John von Neumann Institute for Computing

Performance Analysis on One-sided Communication Mechanisms

B. Mohr, A. Kühnal, M.-A. Hermanns, F. Wolf

published in

Parallel Computing:
Current & Future Issues of High-End Computing,
Proceedings of the International Conference ParCo 2005,
G.R. Joubert, W.E. Nagel, F.J. Peters, O. Plata, P. Tirado, E. Zapata (Editors),
John von Neumann Institute for Computing, Jülich,
NIC Series, Vol. **33**, ISBN 3-00-017352-8, pp. 885-892, 2006.

http://www.fz-juelich.de/nic-series/volume33

Performance Analysis of One-sided Communication Mechanisms

B. Mohr[a], A. Kühnal[a], M.-A. Hermanns[a], F. Wolf[a]

[a]Forschungszentrum Jülich, ZAM, 52425 Jülich, Germany

Abstract. Performance analysis of parallel programs requires information about the dynamic behavior of all participating processes. The dynamic behavior can be modeled as a stream or trace of events. The events are chosen in such a way that they represent important aspects in the execution of the application on a level of abstraction suitable for the analysis task. Based on this idea, the KOJAK toolkit for performance analysis records and analyzes the activities of MPI-1 point-to-point and collective communication.

This paper describes the integration of performance measurement and analysis methods for remote memory access (RMA) or one-sided communication into the KOJAK toolkit, in particular for the MPI-2 and SHMEM interfaces. We introduce the underlying event model used to represent the dynamic behavior of RMA operations and show that our model reflects the relationships between communication and synchronization more accurately than existing models. Then, we present event patterns which are used by KOJAK to locate inefficient situations in a program's dynamic remote memory access behavior.

1. Introduction

Remote memory access (RMA) describes the ability of a process to directly access a part of the memory of a remote process, without explicit participation of the remote process in the data transfer. As all parameters for the data transfer are determined by one process, it is also called *one-sided* or *single-sided* communication. On platforms with special hardware providing efficient RMA support, one-sided communication is often made available to the programmer in the form of libraries, for example SHMEM (Cray/SGI) or LAPI (IBM). However, these libraries are typically platform- or at least vendor-specific.

This is one of the reasons why the MPI forum decided to define a portable one-sided communication interface as part of MPI-2. The Message Passing Interface (MPI) was defined by a group of vendors, government laboratories and universities in 1994 as a community standard [1]. This has become known as MPI-1. In 1997, a second version of the interface (MPI-2) was defined, which added support for parallel I/O, dynamic process creation, and one-sided communication [2].

Until recently there was only rare usage of RMA features in scientific applications and, therefore, the demand for performance tools in this area was limited. As more and more programmers adopt the new features to improve the performance of their codes, this is expected to change. For example, NASA researchers report a 39% improvement in throughput after replacing MPI-1 non-blocking with MPI-2 one-sided communication in a global atmosphere simulation program [3].

KOJAK, our toolkit for automatic performance analysis [9], is jointly developed by the Central Institute for Applied Mathematics of the Research Centre Jülich and by the Innovative Computing Laboratory of the University of Tennessee. It is able to instrument and analyze OpenMP constructs and MPI-1 calls. In this paper we report on the integration of performance analysis methods for one-sided communication into the existing toolkit. We introduce an extension to our event model that realistically represents the dynamic behavior of MPI-2 RMA operations in the event stream. We show that our model reflects the relationships between communication and synchronization more

accurately than existing models. The model is general enough to also cover alternate, but simpler, RMA interfaces. In addition, we present KOJAK's new performance properties used to analyze MPI-2 and SHMEM parallel programs. *Performance properties* are event patterns which are used by KOJAK to locate inefficient situations in a program's dynamic behavior.

In our new prototype implementation, we added support for measurement and analysis of parallel programs using MPI-2 and SHMEM one-sided communication and synchronization. We are also able to handle Co-Array Fortran programs [8], a small extension to Fortran 95 that provides a simple, explicit notation for one-sided communication and synchronization, expressed in a natural Fortran-like syntax. Details of this work can be found in [10].

The remainder of the paper is organized as follows: First, we summarize related work in Section 2. Section 3 gives a short description of the MPI-2 RMA communication and synchronization functions. In Section 4, we present our event model, which realistically represents the dynamic behavior of RMA operations. KOJAK's hierarchy of performance properties for the analysis of RMA communication and synchronization is described in Section 5. Finally, we present conclusions and future work in Section 6.

2. Related Work

Currently, there are only very few tools which 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 [4]. 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 analysis of SHMEM programs. The very portable TAU performance analysis tool environment [5] 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) [6] records MPI execution traces. When used with MPI-2, only a subset of the RMA functions are traced. It also traces the actual RMA transfers, but misrepresents their semantics, as defined by MPI-2. 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.

3. 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 access to arbitrary memory locations with RMA operations, but only to designated parts of a process's memory, the so-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. MPI denotes by *origin* 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` 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 user code. 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, is followed by any number of remote read, write, or update operations in 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*. There is a one-to-one mapping between access epochs on origin processes and exposure epochs on target processes. Distinct epochs for a window on 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 on an origin process or an exposure epoch on a target process are started and completed by such a call.

General Active Target Synchronization (GATS)**:** Here synchronization is minimized: 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 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`. The post call specifies the group of origin processes for that epoch.

Locks: The `MPI_Win_lock` and `MPI_Win_unlock` calls provide shared and exclusive locks. They are used for passive target communication.

In all cases, data read or written is only accessible from user code after the "closing" synchronization call. It is implementation-defined whether some of the described calls are blocking or nonblocking; for example, in contrast to other shared memory programming paradigms, the lock call does not need to be blocking. For a complete description of MPI-2 RMA communication see [2].

4. An Event Model for One-sided Communication

In this section, we summarize the event types and event models used by KOJAK to realistically represent the behavior of MPI-2 as well as Co-Array Fortran and vendor-specific RMA operations. For a more detailed description of the models and the implementation of KOJAK's monitoring components for one-sided communications see [11]. For a complete description of KOJAK's event types for MPI-1 and OpenMP and of its analysis features see [7, 9].

KOJAK's Event Types for Modeling One-sided Communication

Abstraction	Event type	Type specific attributes
Start / end / origin of RMA one-sided transfers	PUT_1TS	window id, rma id, length, dest loc
	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 MPI GATS function	MPIWEXIT	window id, region id, group id
Leaving MPI collective RMA function	MPIWCEXIT	window id, region id, comm id
Locking / unlocking a MPI window	WLOCK	window id, lock loc, type
	WUNLOCK	window id, lock loc

For the analysis of parallel scientific applications, events that capture the most important aspects of the parallel programming paradigm used (e.g., MPI or OpenMP) and the entering and leaving of surrounding user regions (e.g., functions or loops) are typically defined. In the case of collective MPI functions and OpenMP constructs, instead of "normal" EXIT events, special collective events are used to capture the attributes of the collective operation (e.g., the communicator). MPI-1 point-to-point messages are modeled as pairs of SEND and RECV events. In OpenMP applications, FORK and JOIN events mark the start and end of parallel regions and ALOCK and RLOCK events mark the acquisition and release of locks.

In order to be able to also analyze RMA operations, we defined the new event types shown in Table 1. 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 identifier 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 processors. For collective MPI-2 RMA functions we use a MPIWCEXIT event and record the communicator which defines the group of processes which participate in the collective operation. Finally, MPI window lock and unlock operations are marked with WLOCK and WUNLOCK events.

Based on these event types and their attributes, we introduced two event models for describing the dynamic behavior of RMA operations. For each model, we describe its basic features and analyze its strengths and weaknesses.

4.1. Basic Model

In the first and simpler model, it is assumed that the RMA communication functions have a blocking behavior, that is, the data transfer is completed before the function is finished. Also, RMA synchronization functions are treated as if they were independent of the communication functions.

The invocations of RMA communication and synchronization functions are modeled with ENTER and EXIT events. To model the actual RMA transfer, the transfer-start event is associated with the source process immediately after the beginning of the corresponding communication function. Accordingly, the end event is associated with the destination process shortly before the exit of the (same) function.

The advantage of this model is a straight-forward implementation because events and their at-

tributes can be recorded at exactly the place and time where they are supposed to appear in the model. We use this model for analyzing SHMEM and Co-Array Fortran programs. However, for MPI-2, this model is not sufficient because it ignores the necessary synchronization, as described in Section 3. Since the end-of-transfer event is placed before the end of the communication function, the transfers are recorded as completed even when this is not true, for example, in the case of a nonblocking implementation. Even if the implementation is blocking, it still does not reflect the user-visible behavior. Therefore, in case of MPI-2, we use an extended model, which is described in the next subsection.

4.2. Extended Model

The *extended model* observes the MPI-2 synchronization semantics and, therefore, better reflects the user-visible behavior of MPI-2 RMA operations. The end of fences and GATS calls is now modeled with MPIWCEXIT or MPIWEXIT respectively in order to capture their collective nature. The transfer-start event is still located in the source process immediately after the beginning of the corresponding communication function (as it is in the basic model). However, the transfer-end event is now placed in the destination process shortly before the exit of the RMA synchronization function which completes the transfer according to the MPI-2 standard rules. The extended model removes all disadvantages of the basic model, and for most MPI-2 implementations (which have a non-blocking behavior), it is even closer to reality. However, the model is more complex and the events can no longer be recorded at the location where they appear in the model. Therefore, a complex post-processing of the collected event trace becomes necessary.

5. Performance Properties of One-sided Communication and Synchronization

In this section we describe the analysis KOJAK is performing for execution traces of applications using one-sided communication. KOJAK's analyzer, named EXPERT [12], attempts to prove *performance properties* for one execution of a parallel application and to quantify them according to their influence on the performance. A performance property characterizes a class of performance behavior and is specified in terms of a *compound event*, which the analyzer tries to detect in an event trace. A compound event is a set of events matching a specific execution pattern, whose constituents are connected by relationships and constraints. For each property, EXPERT calculates a *severity* measure indicating the fraction of the total execution time spent on that property and, thus, allows the correlation of different properties in a single view.

EXPERT organizes the performance properties in a hierarchy. The upper levels of the hierarchy (i.e., those that are closer to the root) correspond to more general behavioral aspects such as time spent in MPI functions. The deeper levels correspond to more specific situations such as time lost due to blocking communication. Figure 2 shows the hierarchy of predefined performance properties based on time measurements that are supported by the current version (2.2) of EXPERT. It also supports hierarchical analysis based on hardware counter metrics [14].

The set of performance properties consists of two types. The first type, which constitutes the upper layers of the hierarchy and which is indicated by white boxes, is based on summary information involving, for example, the total execution times of special MPI routines, which could also be provided by a profiling tool. However, the second type, which constitutes the lower layers of the hierarchy and which is indicated by gray boxes, involves idle times that can only be determined by comparing the chronological relation between individual events. A detailed description of the properties for MPI-1 and OpenMP can be found in [9]. In the following, the new set of properties for MPI-2 RMA and SHMEM are presented. An exact mathematical definition of the new properties can be found in [13].

5.1. MPI-2 RMA Performance Properties

The performance properties for MPI-2 RMA are of course part of EXPERT's overall hierarchy for MPI (see Figure 2, upper part) under the categories *Communication* and *Synchronization*. The upper part of the *RMA Synchronization* property tree captures how much execution time is spent on the different MPI-2 synchronization methods *Fence*, *Locks*, and *Active Target*, and on *Window Management* functions which also contain synchronization because of their collective nature.

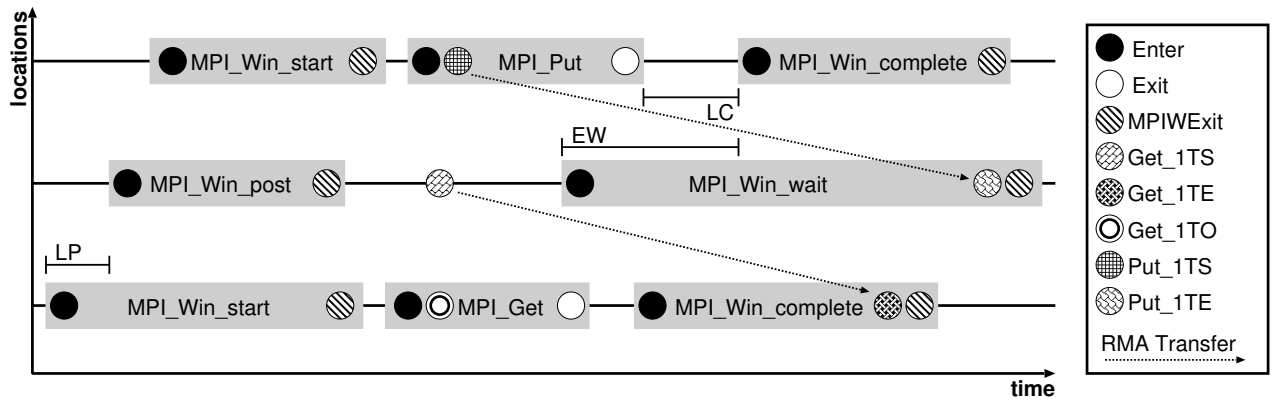

Figure 1. Execution pattern for MPI-2 general active target synchronization.

The first three compound events *Wait at Create*, *Wait at Free*, and *Wait at Fence* simply cover the time spent on waiting in front of these collective operations. The severity for each process is defined by the time from starting the operation until the last participating process arrives. The remaining compound events are related to communication scenarios where general active target synchronization is used. A typical execution sequence is shown in Figure 1. The performance property *Early Wait* is associated with processes providing access to an RMA window (e.g., the middle process in Figure 1) and describes the wasted time waiting for the accesses to complete. The severity is the time spent in `MPI_Win_wait` until the last participating process indicates the end of the accesses by a call to `MPI_Win_complete` (indicated by the interval `EW` in the figure). The subproperty *Late Complete* is associated with the subinterval of this waiting time from the end of the last RMA transfer operation (e.g., the put operation by the upper process in the figure) to the start of the last `MPI_Win_complete` call (marked with `LC`). The property *Late Post* describes the situation where a call to `MPI_Win_start` blocks because the corresponding exposure epoch has not started yet (which is initiated by a `MPI_Win_post` call). As severity we use here the time spent blocked until the start of the post call (see interval `LP` in the figure). On MPI implementations where these synchronization calls are non-blocking, in a similar situation the first RMA transfer call would block. In this case, we call the property *Early Transfer*. It is a subproperty of RMA *Communication* as the blocking occurs during a communication call.

5.2. SHMEM Performance Properties

The performance properties for the SHMEM programming paradigm are modeled after the corresponding properties for MPI (see Figure 2, lower part). The higher level of the SHMEM property hierarchy again captures how much execution time is spent on different parts of the SHMEM programming model. It is either *Communication*, divided into collective and RMA, or *Synchronization* which is broken down to *Barrier*, *Point-to-Point*, *Init/Exit*, or *Memory Management*. The compound patterns *Late Broadcast*, *Wait at NxN*, *Wait at Barrier*, and *Lock Competition* are defined exactly like the corresponding MPI or OpenMP properties.

6. Conclusion and Future Work

We defined two event models describing the dynamic behavior of parallel applications involving RMA transfers. The basic model can be used for RMA implementations with blocking behavior, that is, vendor-specific one-sided communication libraries like SHMEM or language extension like Co-Array Fortran and Unified Parallel C (UPC). For MPI, we defined an extended event model that reflects the user-visible behavior as specified by the MPI-2 standard. We also defined RMA-related performance properties which represent inefficient behavior of RMA communication and synchronization. We implemented an extension to the KOJAK performance analysis toolset to instrument and trace applications based on MPI-2 and SHMEM communication and synchronization and to analyze the collected traces using the EXPERT automatic trace analysis component of KOJAK.

References

[1] M. Snir, S. Otto, S. Huss-Lederman, D. Walker, and J. Dongarra. *MPI - the Complete Reference, Volume 1, The MPI Core*. 2nd ed., MIT Press, 1998.

[2] W. Gropp, S. Huss-Lederman, A. Lumsdaine, E. Lusk, B. Nitzberg, W. Saphir, and M. Snir. *MPI - the Complete Reference, Volume 2, The MPI Extensions*. MIT Press, 1998.

[3] A. Mirin and W. Sawyer. A scalable implementation of a finite volume dynamical core in the Community Atmosphere Model. *International Journal of High Performance Computing Applications*, Vol. 19, No. 3, 203-212, 2005.

[4] K. Mohror and K.L. Karavanic. Performance Tool Support for MPI-2 on Linux. In *Proceedings of SC'04*, Pittsburgh, PA, Nov. 2004.

[5] S. Shende, A. D. Malony, J. Cuny, K. Lindlan, P. Beckman, and S. Karmesin. Portable Profiling and Tracing for Parallel Scientific Applications using C++. In *Proceedings of the SIGMETRICS Symposium on Parallel and Distributed Tools*, pp. 134–145. ACM, Aug. 1998.

[6] Pallas/Intel. *The Intel Trace Collector*. 2004.
→ http://www.intel.com/software/products/cluster/tcollector/

[7] F. Wolf. *Automatic Performance Analysis on Parallel Computers with SMP Nodes*. Dissertation, NIC Series, Vol. 17, Forschungszentrum Jülich, 2002.

[8] R. W. Numrich and J. K. Reid. Co-Array Fortran for Parallel Programming. *ACM Fortran Forum*, 17(2), 1998.

[9] F. Wolf and B. Mohr. Automatic Performance Analysis of Hybrid MPI/OpenMP Applications. *Journal of Systems Architecture, Special Issue 'Evolutions in parallel distributed and network-based processing'*, 49(10–11):421–439, Nov. 2003.

[10] B. Mohr, L. DeRose, and J. Vetter. A Performance Measurement Infrastructure for Co-Array Fortran. In *Proceedings of Euro-Par 2005*, Springer, LNCS 3648, pp. 146–156, Lisboa, Portugal, Sep. 2005.

[11] M.-A. Hermanns, B. Mohr, and F. Wolf. Event-Based Measurement and Analysis of One-Sided Communication. In *Proceedings of Euro-Par 2005*, Springer, LNCS 3648, pp. 156–166, Lisboa, Portugal, Sep. 2005.

[12] F. Wolf, B. Mohr, J. Dongarra, and S. Moore. Efficient Pattern Search in Large Traces through Successive Refinement. In *Proceedings of Euro-Par 2004*, Springer, LNCS 3149, pp. 47–54, Pisa, Italy, Sep. 2004.

[13] A. Kühnal. *Performance Properties for One-Sided Communication Mechanisms* (In German). Diploma Thesis. Forschungszentrum Jülich, 2005.

[14] B. Wylie, B. Mohr, F. Wolf. Holistic hardware counter performance analysis of parallel programs. In *Proceedings of Parallel Computing 2005* (ParCo 2005), Malaga, Spain, Sep 2005 .

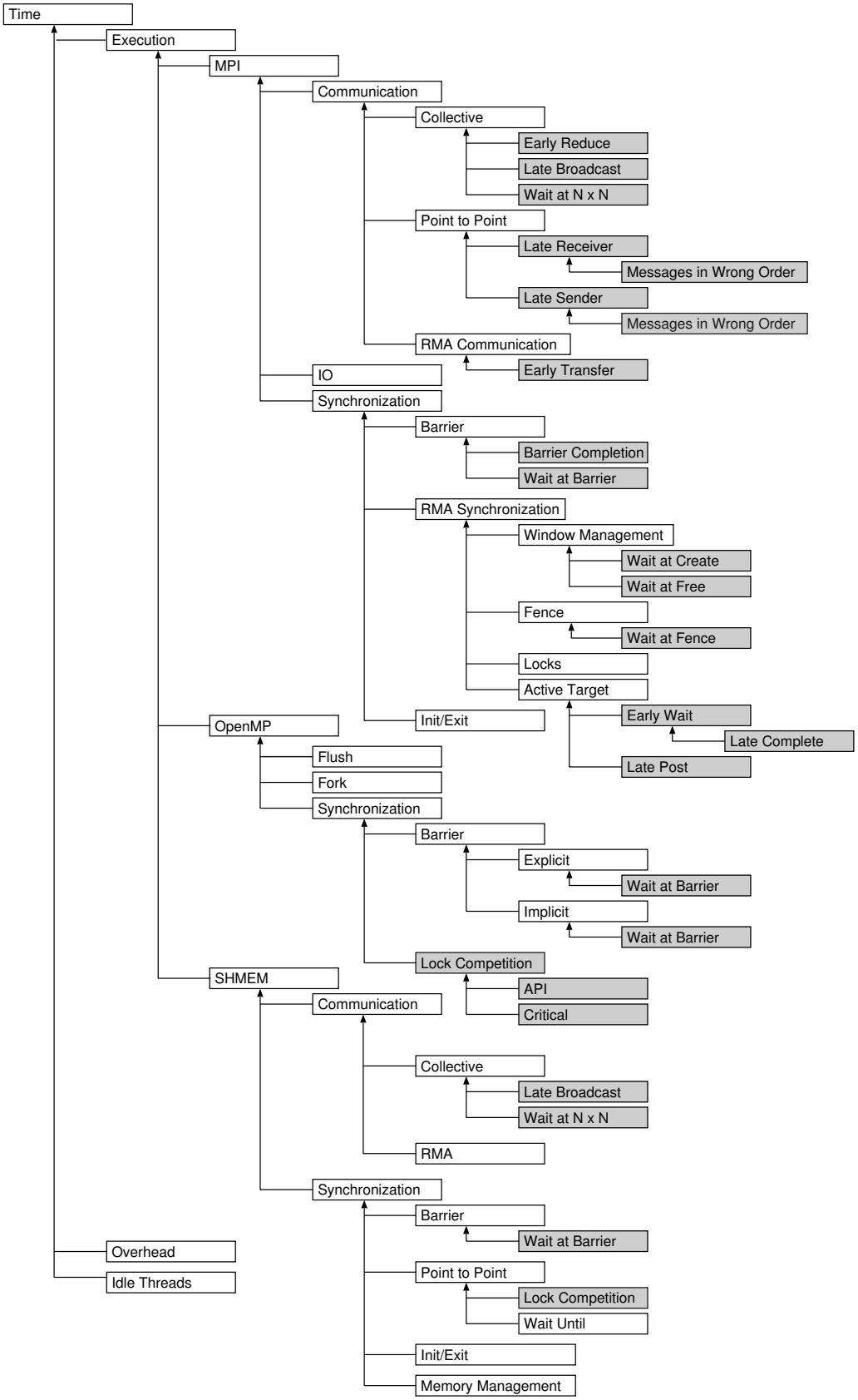

Figure 2. Performance properties defined by KOJAK