Scalable Detection of MPI-2 Remote Memory Access Inefficiency Patterns

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

[1] Jülich Supercomputing Centre
Forschungszentrum Jülich, Germany
{m.a.hermanns,m.geimer,b.mohr,f.wolf}@fz-juelich.de
[2] Department of Computer Science
RWTH Aachen University, Germany

Abstract. Wait states in parallel applications can be identified by scanning event traces for characteristic patterns. In our earlier work, we have defined such patterns for MPI-2 one-sided communication, although still based on a trace-analysis scheme with limited scalability. Taking advantage of a new scalable trace-analysis approach based on a parallel replay, which was originally developed for MPI-1 point-to-point and collective communication, we show how wait states in one-sided communications can be detected in a more scalable fashion. We demonstrate the scalability of our method and its usefulness for the optimization cycle with applications running on up to 8,192 cores.

Keywords: MPI-2, remote memory access, performance analysis, scalability, pattern search.

1 Introduction

Remote memory access (RMA) describes the ability of a process to access all or parts of the memory belonging to 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 a single process, it is also called one-sided communication. This programming model is made available to the programmer often 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 with version 2 [1], and since then has been adopted by the majority of the available MPI implementations.

Although it has been shown that the use of MPI-2 RMA can improve application performance [2], it has not yet been widely adopted among the MPI user community. But we believe that the availability of suitable programming tools, in particular for performance analysis, can encourage more developers to exploit the benefits of this model. However, since increasing demand for compute power in combination with recent trends in microprocessor design towards multicore chips forces applications to scale to much higher processor counts, such tools must be scalable to be useful.

A non-negligible fraction of the execution time of MPI applications can often be attributed to wait states, which occur when processes fail to reach synchronization points in a timely manner, for example, due to load imbalance. Especially when trying to scale communication-intensive applications to large processor counts, such wait states can

M. Ropo et al. (Eds.): EuroPVM/MPI 2009, LNCS 5759, pp. 31–41, 2009.
© Springer-Verlag Berlin Heidelberg 2009

present severe challenges to achieving good performance. In our earlier work [3], we have shown how wait states related to MPI-2 RMA can be identified by searching event traces for characteristic patterns. However, the search algorithm applied was sequential and intended to operate on a single global trace file, offering only limited scalability. In the meantime, we developed a general framework to make pattern search in event traces more scalable [4]. Instead of sequentially analyzing a single global trace file, the framework analyzes multiple process-local trace files in parallel while performing a parallel replay of the target application's communication behavior. In this paper, we present a synthesis of the two approaches, making the search for wait states in the context of MPI-2 RMA more scalable by enacting a parallel replay of one-sided operations, which had previously only been tried for two-sided and collective operations. The new scalable detection scheme for one-sided communication has been integrated into Scalasca [5], a performance analysis toolset specifically designed for large-scale systems.

The remainder of this paper is organized as follows. Section 2 gives a brief overview of the work done on this topic so far. Afterwards, the semantics of the MPI RMA programming model are explained in Section 3, before specifying the supported MPI RMA inefficiency patterns and their replay-based detection algorithms in Section 4. Moreover, results with two RMA-based applications running on up to 8,192 cores demonstrate the scalability of our method and its usefulness for the optimization cycle in Section 5. Finally, Section 6 concludes the paper and gives a brief outlook on future work.

2 Related Work

The number of portable performance-analysis tools supporting MPI-2 RMA is quite limited. The Paradyn tool, which conducts an automatic on-line bottleneck search, supports several major features of MPI-2 [6]. To analyze RMA operations, it collects process-local statistical data (i.e., transfer counts and time spent in RMA functions). Yet, it does not take inter-process relationships into account. By contrast, the TAU performance system [7] supports profiling and tracing of MPI-2 one-sided communication, though only by monitoring the entry and exit of RMA functions. Therefore, it does not provide RMA transfer statistics nor does it record the transfers in tracing mode. Recently, the trace collection and visualization toolset VampirTrace/Vampir [8] was extended to provide experimental support for MPI-2 one-sided communication [9].

In our previous work [3], we defined a formal event model as well as a number of characteristic patterns of inefficient behavior that can arise in the context of MPI-2 RMA communication. The detection of these patterns was implemented as an extension of the serial trace analyzer KOJAK [10] and constitutes the foundation for our new, scalable bottleneck detection algorithm.

The *Parallel Performance Wizard* (PPW) [11] is an automatic performance tool specifically designed for *partitioned global address space* (PGAS) languages, which provide the abstraction of shared memory to the user while internally converting all remote accesses to one-sided communication calls. Some PGAS languages, such as UPC, also support explicit one-sided communication. PPW supports the analysis of programs written in such languages by providing so-called generic operation types that are defined on top of an RMA event model.

3 MPI-2 Remote Memory Access

The interface for RMA operations defined by MPI differs from 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 RMA API 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. Instead, they can access only designated parts of the memory, which are called *windows*. Such windows must be explicitly initialized with a call to `MPI_Win_create` and released with a call to `MPI_Win_free` by all processes that either want to expose or to access this memory. These calls are collective between all participating partners and may include an internal barrier operation. By *origin* MPI denotes the process that performs an RMA read or write operation, and by *target* the process the memory of which is accessed.

There are three RMA communication calls in MPI: `MPI_Get` to read from and `MPI_Put` and `MPI_Accumulate`[1] to write to the target window. MPI-2 RMA synchronization falls in two categories: *active target* and *passive target* synchronization. In active mode both processes, origin and target, have to participate in the synchronization, whereas in passive mode explicit synchronization occurs only on the origin process. MPI provides three RMA synchronization mechanisms:

Fences: The function `MPI_Win_fence` is used for active target synchronization and is collective over the communicator used when creating the window. RMA operations need to occur between two fence calls.

General Active Target Synchronization (GATS): In this scheme, synchronization occurs between a group of processes that is explicitely supplied as a parameter to the synchronization calls. A so-called *access epoch* is started at an origin process by `MPI_Win_start` and terminated by a call to `MPI_Win_complete`. The start call specifies the group of targets for that epoch. Similarly, an *exposure epoch* is started at a target process by `MPI_Win_post` and completed by `MPI_Win_wait` or `MPI_Win_test`. Again, the post call specifies the group of origin processes for that epoch.

Locks: Finally, shared and exclusive locks are provided for the so-called passive target synchronization through the `MPI_Win_lock` and `MPI_Win_unlock` calls, defining the access epoch for this window at the origin.

It is implementation-defined whether some of the above-mentioned calls are blocking or non-blocking, for example, in contrast to other shared memory programming paradigms, the lock call may not be blocking. In the remainder of this paper, we exclusively focus on active target communication. However, as part of our future work, we plan to address also passive target communication.

[1] A generalized version of `MPI_Put` with the possibility of using a reduction operator.

4 Automatic Detection of RMA Inefficiency Patterns

In this section, we describe how the MPI RMA-related inefficiency patterns defined in [3] can be automatically detected in a scalable way within the framework of the Scalasca performance-analysis toolset. Scalasca is an open-source toolset that can be used to analyze the performance behavior of parallel applications and to identify opportunities for optimization. As a distinctive feature, Scalasca provides the ability to identify wait states in a program that occur, for example, as a result of unevenly distributed workloads, by searching event traces for characteristic patterns. To make the trace analysis scalable, process-local traces are analyzed in parallel without prior merging. The central idea behind Scalasca's parallel trace analyzer is to reenact the application's communication behavior recorded in the trace, analyzing communication operations using operations of the same type. For example, to detect wait-states related to point-to-point message transfers, the events necessary to analyze such a communication are exchanged between the participating processes in point-to-point mode as well. This techniques relies on reasonably synchronized timestamps between the different processes. On platforms without synchronized clocks, a software correction mechanism is applied post mortem [12]. The scalability of the parallel replay mechanism has already been demonstrated for up to 65,536 cores [13].

Here, we apply the same methodology to MPI RMA operations, that is, RMA transfers are used to exchange the data required for the analysis. For this purpose, our analysis creates a small buffer window for every window created by the application itself. The buffers are used by origin and target processes to exchange the timestamps needed for the calculation of waiting times. Earlier, during trace acquisition (i.e., at application runtime), Scalasca's measurement layer keeps track of all windows being created and records the window definitions plus all synchronization and communication operations acting on these windows. When the replay is performed in the analysis step, all those windows can be recreated using the same set of processes based on the recorded window definitions. The timestamps are subsequently transferred using `MPI_Get` and `MPI_Accumulate` operations.

To ensure that the access and exposure epochs are available at the time when the analyzer processes the corresponding part of the event trace, the synchronization pattern used by the original application is reconstructed during the replay. That is, synchronization on the exchange window is triggered by the exit events recorded for the RMA synchronization calls involved. The exit event for `MPI_Win_fence` collectively synchronizes the exchange window, whereas the exit of `MPI_Win_start` opens an access epoch for the recorded group of processes, which is closed whenever the exit event of the corresponding `MPI_Win_complete` call is found. Similarly, an exposure epoch is opened and closed at the exit events of `MPI_Win_post`, `MPI_Win_wait`, and `MPI_Win_test` regions completing an exposure epoch. Please note that the analysis relies on correctly applied synchronization, which is why it may deadlock in cases of erroneous synchronization by the application.

During the replay, specific call backs are triggered for RMA-related events to detect the different inefficiency patterns, as described below. For the sake of simplicity, the individual actions taken are described in the context of the respective pattern. However, our implementation actually combines all these actions using a sophisticated

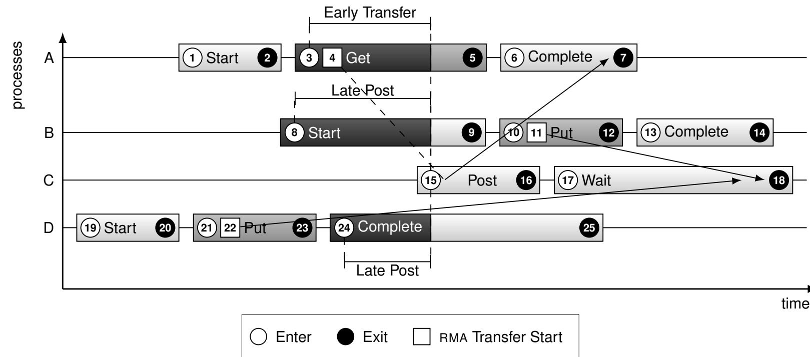

Fig. 1. The Early Transfer and Late Post (in two variants) inefficiency patterns. The waiting time attributed to each pattern is marked in dark gray. Origin and target roles are isolated in different processes—process C is the target for processes A, B, and D.

notification and call-back mechanism not to transfer the same data twice, thereby minimizing the communication costs of the analysis.

Late Post. The Late Post inefficiency pattern refers to waiting time occurring during general active target synchronization (GATS) operations of an access epoch that block until access is granted by the corresponding exposing process as depicted in Figure 1. Depending on the MPI implementation, this may happen either during `MPI_Win_start` (proc. B and C) or `MPI_Win_complete` (proc. D and C). However, the exact blocking semantics are usually not known. Therefore, we use a heuristic to determine which calls are blocking. If and only if the enter event of the call to the latest `MPI_Win_post` on the exposing processes (15) occurs within the time interval of the `MPI_Win_start` call on the accessing process (8,9), we assume that the call to `MPI_Win_start` is blocking, and the waiting time is determined by the time difference between entering the `MPI_Win_post` operation (15) and entering `MPI_Win_start` (8), to which the waiting time is finally ascribed. Likewise, waiting time during the call to `MPI_Win_complete` is determined on the accessing process, where the enter event of the complete call (24) is used to calculate the waiting time. In the case one of these calls is falsely assumed to be blocking, the overall time spent in the call will be very small, resulting in a negligible inaccuracy with respect to the overall severity of this pattern.

To detect the Late Post pattern, the following MPI RMA operations occur during the replay: The exit event of the `MPI_Win_post` call (16) triggers the start of the exposure epoch on the target process after initializing the exchange buffer with the timestamp of the post enter event (15) and default values for all other fields. At the origin processes, the exit events of the call to `MPI_Win_start` (2,9,20) trigger the start of the access epochs for the exchange window and the post enter timestamp of each target process is retrieved using `MPI_Get`. Accordingly, the exit events of the calls to `MPI_Win_complete` (7,14,25) close the access epoch and the post enter timestamps can be accessed to locally determine the latest post. This timestamp can then be compared to the timestamps of the locally available events to determine the Late Post variant and finally calculate the waiting time if applicable. On the target processes, the end of the

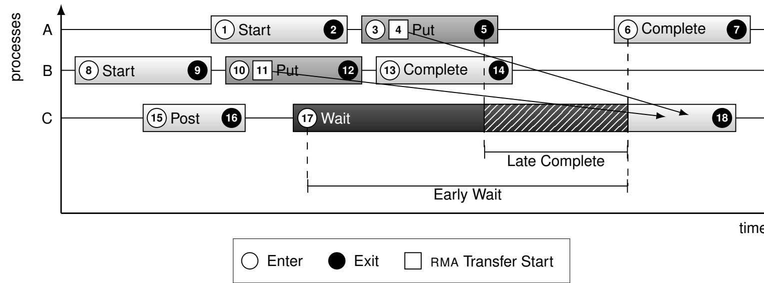

Fig. 2. The Early Wait (dark gray+hatched) and Late Complete (hatched) inefficiency patterns

exposure epoch is ensured by calling `MPI_Win_wait` when reaching the corresponding exit event (18).

Early Transfer. The Early Transfer pattern occurs when an RMA operation is blocking because the relevant exposure epoch has not yet been started (Fig. 1, proc. A and C). It is therefore similar to Late Post, and in fact requires exactly the same data to be transferred (i.e., the post enter timestamps), but the waiting time is attributed to the remote access operation. As before, it can not easily be determined whether the RMA transfer call was actually blocking. However, we assume this to be the case if the corresponding `MPI_Win_post` (15) call was issued on the target side within the time interval of the remote access in question (3,5). Since the post enter timestamps are only accessible after closing the access epoch, a backward traversal of the event data is required, comparing the timestamps recorded for each RMA operation with the post enter timestamp of the corresponding target process. If the RMA operation was non-blocking in reality, the time falsely classified as waiting time would again be very small.

Early Wait. This pattern refers to the situation where the exposing process is waiting for other processes to complete the remote accesses of their access epoch (Fig. 2). As the call to `MPI_Win_wait` cannot return until all access epochs have been finished, the time span between the enter event of the call to `MPI_Win_wait` and the latest enter event of the corresponding calls to `MPI_Win_complete` on the accessing processes is counted as waiting time.

To detect the Early Wait pattern, the timestamps of the enter events of calls to `MPI_Win_complete` (6,13) are transferred to the target processes via `MPI_Accumulate` using the `MPI_MAX` operator just before closing the access epoch, thereby storing the latest complete enter timestamp in the target's exchange buffer. The waiting time can then be determined by subtracting the timestamp of the wait enter event (17) from the latest complete enter timestamp (6) stored in the exchange buffer. As can be seen, the one-sided model naturally lends itself to perform this type of analysis.

Late Complete. To allow for efficient synchronization, access epochs should be as compact as possible. As the target process can close the exposure epoch only after all access epochs have been completed, waiting time in the Early Wait pattern that occurs between the last RMA operation and the completion of the respective access epoch is attributed to the Late Complete pattern (Fig. 2, hatched area), a sub-pattern of Early Wait.

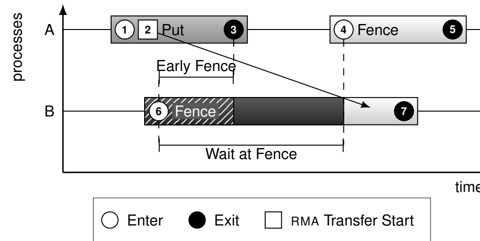

Fig. 3. The Wait-at-Fence (dark gray+hatched) and Early Fence (hatched) inefficiency patterns

During the detection, each origin caches the exit event of the latest RMA operation (5,12) separately for each target. If no RMA operation is present in the access epoch, the exit timestamp of the `MPI_Win_start` call is taken. Then, all the origins of a given target transfer their cached timestamp to the target via `MPI_Accumulate` using the `MPI_MAX` operator just before closing the access epoch while processing the exit events of the calls to `MPI_Win_complete` (7,14). There the maximum value obtained can then be subtracted from the timestamp of the latest complete enter event, which is already available from the Early Wait detection algorithm.

Wait at Fence. This pattern refers to a wait state during the completion of a fence operation as shown in Figure 3. Although `MPI_Win_fence` is a collective call, it may not be synchronizing, depending on given assertions or MPI-internal window status information. However, as potentially all processes of the communicator may access the local window, a confirmation is needed from the remaining processes that their access epoch on this window has ended. Thus, at least a partial synchronization is required. We assume a collective call of `MPI_Win_fence` to be globally synchronizing if the timestamps of all associated enter events occur before any exit event of the same fence call. Unfortunately, this heuristic does not detect and attribute time to the Wait at Fence pattern if only some of the processes synchronize. However, waiting time due to partial synchronization is detected, if the sub-pattern Early Fence is present (see below).

To detect the Wait at Fence pattern, the latest enter and earliest exit timestamps of the fence (4,7) are determined with a single `MPI_Allreduce` call using a user-defined operator. If the above-mentioned overlap criterion is met, the difference between the latest enter event (4) and the local enter event (6) is counted as waiting time.

Early Fence. Waiting time for entering a fence before all remote accesses have finished is attributed to the Early Fence pattern, a sub-pattern of Wait at Fence (Fig. 3, hatched area). It will always be accounted as waiting time, even in situations where only a subset of the processes are synchronizing.

Here, all processes locally determine the latest exit timestamp of their remote accesses (3) for each target and transfer them to the matching target processes via accumulate, again using the `MPI_MAX` operator. These transfers are surrounded by two calls to fence to ensure correct synchronization. In this way the earliest possible completion of the latest RMA operation of all accessing origin processes is determined and used to calculate the waiting time of this pattern as the time difference between leaving the

Table 1. Event statistics and analysis times for the red-black SOR Poisson solver. The last column shows the analysis time in percent of the application runtime.

# cores	# events	trace size [MB]	analysis time [s]	analysis time [%]
128	12,681,376	106.78	2.19	0.75
256	26,130,816	217.13	2.22	0.79
512	53,029,696	437.63	2.78	0.77
1,024	107,595,520	883.25	2.37	0.84
2,048	216,727,168	1,774.63	2.54	0.86
4,096	436,536,592	3,565.63	2.91	1.01
8,192	876,125,440	7,151.25	3.60	1.19

latest RMA operation (3) and the local enter event for the fence (6). Time attributed to the Early Fence pattern is also attributed as time in Wait at Fence, even if due to our heuristic the analyzer was unable to detect the Wait at Fence pattern directly.

5 Results

In this section, we present early results for two different MPI-2 RMA codes. We took our measurements at the Jülich Supercomputing Centre on the IBM Power6 575 cluster "Jump" and an IBM Blue Gene/P system at IBM Rochester. Based on the results collected with up to 8,192 processes on the two-rack Blue Gene/P so far, and the experiences with our replay-based analysis method in general, we believe that the RMA analysis scales well beyond this point.

5.1 SOR Solver

With the first code, SOR, we verify the scalability of our analysis. SOR solves the Poisson equation using a red-black successive over-relaxation method. The two main communication steps are halo-exchange and scalar reduction operations. The former was adapted to use MPI RMA instead of the original non-blocking point-to-point communication. The latter still uses MPI collective communication as before. The global domain is a three-dimensional grid of size $N_{horiz} \times N_{horiz} \times N_{vert}$, which is partitioned along the two horizontal dimensions using a 2D process mesh. The communication pattern of this application is typical for grid-point codes used in earth and environmental science.

A series of experiments was collected and analyzed at different scales on a two-rack IBM Blue Gene/P system at IBM Rochester. The solver was configured to run for approx. 5 minutes using weak scaling and a problem size where no convergence was reached within the maximum number of 1000 iterations. The key numbers are given in Table 1. As can be seen, the total number of events increases linearly with the number of cores, as is the case for the total size of the compressed trace data. The time exclusively needed for the replay analysis (i.e., without loading the traces and writing the results, which together took less than 16 seconds for the 8,192-core run) is only mildly ascending, as expected for weak scaling.

5.2 BT-RMA

To evaluate the usefulness of our analysis for application optimization and to verify that the inefficiency patterns described earlier appear in practice, we incrementally developed a version of the BT benchmark from the NAS Parallel Benchmark Suite 2.4 [14] that uses one-sided instead of non-blocking point-to-point communication named BT-RMA. The BT benchmark solves three sets of uncoupled systems of equations in the three dimensions x, y, and z. The systems are block tridiagonal with 5×5 blocks. The domains are decomposed in each direction, with data exchange in each dimension during the solver part, as well as a so-called face exchange after each iteration. Those exchanges are implemented using non-blocking point-to-point communication in BT. Due to a known problem with fence synchronization on Blue Gene/P systems with V1R3M0 runtime environments, we measured, analyzed and subsequently optimized the BT-RMA code on our IBM Power6 575 cluster "Jump" using the "class D" problem size on 256 cores in ST mode. For measurement, five purely computational subroutines were excluded from instrumentation, lowering the runtime intrusion to about 1% and keeping the trace size manageable.

For simplicity, our initial version of BT-RMA used fence synchronization for both data exchanges. The analysis results (Tab. 2) showed that more than 44% of the overall runtime was spent in active target synchronization calls, that is, `MPI_Win_fence`. Approximately 6% of the total time was found to be waiting time attributable to the Wait at Fence pattern. Further investigation revealed that most of this time was spent in synchronizing the solver exchanges.

We subsequently modified the code to use GATS synchronization in the solver, while still using fences in the face exchange. This version shows a dramatic reduction of the overall execution time to only 57% of the runtime of the fence-only variant. Although significantly faster, active target synchronization still accounts for about 4.2% of the application runtime, with Wait at Fence requiring 1.3% and Early Wait about 0.9%. In addition, this variant uses 2.5× more time for remote access operations compared to the fence-only version, now spending 1.6% of the total time in the Early Transfer wait state.

Table 2. Performance metrics for different variants of BT-RMA in CPU seconds (first number) and percent of total CPU seconds (second number). All values are inclusive, that is, they include the time for sub-patterns (indicated through indentation).

Metric	fence only		GATS/fence		GATS only		GATS only (opt)	
Total time	109,361.7	100.0	61,888.9	100.0	61,248.7	100.0	60,504.0	100.0
MPI time	51,252.5	46.9	7,156.5	11.6	6,882.5	11.3	6,284.3	10.4
RMA sync.	48,703.8	44.5	2,585.9	4.2	2,177.9	3.6	3,476.0	5.8
Wait at Fence	6,080.0	5.7	805.9	1.3	0.0	0.0	0.0	0.0
Early Wait	0.0	0.0	568.5	0.9	950.1	1.6	1,923.8	3.2
Late Complete	0.0	0.0	1.2	0.0	289.6	0.5	2.0	0.0
Late Post	0.0	0.0	2.9	0.0	4.4	0.0	0.9	0.0
RMA comm.	1,324.9	1.2	3,246.6	5.3	3,507.4	5.7	1,603.2	2.7
Early Transfer	0.0	0.0	980.5	1.6	2,299.6	3.8	848.6	1.4

As a next step, we completely eliminated the calls to MPI_Win_fence by adapting the face exchange to also use GATS synchronization with individual windows for each of the six neighbors. Although the Wait at Fence wait state disappeared, the waiting time almost entirely migrated to the Late Complete (mostly in the face exchange) and Early Transfer patterns (predominantly in the solver), thus only providing an additional speedup of approximately one percent.

Based on these analysis results, we finally rearranged the GATS synchronization calls slightly, starting the exposure epochs as early as possible and shortening the access epochs by moving the start/complete calls close to the RMA transfers, decreasing the overall runtime again. BT-RMA is now almost 45% faster than the first fence-based version and marginally faster than the original BT code.

6 Conclusion

MPI-2 remote memory access is a portable interface for one-sided communication on current large-scale HPC systems. To better support developers in using this interface, we have presented a scalable method for identifying wait states in event traces of RMA applications. A particular challenge to overcome was the availability of the communication parameters on only one side of an interaction between two processes, requiring one-sided transfers of analysis data during the parallel replay. We have shown the scalability of our method using one application kernel with up to 8,192 cores and incrementally optimized a second and more complex code guided by results of our analysis.

Future research will evaluate the scalability on even larger process configurations. In addition, we plan to investigate further inefficiency patterns for MPI-2 RMA such as passive target lock competition. We also consider leveraging our method for the scalable automatic analysis of applications written in PGAS languages such as UPC.

Acknowledgments

This work has been supported by the German Ministry for Education and Research (BMBF) under Grant No. 01IS07005C ("ParMA") and the Helmholtz Association of German Research Centers under Grant No. VH-NG-118. The authors also would like to thank the Rechenzentrum Garching (RZG) of the Max Planck Society and the IPP for the opportunity to run the BT-RMA benchmark on their IBM Power6 system "VIP", as well as the IBM Rochester Blue Gene Benchmark Center for providing access to their two-rack Blue Gene/P system.

References

1. Message Passing Interface Forum: MPI: A Message-Passing Interface Standard, Version 2.1 (June 2008), http://www.mpi-forum.org/
2. Mirin, A.A., Sawyer, W.B.: A scalable implementation of a finite-volume dynamical core in the community atmosphere model. International Journal on High Performance Computing Applications 19(3), 203–212 (2005)

3. Kühnal, A., Hermanns, M.-A., Mohr, B., Wolf, F.: Specification of inefficiency patterns for MPI-2 one-sided communication. In: Nagel, W.E., Walter, W.V., Lehner, W. (eds.) Euro-Par 2006. LNCS, vol. 4128, pp. 47–62. Springer, Heidelberg (2006)
4. Geimer, M., Wolf, F., Wylie, B.J.N., Mohr, B.: Scalable parallel trace-based performance analysis. In: Mohr, B., Träff, J.L., Worringen, J., Dongarra, J. (eds.) PVM/MPI 2006. LNCS, vol. 4192, pp. 303–312. Springer, Heidelberg (2006)
5. Scalasca, `http://www.scalasca.org/`
6. Mohror, K., Karavanic, K.L.: Performance tool support for MPI-2 on Linux. In: Proceedings of the Supercomputing Conference (SC), Pittsburgh, PA (2004)
7. Shende, S.S., Malony, A.D.: The TAU parallel performance system. International Journal of High Performance Computing Applications 20(2), 287–331 (2006)
8. Knüpfer, A., Brunst, H., Doleschal, J., Jurenz, M., Lieber, M., Mickler, H., Müller, M.S., Nagel, W.E.: The Vampir performance analysis tool set. In: Resch, M., Keller, R., Himmler, V., Krammer, B., Schulz, A. (eds.) Tools for High Performance Computing, pp. 139–155. Springer, Heidelberg (2008)
9. Knüpfer, A.: Personal communication (2009)
10. Wolf, F., Mohr, B.: Automatic performance analysis of hybrid MPI/OpenMP applications. Journal of Systems Architecture 49(10-11), 421–439 (2003)
11. Leko, A., Su, H.H., Bonachea, D., Golden, B., Billingsley, M., George, A.: Parallel Performance Wizard: A performance analysis tool for partitioned global-address-space programming models. In: Proc. of the Supercomputing Conference (SC), vol. 186. ACM, New York (2006)
12. Becker, D., Rabenseifner, R., Wolf, F., Linford, J.: Replay-based synchronization of timestamps in event traces of massively parallel applications. Scalable Computing: Practice and Experience 10(1), 49–60 (2009); Special Issue International Workshop on Simulation and Modelling in Emergent Computational Systems (SMECS)
13. Geimer, M., Wolf, F., Wylie, B.J., Mohr, B.: A scalable tool architecture for diagnosing wait states in massively parallel applications. Parallel Computin (in press) (2009)
14. Bailey, D.H., Barzcz, E., Dagum, L., Simon, H.D.: NAS parallel benchmark results. IEEE Parallel Distrib. Technol. 1(1), 43–51 (1993)