 patterns containing wait states the application developer may wish to identify. In Section 7, we conclude our paper and give an outlook on future work.

2 MPI-2 One-Sided Communication

The interface for RMA operations defined by MPI-2 differs from the vendor-specific APIs in many respects. This is to ensure that it can be efficiently implemented on a wide variety of computing platforms even if a platform does not provide any direct hardware

support for RMA. The design behind the MPI-2 RMA API specification is similar to that of weakly coherent memory systems: correct ordering of memory accesses has to be specified by the user with explicit synchronization calls; for efficiency, the implementation can delay communication operations until the synchronization calls occur.

MPI does not allow RMA operations to access arbitrary memory locations. They can access only designated parts of a process' memory, which are called *windows*. Windows must be explicitly initialized (with a call to `MPI_Win_create`) and released (with `MPI_Win_free`) by all processes that either provide memory or want to access this memory. These calls are *collective* between all participating partners and include an internal barrier operation. By *origin* MPI denotes the process that performs an RMA read or write operation, and by *target* the process in which the memory is accessed.

There are three RMA communication calls in MPI: `MPI_Put` transfers data from the caller's memory to the target memory (*remote write*); `MPI_Get` transfers data from the target to the origin (*remote read*); and `MPI_Accumulate`[1] updates locations in the target memory, for example, by replacing them with sums or products of the local and remote data values (*remote update*). These operations are *nonblocking*: the call initiates the transfer, but the transfer may continue after the call returns. The transfer is completed, both at the origin and the target, only when a subsequent synchronization call is issued by the caller on the involved window object. Only then are the transferred values (and the associated communication buffers) available to the program. RMA communication falls in two categories: *active target* and *passive target* communication. In both modes, the parameters of the data transfer are specified only at the origin, however in active mode, both origin and target processes have to participate in the synchronization of the RMA accesses. Only in passive mode is the communication and synchronization completely one-sided.

RMA accesses to locations inside a specific window must occur only within an *access epoch* for this window. Such an access epoch starts with an RMA synchronization call, proceeds with any number of remote read, write, or update operations on this window, and finally completes with another (matching) synchronization call. Additionally, in active target communication, a target window can only be accessed within an *exposure epoch*. RMA operations issued by an origin process for a target window will access that target window during the same exposure epoch if and only if they were issued during the same access epoch. Distinct epochs for a window at the same process must be disjoint. However, epochs pertaining to different windows may overlap.

MPI provides three RMA synchronization mechanisms:

Fences: The `MPI_Win_fence` collective synchronization call is used for active target communication. An access epoch at an origin process or an exposure epoch at a target process are started and completed by such a call. All processes who participated in the creation of the window synchronize, which in most cases includes a barrier. The data transfered is only accessible to user code after the fence.

General Active Target Synchronization (GATS): Here, synchronization is reduced: only pairs of communicating processes synchronize, and they do so only when needed to correctly order accesses to a window with respect to local accesses to

[1] In our model, we consider an accumulate operation as a special version of a put operation and, therefore, distinguish only between get and put in the remainder.

that window. An access epoch is started at an origin process by `MPI_Win_start` and is terminated by a call to `MPI_Win_complete`. The start call specifies the group of targets for that epoch. An exposure epoch is started at a target process by `MPI_Win_post` and is completed by `MPI_Win_wait` or `MPI_Win_test`. Again, the post call specifies the group of origin processes for that epoch. Data written is only accessible after the wait (or test) call, however data can only be read after the complete call.

Locks: Finally, shared and exclusive locks are provided through the `MPI_Lock` and `MPI_Unlock` calls. They are used for passive target communication. In addition, they also define the access epoch for this window at the origin. Data read or written is only accessible from user code after the unlock operation has completed.

It is implementation-defined whether some of the described calls are blocking or non-blocking; for example, in contrast to other shared memory programming paradigms, the lock call must not be blocking. For a complete description of MPI-2 RMA communication see [2].

3 Related Work

Currently, there are only very few tools that support the measurement and analysis of one-sided communication and synchronization on a wide range of platforms. The well-known Paradyn tool which performs an automatic on-line bottleneck search, was recently extended to support several major features of MPI-2 [7]. For RMA analysis, it collects basic, process-local statistical data (i.e., transfer counts and execution time spent in RMA functions). It does not take inter-process relationships into account nor does it provide detailed trace data. Also, it does not support the analysis of SHMEM programs. The very portable TAU performance analysis tool environment [8] supports profiling and tracing of MPI-2 and SHMEM one-sided communication. However, it only monitors the entry and exit of the RMA functions; it does not provide RMA transfer statistics nor are the transfers recorded in tracing mode. The commercial Intel Trace Collector tool (formerly known as VampirTrace) [9] records MPI execution traces. When used with MPI-2, it does not measure the routines of the general active target synchronization, creating the wrong impression that useful user calculations are done instead. Also, message lines show the RMA transfer as completed by the end of the put or get operation, which does not reflect the user-visible behavior, as specified by the MPI-2 standard. Finally, it does not record the collective nature of MPI-2 window functions. Besides these there are also some non-portable vendor tools with similar limitations.

4 State Sequences and Pointer Attributes

Event tracing models the execution of a program as a sequence of events representing actions relevant to the purpose of the observation. Therefore, the selection of event types to be observed defines the view of program execution an event trace can provide. An *event model* defines the formal properties of that view. It comprises a set of event types with an associated set of attributes and constraints defining correct event ordering. Each event has a location attribute as well as a wall-clock time stamp. The event

location is an abstraction usually referring to the process or the thread generating an event. Since the following discussion only considers pure MPI applications, the location of an event can be regarded as equivalent to the MPI process, as identified by the rank in `MPI_COMM_WORLD`. Another attribute denotes the event type. An *event trace* is a finite indexed set of events $E := \{e_1, \ldots, e_{n_e}\}$. The indexing reflects the time-sequenced order of event records in the trace file.

To be able to express complex relationships among the constituents of a compound event, the event model of system observation can be extended by creating instances of two different categories of abstractions: (i) state sequences and (ii) pointer attributes. The process of creating these abstractions is called *event model enhancement* because it enhances the model's capabilities to describe complex situations of execution behavior. We summarize the key concepts below. The interested reader may refer to [10] for more details.

State sequences. An event happening in a parallel system indicates a change in its state, thus, events can be regarded as state transitions. An event trace can be seen as a sequence of state transitions starting at an initial state and changing into the next state, event by event, until a final state is reached after the last event. The state entered as the result of an event is a useful abstraction when specifying compound events that represent inefficient behavior.

The overall state of a parallel system is characterized by different aspects. For example, one aspect might be the set of messages being transferred at a given moment, another aspect might be the dynamic call stack of a process or thread. Such a state aspect can be conveniently characterized in terms of the events that caused that aspect's state. For each of these aspects we can define a *state sequence* that describes the evolution of that aspect over time. A state sequence is inductively defined by a transition operator. The transition operator is applied to the current state and the next event to compute the next state in the sequence. Since a state sequence describes only one aspect of the system, we can combine all state sequences into a vector of state sequences to obtain the *overall-state sequence*.

In our earlier work, a state sequence has been defined as a sequence of event sets. Starting with the empty set, the transition operator either added the current event or removed events related to the current event, changing the event set describing an aspect of the overall system. To conveniently retrieve event sets of interest during trace analysis, we have defined auxiliary functions that can be applied to individual states of a sequence. For example, a scheme that proved to be useful to identify individual collective-operation instances was to collect all events belonging to an instance and retrieving it using an auxiliary function upon its completion if needed. Immediately after this point, the transition operator removes the instance from the set. Later, we will see that simple event sets are inconvenient to describe patterns involving intertwined steps of communication and synchronization, such as occur in one-sided communication, and that a hierarchical grouping of events becomes necessary.

Pointer attributes. Another useful abstraction is a link connecting related events, so that one can navigate from one event to another related event. An example is a link from the event of receiving a message back to the corresponding event of sending it.

This mechanism permits navigation along a path of related events and the definition of relationships among the constituents of a compound event using such paths. A natural way of representing such links is to provide event attributes with pointer semantics, which we call *pointer attributes*.

5 One-Sided Abstractions

In this section, we describe abstractions suitable as building blocks for the specification of inefficiency patterns related to MPI-2 one-sided communication. For reasons of brevity, we refrain from presenting the unabridged formalism underlying our abstractions and try to restrict ourselves to key concepts explained in natural language as far as possible. See [11] for a complete specification.

The state sequences and pointer attributes presented in this article apply to the underlying KOJAK event model, whose relevant portions are summarized in Table 1.

Table 1. Event types in KOJAK relevant to MPI-2 RMA analysis

Abstraction	Event type	Type specific Attributes
Entering / leaving a region	ENTER	region id
(e.g., a function)	EXIT	region id
Leaving an MPI collective function	MPICEXIT	region id, comm id, root loc, sent, recvd
Start / end / origin of RMA	PUT_1TS	window id, rma id, length, dest loc
one-sided transfers	PUT_1TE	window id, rma id, length, src loc
	GET_1TO	window id, rma id
	GET_1TS	window id, rma id, length, dest loc
	GET_1TE	window id, rma id, length, src loc
Leaving an MPI GATS function	MPIWEXIT	window id, region id, group id
Leaving an MPI collective RMA function	MPIWCEXIT	window id, region id, comm id
Locking / unlocking an MPI window	WLOCK	window id, lock loc, type
	WUNLOCK	window id, lock loc

The table lists type-specific attributes that are added to the location attribute and the timestamp mentioned in Section 4. For entries and exits of regions and ,in particular, MPI functions, we record which region was entered or left. In the case of collective MPI functions, instead of "normal" EXIT events, special collective events are used to capture the attributes of the collective operation. This is the communicator, the root process, and the amounts of data sent and received during this operation. Start and end of RMA one-sided transfers are marked with PUT_1TS and PUT_1TE (for remote writes and updates) or with GET_1TS and GET_1TE (for remote reads). For these events, we collect the source and destination and the amount of data transferred, as well as a unique RMA operation identifier which allows an easier mapping of #_1TE to the corresponding #_1TS events in the analysis stage later on. For all MPI RMA communication and synchronization operations we also collect an identification for the window on which the operation was performed. Exits of MPI-2 functions related to general active target

synchronization (GATS) are marked with a MPIWEXIT event which also captures the groups of origin or target processes. For collective MPI-2 RMA functions, we use an MPIWCEXIT event and record the communicator that defines the group of processes participating in the collective operation. Finally, MPI window lock and unlock operations are represented by WLOCK and WUNLOCK events. A more detailed description of the MPI-2-specific events and their implementation in KOJAK can be found in [6].

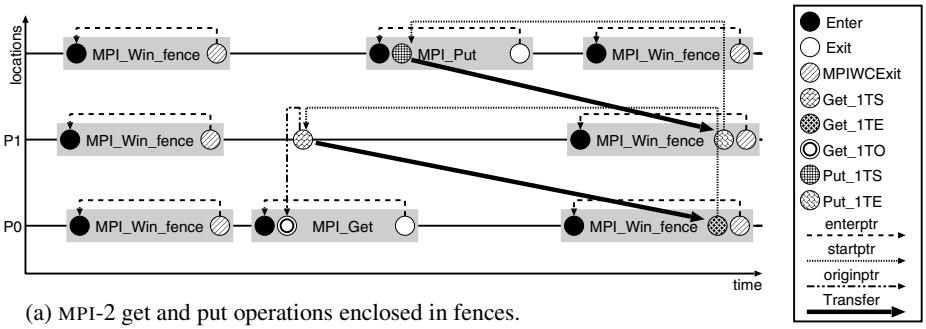
Collective operations. In active target mode, access and exposure epochs may be enclosed in collective fence synchronization operations. The synchronizing character of these operations may result in wait times when processes reach the fence at different points in time. The same applies to functions to create and destroy windows. To detect wait states resulting from collective synchronization, we have defined a state sequence modeling the progress of collective operations on RMA windows - similar to the one for MPI-1 collective communication defined in our previous model.

Since the structure of the RMA-collective sequence is nearly identical to the sequence used in our previous model, we have introduced the concept of generic meta-sequences that can be instantiated with a type argument to simplify the formulation of sequences describing arbitrary collective operations. We have created a meta-sequence $\mathfrak{C}^g < T >$ collecting the exit events of collective operations carried out by members of a group g of processes. Depending on the type T of these exit events, this group is identified either by an MPI communicator, an OpenMP team, or an RMA window. Once all events of type T belonging to a collective operation instance are present, the complete instance is removed upon the next event applied to the set. An auxiliary function $complete < T > (e)$ is provided to query for instances completed by an event e, which is useful to measure waiting times. The state sequence for collective window operations is created by instantiating $\mathfrak{C}^g < MPIWCExit >$. Note that this abstraction can also be used for SHMEM collective operations.

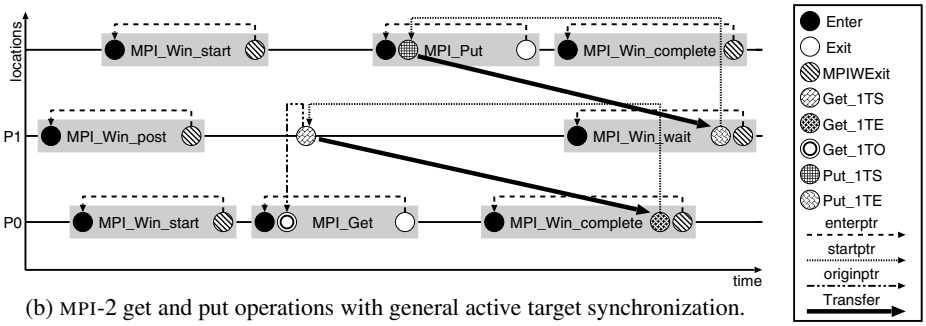
Data transfers. Data transfers are modeled as pairs of events: (i) a start event initiating the transfer (i.e., PUT_1TS or GET_1TS) and (ii) an end event completing the transfer (i.e., PUT_1TE or GET_1TE). KOJAK's event model observes the MPI-2 synchronization semantics and, therefore, reflects the user-visible behavior of MPI-2 RMA operations. Figure 1 shows the model for the three different synchronization methods defined by MPI-2. The transfer line shown in the picture is not part of the model and is only shown for clarity.

The end GATS calls is modeled with MPIWEXIT events, the end of fences with MPIWCEXIT events to capture their collective nature. The transfer-start event is placed at the source process immediately after the begin of the corresponding communication function. However, the transfer-end event is placed at the destination process shortly before the exit of the RMA synchronization function which completes the transfer according to the MPI-2 standard rules. Unfortunately, this has an undesired side effect. As one can see in the figure, this results in a separation of the data transfer for remote reads from the corresponding `MPI_Get` function. To rectify this situation, we have introduced a new event GET_1TO indicating time and location of the transfer's origin.

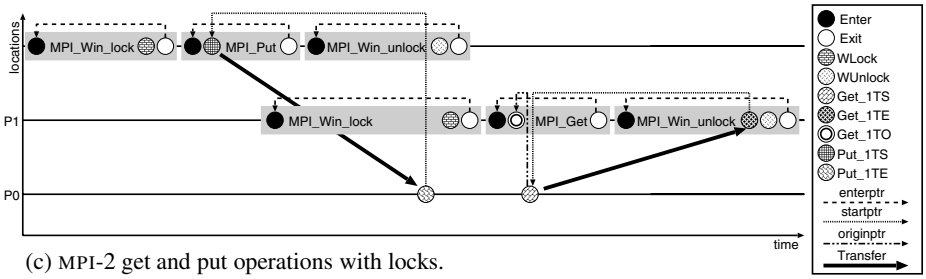
To access all events belonging to the same data transfer, we have defined pointer attributes $startptr$ and $originptr$, which connect the end event with its corresponding



(a) MPI-2 get and put operations enclosed in fences.



(b) MPI-2 get and put operations with general active target synchronization.



(c) MPI-2 get and put operations with locks.

Fig. 1. Examples of KOJAK's MPI-2 event model

start event and the start event with its corresponding origin event, respectively. Their definition is based on state sequences collecting transfer events separately for each location (i.e., process) - similar to the queue for point-to-point messages defined in our earlier model. The identification of events belonging to the same transfer is based on the *rma id* attribute assigned during trace generation. Subsequently, we use these pointer attributes to reach start and origin events for given transfer-end events. Beyond that, these pointer attributes can be useful to calculate matrices with amounts of data transferred between processes.

Access and exposure epochs in general active target synchronization. The most challenging part of analyzing MPI-2 one-sided communication is GATS synchronization. To facilitate cross-process analysis in GATS mode, it is necessary to identify corresponding access and exposure epochs. Here, we present a multi-step method to recognize all

access epochs belonging to a given exposure epoch (and vice versa) with the goal of providing all data needed for their analysis. This is the most intricate part of our model as it requires considering sets of sets of events to reflect the hierarchical grouping of the events involved. Note that this constitutes an important difference to the abstractions defined in our earlier work. We start with an introduction of the overall structure of event sets related to GATS communication:

Data transfer. A put or get operation.
Put operation. A PUT_1TE and its corresponding PUT_1TS event connected by the $startptr$ attribute.
Get operation. A GET_1TE and its corresponding GET_1TS and GET_1TO events connected by $startptr$ and $originptr$ attributes.
Epoch. An access or exposure epoch.
Access epoch. Includes two MPIWCEXIT events, one for each call to `MPI_Win_start` and `MPI_Win_complete`, plus all GET_1TE events in between at the same location and referencing the same window to represent all get operations belonging to this epoch. Note that put operations are represented by their respective PUT_1TE events inside the exposure epoch.
Exposure epoch. Includes two MPIWCEXIT events, one for each call to `MPI_Win_post` and `MPI_Win_wait`, plus all PUT_1TE events in between at the same location and referencing the same window to represent all put operations belonging to this epoch. Note that get operations are represented by their respective GET_1TE events inside the access epoch.
Epoch pair. Union of an access epoch at location l with a corresponding exposure epoch at location k but without any communication events not related to communication between l and k.
Access transaction. Union of an access epoch at location l with all corresponding exposure epochs at locations $k_1, \ldots, k_n$, but without any communication events not related to communication between l and $k_1, \ldots, k_n$. Figure 2 (left) shows an access transaction involving one access and two exposure epochs.
Exposure transaction. Union of an exposure epoch at location l with all corresponding access epochs at locations $k_1, \ldots, k_n$, but without any communication events not related to communication between l and $k_1, \ldots, k_n$. Figure 2 (right) shows an exposure transaction involving one exposure and two access epochs.

Matching GATS-based patterns requires the recognition of the above structures in the event trace. For this purpose, we have defined a hierarchical system of state sequences

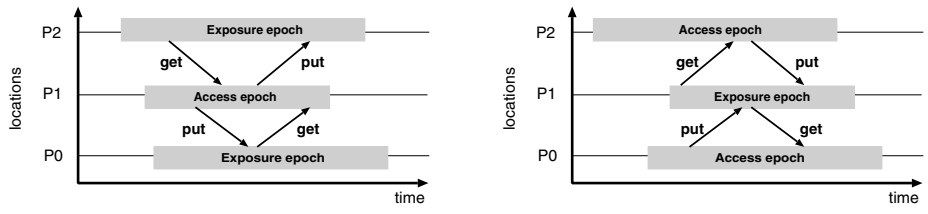


Fig. 2. An access transaction (left) and an exposure transaction (right)

that detects higher-level structures step-by-step based on lower-level structures already detected.

At the bottom, there are two state sequences $\mathfrak{A}^{l,w}$ and $\mathfrak{E}^{l,w}$ responsible for collecting all events belonging to an access or exposure epoch taking place at location l and referring to window w. The separation by window ensures that epochs belonging to the same window do not overlap in time at the same location. Once the event set describing an epoch is complete, the state is cleared upon the occurrence of the next event.

Completed epochs are combined into epoch pairs by a state sequence $\bar{\mathfrak{P}}^{k,l,w}$, which is defined for a target location k, an origin location l, and a window w. Before combining the two epochs, however, all events not related to communication between the two sides of the pair are removed. Again, after completion of the whole pair, the state is cleared. Different from our earlier sequence, the states of this sequence contain sets of event sets. This is necessary to express the hierarchical grouping of events typical for GATS transactions that consist of zero or more data transfer events enclosed by synchronization operations at each participating location. The auxiliary function $epoch\_pair(e, l)$ extracts a complete epoch pair as a flat set if e constitutes the last event of a pair with l being the location of the counter epoch.

The next level of composition is achieved through an auxiliary function $expta(e, \bar{P})$ that can be applied to an event e and a set of epoch pairs $\bar{P}$ and that returns all epoch pairs belonging to an exposure transaction if e constitutes the last event of this transaction. Using this and the function above, we have defined a state sequence $\bar{\mathfrak{E}}^{l,w}$ for a location l and a window w that successively adds epoch pairs as they are finished until a full exposure transaction has been completed, which then can be extracted using $expta(e, \bar{P})$. l denotes the location of the access epoch. Similarly, we have defined a function $accta(e, \bar{P})$ and a state sequence $\bar{\mathfrak{A}}^{l,w}$ to identify whole access transactions for later performance analysis.

6 One-Sided Patterns

Now, we use the abstractions defined in the previous section to specify complex inefficiency patterns spanning more than one process as a prerequisite for their automatic detection in event traces. The general structure of a pattern consists of a *root event* described by a simple test condition and zero or more constituents that can be located from the root event using the abstractions. The root event is the latest constituent event because the search for the remaining ones occurs backwards for efficiency reasons. An additional rule specifies how to quantify the pattern's performance impact (i.e., the time lost). Since it is the most complicated part of MPI-2 one-sided communication, we have focused mostly on patterns related to GATS synchronization.

A major challenge in specifying appropriate detection mechanism has been the fact that the latest event in an epoch pair can either belong to an access or an exposure epoch. This can lead to complicated case distinctions that are not necessary for traditional point-to-point communication, where a send event always precedes a receive event. Another important difference to point-to-point communication arises from the one-to-many relationships existing between access and exposure epochs involving more than two processes. For example, during an exposure epoch, a window may be accessed

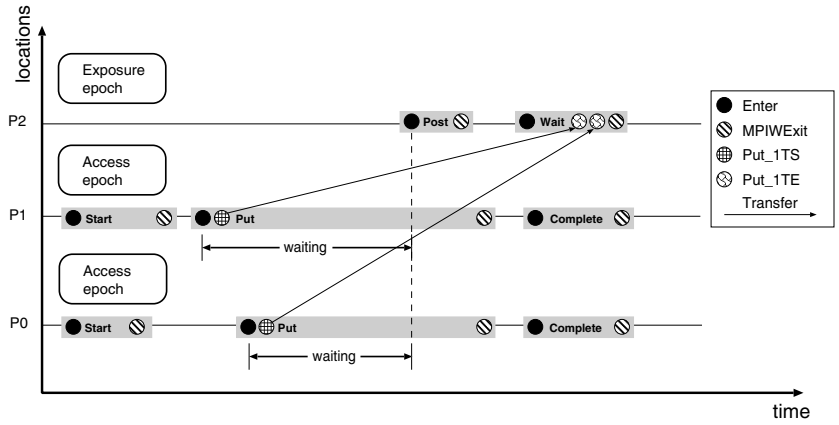


Fig. 3. Early Transfer: `MPI_Get/Put()` blocks during an access epoch until the related exposure epoch is started with `MPI_Win_post()`

by multiple processes each passing through a separate access epoch according to our definition above.

Early Transfer. This pattern describes a situation that may happen when communicating in GATS mode. `MPI_Get/Put()` blocks during an access epoch until the related exposure epoch is started with `MPI_Win_post()` (Figure 3). Recognizing the pattern requires considering epoch pairs. The root event is the last event of an epoch pair and is of type MPIWEXIT. It either completes the access or the exposure epoch and, therefore, either belongs to `MPI_Win_complete()` or to `MPI_Win_wait/test()`.

The complete set of epoch pairs finished by the root event is determined by calculating $epoch\_pair(root, l)$ for every location l being a member of the partner group recorded with the root event. If the root event belongs to an exposure epoch, the pattern covers all corresponding access epochs already finished.

The waiting time is counted from the start of an access operation within the access epoch until the corresponding post operation has been issued during the matching exposure epoch. The begin of the access operation is identified using the pointer attributes $startptr$ and $originptr$ in the case of a get operation.

Early Wait. This pattern represents the premature request to finish an exposure epoch using `MPI_Win_wait()` and is depicted in Figure 4. We consider the request as premature if it was posted before the last access epoch's closure has been requested using `MPI_Win_complete()` within the same exposure transaction.

The recognition of this pattern requires the recognition of an exposure transaction. Two cases must be distinguished: (i) the transaction is completed by an exposure epoch or (ii) the transaction is completed by an access epoch. In the first case, the root event is the MPIWEXIT event of the wait operation and the full transaction is easily obtained by applying $expta()$ to the root event and $\bar{\mathfrak{E}}^{l,w}$ with l being the location of the root event.

In the second case, the root event is the MPIWEXIT event of a complete operation and, therefore, finishes an access epoch. Now, the detection mechanism needs to

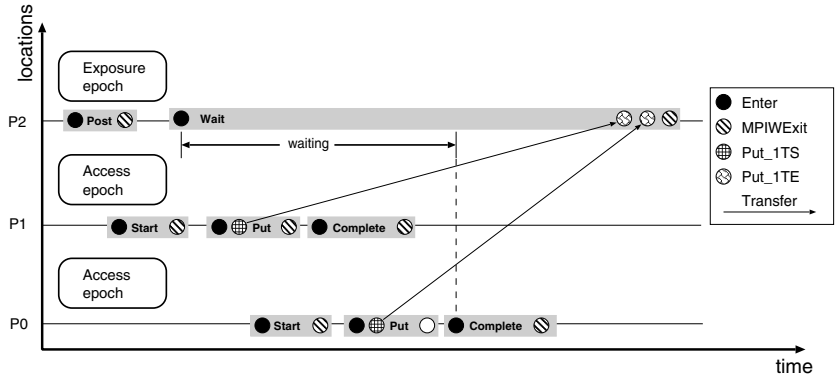


Fig. 4. Early Wait: `MPI_Win_wait()` blocks during completion of an exposure epoch until all related access epochs are completed

find all exposure transactions finished with this access epoch. This is accomplished by iterating over all exposure epochs belonging to epoch pairs completed by the root event and extracting completed exposure transactions from $\bar{\mathfrak{E}}^{k,w}$ using the $expta()$ function. The exposure epochs are found by means of $\bar{\mathfrak{P}}^{k,l,w}$ with l being the location of the root event and k being a location in the root event's partner group. Since the exposure transactions we are looking for have been completed by the root event, we need to consider $\bar{\mathfrak{E}}^{k,w}$ at the time of the root event. The waiting time is the period between the start of the wait until the beginning of the latest complete operation in the transaction.

Late Complete. If a process delays the completion of an access epoch by performing work between the last access and the complete operation and the wait operation has already been posted, a situation named Late Complete occurs (Figure 5). It is actually a sub-property of Early Wait. This pattern considers an exposure transaction and

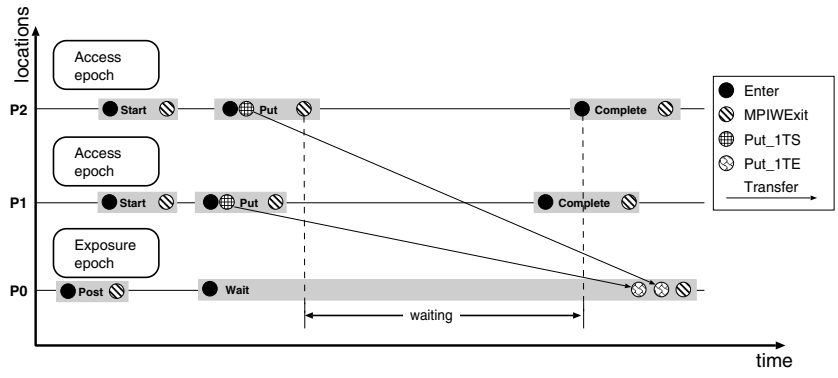


Fig. 5. Late Complete: Describes the time wasted between the last access and the call to the corresponding `MPI_Win_complete()` operation

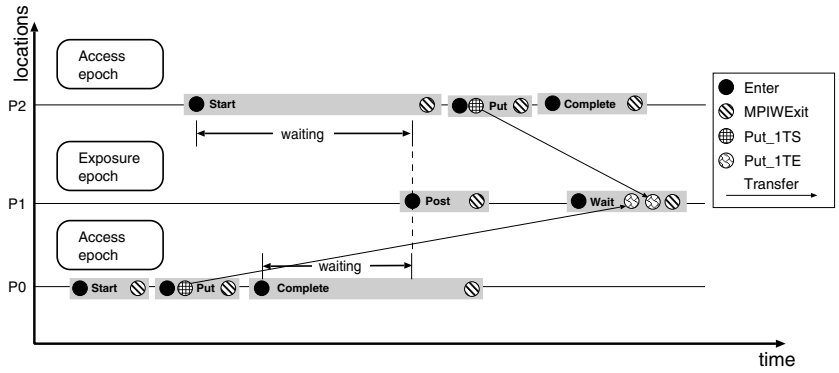


Fig. 6. Late Post: `MPI_Win_start()` or `MPI_Win_complete()` block because the corresponding exposure epoch has not started yet

measures the time spent in the wait operation between exiting the last put or get and entering the corresponding complete (or the latest complete if the last get/put is not unique). The recognition of the exposure transaction is similar to Early Wait.

Late Post. Refers to access-sided synchronization operations that block until access is granted by an exposing process (Figure 6). Depending on the MPI implementation, this may happen either during `MPI_Win_start()` or during `MPI_Win_complete()`. Since the exact blocking semantics are usually not known to a performance tool, our pattern counts time spent in both operations before the earliest post call within the same access transaction is issued in the case that `MPI_Win_start()` does not block. Then, however, the time spent in the start operation will be small and the resulting inaccuracy negligible. Whereas the semantics of the pattern are closer to Early Transfer, its recognition is very similar to Early Wait, only that it requires the recognition of an access transaction using $\bar{\mathfrak{A}}^{l,w}$. Like Early Wait, this pattern needs to distinguish two cases: (i) the root event finishes an access epoch or (ii) the root event finishes an exposure epoch, in which case the access transactions have to be identified by iterating over all related access epochs.

Wait at Fence. Whereas the previous patterns all refer to GATS synchronization, this pattern covers the simpler case of synchronization with `MPI_Win_fence()`. Since fence normally[2] implies a barrier, waiting times occur if the fence is not reached simultaneously by all participating processes (Figure 7). Early processes have to wait for the latest one. The recognition of Wait at Fence is accomplished using $\mathfrak{C}^g < MPIWCExit >$, which collects collective window operation instances. After retrieving such an instance using $complete < MPIWCExit > ()$ and identifying the latest entry into the operation, the waiting times of different processes can be easily determined.

[2] The internal barrier can be avoided by passing additional "hints" to the fence call as a second parameter.

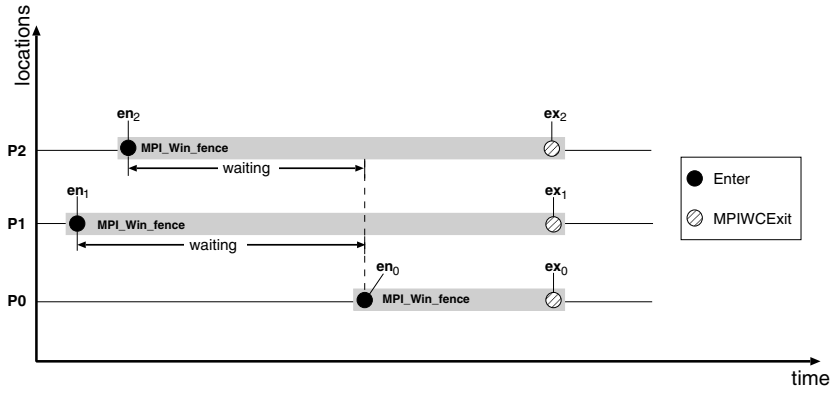


Fig. 7. Wait at Fence: Time spent waiting in front of a synchronizing `MPI_Win_fence()` operation

7 Conclusion

To the best of our knowledge, this is the first systematic approach of automatically identifying wait states related to MPI-2 one-sided communication in event traces. Building upon our earlier framework to identify wait states in traditional two-sided and collective communication, we have defined new abstractions representing higher-level events related to one-sided operations. These abstractions serve as a useful prerequisite to specify inefficiency patterns in a way facilitating their automatic detection in the event stream.

A major difficulty that has been solved within our new framework is the fact that one-sided communication is accomplished in complex sequences of synchronization and communication, where the notion of send and receive operations is replaced by the notion of access and exposure epochs comprising both synchronization and access operations. Also, a single epoch may perform communication with an entire group of processes, which requires the recognition of all counter epochs performed by members of this group. In addition, the root event from where the constituents of a pattern may be located may reside on either side of an epoch pair, which involves complex case distinctions on the side of the detection mechanism.

To demonstrate the usefulness of our framework, we have specified several complex patterns of inefficient behavior targeting, in particular, general active target synchronization, which can be challenging for programmers. Meanwhile, we have completed the implementation of all KOJAK modules necessary for the instrumentation, measurement, conversion, and analysis of parallel applications based on MPI-2 RMA and we have a prototype version for SHMEM programs. Figure 8 shows a summary of the currently implemented pattern hierarchy. We have also extended our internal test suite to cover one-sided communication and used it to verify our implementation. As a next step, we need to evaluate the relevance of these patterns using real-world applications.

Finally, we hope that some of the complexity in the analysis can be avoided, when transferring this approach to the new parallel analyzer architecture developed in the SCALASCA [12] project. By exploiting distributed memory and parallel processing

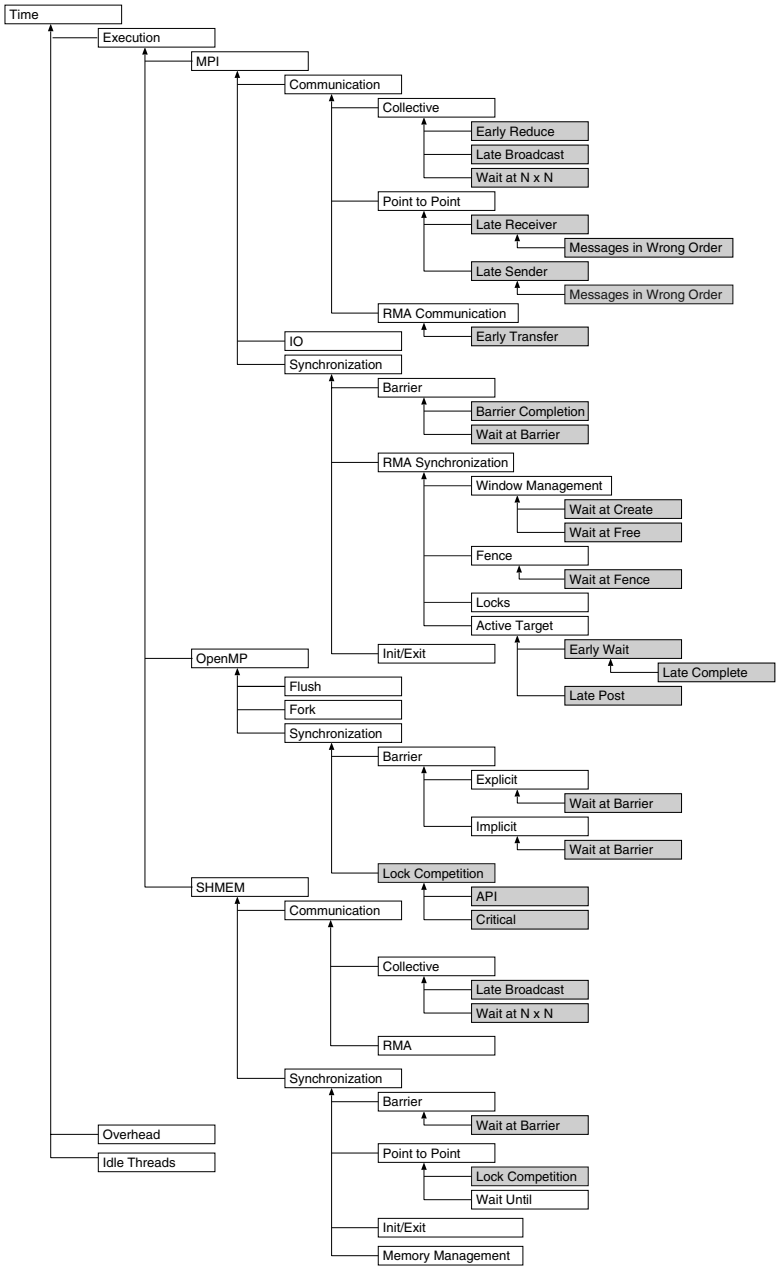


Fig. 8. Performance properties defined by KOJAK. White boxes indicate performance properties based on summary information which could also be provided by a profiling tool. However, the second type, indicated by gray boxes, involves idle times that can only be determined by comparing the chronological relation between individual events.

capabilities, the analysis is carried out entirely in main memory, relaxing the efficiency-motivated forward-analysis requirement imposed by our previous sequential analysis approach.

References

1. Wolf, F., Mohr, B.: Specifying Performance Properties of Parallel Applications Using Compound Events. Parallel and Distributed Computing Practices **4** (2001) 301–317 Special Issue on Monitoring Systems and Tool Interoperability.
2. Message Passing Interface Forum: MPI-2: Extensions to the Message-Passing Interface (1997) http://www.mpi-forum.org.
3. Mirin, A., Sawyer, W.: A scalable implementation of a finite volume dynamical core in the Community Atmosphere Model. International Journal of High Performance Computing Applications **19** (2005) 203–212
4. Wolf, F., Mohr, B.: Automatic performance analysis of hybrid MPI/OpenMP applications. Journal of Systems Architecture **49** (2003) 421–439 Special Issue "Evolutions in parallel distributed and network-based processing".
5. Mohr, B., Kühnal, A., Hermanns, M.A., Wolf, F.: Performance Analysis of One-sided Communication Mechanisms. In: Proceedings of Parallel Computing (ParCo), Malaga, Spain (2005) Mini-Symposium "Tools Support for Parallel Programming".
6. Hermanns, M.A., Mohr, B., Wolf, F.: Event-based Measurement and Analysis of One-sided Communication. In: Proc. of the European Conference on Parallel Computing (Euro-Par). Volume 3648 of Lecture Notes in Computer Science., Lisboa, Portugal, Springer (2005) 156–165
7. Mohror, K., Karavanic, K.L.: Performance Tool Support for MPI-2 on Linux. In: Proc. of the Supercomputing Conference (SC), Pittsburgh, PA (2004)
8. Malony, A.D., Shende, S.: Performance Technology for Complex Parallel and Distributed Systems. In Kacsuk, P., Kotsis, G., eds.: Quality of Parallel and Distributed Programs and Systems, Nova Science Publishers, Inc., New York (2003) 25–41
9. Pallas/Intel: Intel Trace Collector (2006) `http://www.intel.com/software/products/cluster/tcollector/`
10. Wolf, F.: Automatic Performance Analysis on Parallel Computers with SMP Nodes. PhD thesis, RWTH Aachen, Forschungszentrum Jülich (2003) ISBN 3-00-010003-2.
11. Kühnal, A.: Performance Properties for One-Sided Communication Mechanisms. Diploma Thesis. Forschungszentrum Jülich (2005) In German.
12. Forschungszentrum Jülich: SCALASCA (2006) `http://www.scalasca.org`.