Event-based Measurement and Analysis of One-sided Communication

Marc-André Hermanns[1], Bernd Mohr[1], and Felix Wolf[2]

[1] Forschungszentrum Jülich,
Zentralinstitut für Angewandte Mathematik,
52425 Jülich, Germany
{m.a.hermanns,b.mohr}@fz-juelich.de

[2] University of Tennessee, ICL
1122 Volunteer Blvd Suite 413
Knoxville, TN 37996-3450, USA
fwolf@cs.utk.edu

Abstract. To analyze the correctness and the performance of a program, information about the dynamic behavior of all participating processes is needed. The dynamic behavior can be modeled as a stream of events required for a later analysis including appropriate attributes. Based on this idea, KOJAK, a trace-based toolkit for performance analysis, records and analyzes the activities of MPI-1 point-to-point and collective communication.
To support remote-memory access (RMA) hardware in a portable way, MPI-2 introduced a standardized interface for remote memory access. However, potential performance gains come at the expense of more complex semantics. From a programmer's point of view, an MPI-2 data transfer is only completed after a sequence of communication and associated synchronization calls.
This paper describes the integration of performance measurement and analysis methods for RMA communication into the KOJAK toolkit. Special emphasis is put on the underlying event model used to represent the dynamic behavior of MPI-2 RMA operations. We show that our model reflects the relationships between communication and synchronization more accurately than existing models. In addition, the model is general enough to also cover alternate but simpler RMA interfaces, such as SHMEM and Co-Array Fortran.

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. This distinguishes the one-sided communication from point-to-point messages, where explicit send and receive statements are required on both sides. Providing one-sided in addition to two-sided communication significantly expands the flexibility to chose a communication scheme most suitable for a given problem on a given hardware.

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), LAPI (IBM), or ELAN (Quadrics). However, these libraries are typically platform- or at least vendor-specific. The exception is SHMEM, which is offered by a group of vendors. Since this restricts portable programming, many programmers do not utilize one-sided communication.

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. It is fully supported by all freely-available and commercial MPI implementations and was quickly adopted by the scientific computing community as a de-facto standard. As MPI also provides a standard monitoring interface (PMPI), there is a wide variety of tools for MPI performance analysis and visualization. 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]. However, only now, seven years after its definition, is support for all MPI-2 features portably available for all major parallel computing platforms.

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 communication with MPI-2 one-sided communication in a global atmospheric simulation program [3].

Currently, there are only very few tools which support the measurement and analysis of one-sided communication and synchronization in a portable way on a wider 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, it records enter and exits of only a subset of the RMA functions. It also traces the actual RMA transfers, but misrepresents their semantics, as defined in 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 disadvantages.

KOJAK, our toolkit for automatic performance analysis [10], 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 put special emphasis on the development of a new 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. In addition, the model is general enough to also cover alternate, but simpler RMA interfaces. 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. In addition, we are also able to handle Co-Array Fortran programs [9], 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 [11].

The remainder of the paper is organized as follows: In Section 2 we give a short description of the MPI-2 RMA communication and synchronization functions. In Section 3, we present our event model, which allows the realistic representation of the dynamic behavior of vendor-specific and MPI-2 RMA operations. The extensions to KOJAK components allowing the instrumentation, measurement, analysis, and visualization of parallel programs based on one-sided communication are described in Section 4. Finally, we present conclusions and future work in Section 5.

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 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, 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*. There is a one-to-one mapping between access epochs on origin processes and exposure epochs on target processes. 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 on an origin process or an exposure epoch on 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: 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`. Again, the post call specifies the group of origin processes for that epoch. Data written is only accessible after the wait call, however data can only be read after the complete operation.

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 nonblocking; 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 An Event Model for One-sided Communication

Many performance analysis tools use an *event-based* approach, that is, they instrument user applications only at specific points to collect the performance data they need for their analysis. These points, called *events*, are chosen in a way that they represent important aspects in the dynamic behavior of the application on a level of abstraction suitable for the tools' task. Trace-based tools record the occurrence of events as a stream or *trace* of event records for later analysis.

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) are defined. Often, to provide a context for events representing specific actions related to a parallel programming interface, the entering and leaving of surrounding user regions (e.g., functions, loops or basic blocks) are also captured.

Abstraction	**Event type**	**Type specific Attributes**
Entering / leaving a region (a function)	ENTER	region id
	EXIT	region id
Leaving a collective MPI	MPICEXIT	region id, comm id, root loc, sent, revd
or OpenMP region	OMPCEXIT	region id
Sending / receiving a message	SEND	dest loc, tag, comm id, length
	RECV	src loc, tag, comm id, length
Start / end of OpenMP parallel region	FORK	
	JOIN	
Acquiring / releasing an OpenMP lock	ALOCK	lock id
	RLOCK	lock id
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 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

Table 1. KOJAK's Event Types

Table 1 lists all event types used by the KOJAK performance analysis toolset. In the upper half, the already existing events for modeling MPI-1 and OpenMP behavior are shown. in addition to type-specific attributes for each event we also collect the *timestamp* and *location* which describe when and where the event occurred. For user regions, MPI functions, and OpenMP constructs and runtime functions, we record which region was entered or left. 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. For MPI this is the communicator, the root process, and the amounts of data sent and received during this operation. MPI-1 point-to-point messages are modeled as pairs of SEND and RECV events with the source or destination of the message, the tag and communicator used, and the amount of data transferred being attributes. In OpenMP applications, FORK and JOIN events mark the start and end of parallel regions and ALOCK and RLOCK events the acquisition and release of locks. For a complete, more detailed description of KOJAK's event types and of its analysis features see [7,10]. A similar event model is also used by most other event-based tools (e.g., by TAU).

In order to be able to also analyze RMA operations, we defined new event types to realistically model the behavior of MPI-2 as well as Co-Array Fortran and vendor-specific RMA operations. These new event types are shown in the bottom compartment of 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 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 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.

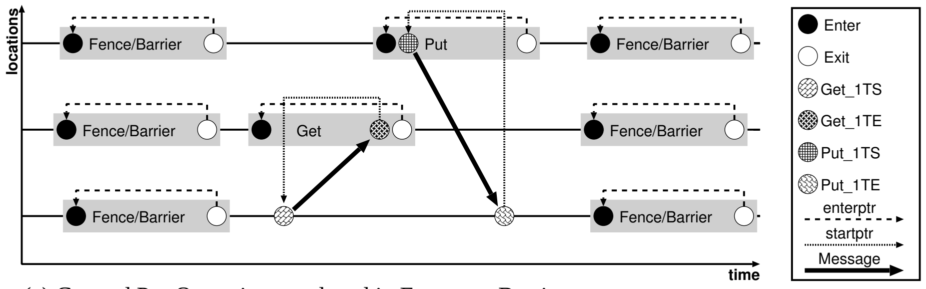
Based on these event types and their attributes, we now introduce two event models for describing the dynamic behavior of RMA operations. For each model, we describe its basic features and analyze its strengths and weaknesses. To illustrate the location of events and relationships between them, we use simple *time-line* diagrams. In these diagrams, time progresses from left to right. Event instances are shown as colored circles on different "time lines", one for each process involved in the execution. Invocations of functions are shown as gray boxes with the name of the function executed. Finally, relationships between events are displayed as arrows with different line styles. Following KOJAK conventions [7], relationships are always named with a suffix `ptr` (for pointer) and always point from a later event back to an earlier event related to the later one. This allows for an efficient analysis process with a single pass through the event trace.

3.1 Basic Model

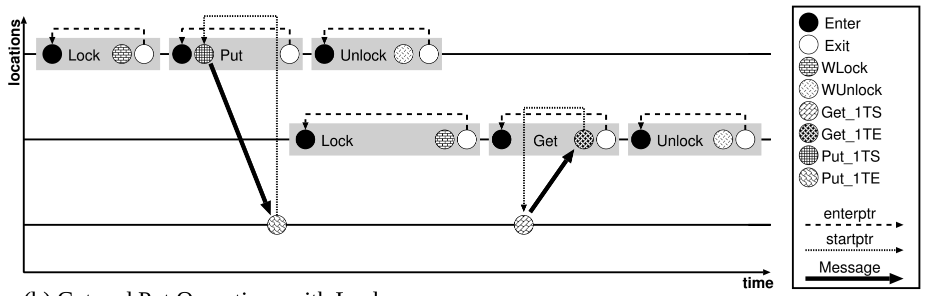
In the first and simpler model, it is assumed that the RMA communication functions have a blocking behavior, that is, the data transfer is completed before the function is finished. Also, RMA synchronization functions are treated as if they were independent of the communication functions.

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 begin of the corresponding communication function. Accordingly, the end event is associated with the destination process shortly before the exit of the function. Finally, we define a relationship *startptr* which allows analysis tools to easily locate the matching start event from the transfer end event. Figure 1 shows the model for typical usage patterns of one-sided communication. A sequence of get and put operations is guarded by fences, barriers, or lock/unlock operations. The message line shown in the picture is not part of the model and only shown for clarity.

The advantage of this model is a straight-forward implementation because events and their attributes 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 2. Since the end-of-transfer event is placed before the end of the communication function, the transfers are recorded as completed even when, for example, in the case of a nonblocking implementation, this is not true. Even if the implementation is blocking, it still does not reflect the user-visible be-



(a) Get and Put Operations enclosed in Fences or Barriers



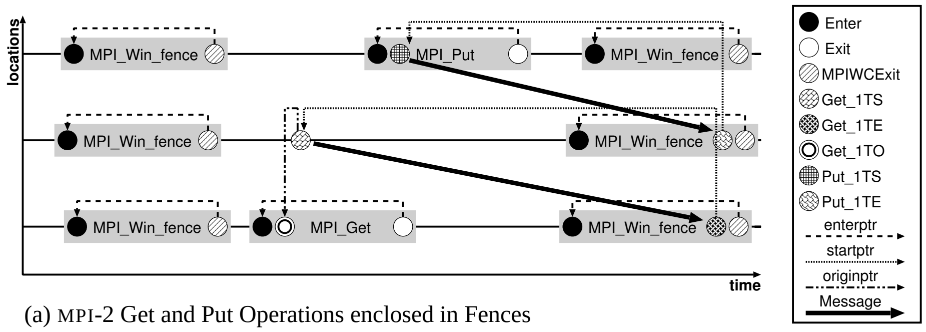
(b) Get and Put Operations with Locks

Fig. 1. Examples for Basic Event Model

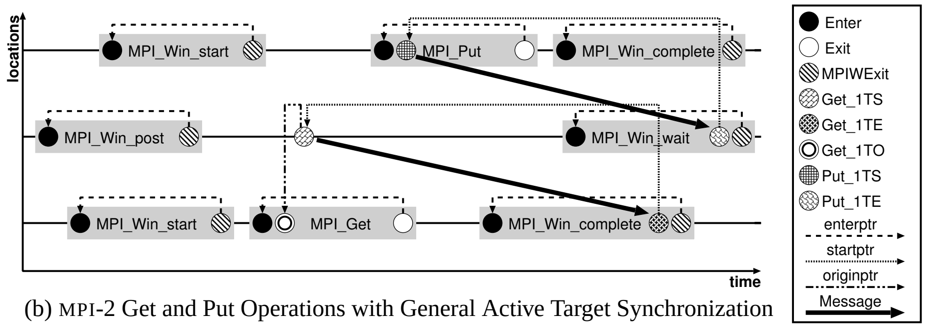
havior. Therefore, in case of MPI-2, we use an extended model, which is described in the next subsection.

3.2 Extended Model

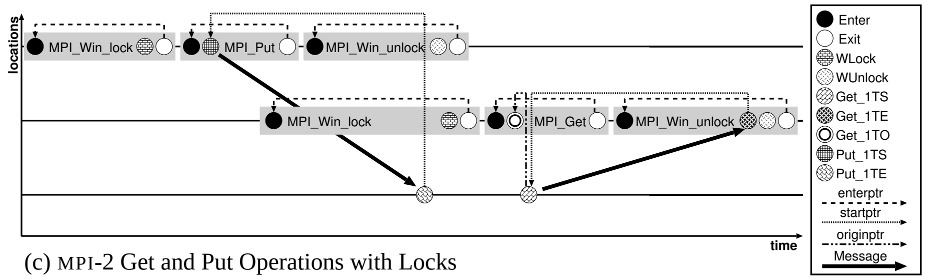
The *extended model* observes the MPI-2 synchronization semantics and, therefore, better reflects the user-visible behavior of MPI-2 RMA operations. Figure 2 shows the model for the three different synchronization methods defined by MPI-2. The end of fences and GATS calls is now modeled with MPIWEXIT or MPIWCEXIT respectively in order to capture their collective nature. The transfer-start event is still located at the source process immediately after the begin of the corresponding communication function (as it is in the basic model). However, the transfer-end event is now 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. In order to rectify this situation, we introduced a new event GET_1TO, which marks the origin's location and time, as well as a new relationship *originptr* associating this new event with the start of the transfer (GET_1TS). This allows us in the analysis phase to locate all events related to RMA transfers. 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 post-processing of the collected event trace becomes necessary.



(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

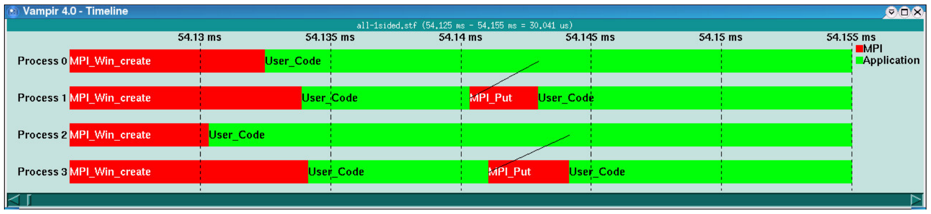
Fig. 2. Examples for Extended Event Model

4 Analysis and Visualization

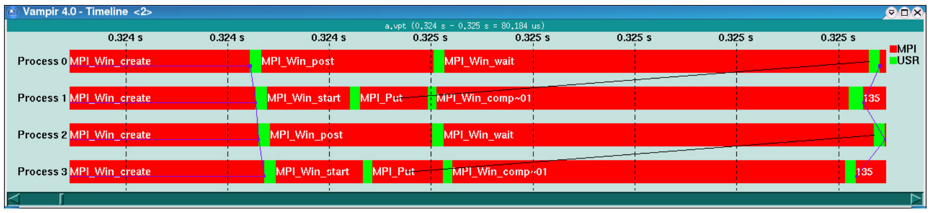
In this section, we outline the changes to KOJAK components that were necessary to implement support for the event models introduced in the last section. For a detailed description of the implementation see [13].

To record the new RMA related events, we implemented a set of wrapper functions for all SHMEM and MPI-2 communication and synchronization functions for C/C++ and Fortran. As MPI uses opaque types for representing windows and groups, we also had to add code for tracking these objects to the PMPI wrappers. Since the code for tracking the communication is only executed by the origin process, but the events for marking the start of a remote read (GET_1TS) and for the end of a remote write (PUT_1TE) are associated with the target process in our model, we cannot directly place the events in the correct trace buffer, which resides in the target process, during measurement. We solve this problem by writing temporary REMOTE_PUT_1TE and REMOTE_GET_1TS events to the local trace buffer and later, during the merge phase, which generates a global trace, replace these with the correct events. This is done by manipulating their location and destination/source attributes. For MPI-2 remote read operations, we also generate the additional GET_1TO event. Moreover, we adjust the timestamp of transfer-end events in compliance with the extended event model. To do this, the merge process places #_1TE first into queues (which we keep for each location and window), then uses the recorded attributes of MPI RMA operations to locate the positions in the event stream when RMA transfers are complete, and finally at that point ejects the corresponding queued events into the stream with corrected timestamps. Performing these operations during the merge has also the advantage of lowering the measurement overhead.

Finally, we extended our tool which converts our internal EPILOG event trace format to VTF3 to handle the new RMA event types. This allows us to use the well-known VAMPIR tool [8] to analyze and visualize traces of RMA applications. RMA transfers are mapped to message lines but with special unique MPI tag values which enables us to get VAMPIR to use different visual attributes (color and/or line style) so they can be distinguished from normal point-to-point messages.



(a) Recorded with Intel Trace Collector



(b) Recorded with KOJAK

Fig. 3. Time-line of MPI-2 Put Operation and General Active Target Synchronization

As a result, Figure 3 presents two time-line displays of the same simple example program, which uses `MPI_Put` together with general active target synchronization. The first one shows trace recorded with the Intel Trace Collector and the second one a trace recorded with our new prototype measurement system. The Intel library does not measure the routines of the general active target synchronization, creating the wrong impression that useful user calculations are done instead. Also, the message lines show the RMA transfer as completed by the end of the put operation which does not reflect the user-visible behavior, as specified by the MPI-2 standard.

5 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 implemented an extension to the KOJAK performance analysis toolset to instrument and trace applications based on one-sided communication and synchronization and to analyze the collected traces using the VAMPIR event trace visualizer. The next step will be to extend EXPERT [12], the automatic trace analysis component of KOJAK, to handle one-sided communication. This will include the definition of RMA-related performance properties (i.e., event patterns which represent inefficient behavior of RMA communication and synchronization).

Acknowledgments. We would like to thank Rolf Rabenseifner for helping us better understand MPI-2 one-sided communication and synchronization and for many helpful suggestions to improve our event models.

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*. Accepted for publication in the International Journal of High-Performance Computing Applications.
4. K. Mohror and K.L. Karavanic. *Performance Tool Support for MPI-2 on Linux*. In Proceedings of SC'04, Pittsburgh, PA, November 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, August 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. W. Nagel, A. Arnold, M. Weber, H.-C. Hoppe, and K. Solchenbach. Vampir: Visualization and Analysis of MPI Resources. *Supercomputer*, 12:69–80, January 1996.
9. R. W. Numrich and J. K. Reid. Co-Array Fortran for Parallel Programming. *ACM Fortran Forum*, 17(2), 1998.
10. F. Wolf and B. Mohr. Automatic Performance Analysis of Hybrid MPI/OpenMP Applications. *Journal of Systems Architecture*, 49(10–11):421–439, November 2003.
11. B. Mohr, L. DeRose, and J. Vetter. *A Performance Measurement Infrastructure for Co-Array Fortran*. In *Proceddings of of Euro-Par 2005*, Springer, Lisboa, Portugal, September 2005.
12. F. Wolf, B. Mohr, J. Dongarra, and S. Moore. Efficient Pattern Search in Large Traces through Successive Refinement. In *Proceddings of Euro-Par 2004*, Springer, LNCS 3149, pp. 47–54, Pisa, Italy, September 2004.
13. M. -A. Hermanns. *Event-based Performance Analysis of Remote Memory Access Operations* (In German). Diploma Thesis, Forschungszentrum Jülich, 2004.