Scalable Detection of MPI-2 Remote Memory Access Inefficiency Patterns

Marc-André Hermanns[2], Markus Geimer[1], Bernd Mohr[1], and Felix Wolf[1,2,3]

[1] Forschungszentrum Jülich
Jülich Supercomputing Centre, Germany
{m.geimer,b.mohr}@fz-juelich.de

[2] German Research School for Simulation Sciences
Laboratory for Parallel Programming, Germany
{m.a.hermanns,f.wolf}@grs-sim.de

[3] RWTH Aachen University,
Department of Computer Science, 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 inefficiency patterns for MPI-2 one-sided communication, although still based on a serial trace-analysis scheme with limited scalability. In this article, we show how wait states in one-sided communications can be detected in a more scalable fashion by 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. Moreover, we demonstrate the scalability of our method and its usefulness for the optimization cycle with applications running on up to 32,768 cores.

Key words: 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 [9], 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 [10], it has not yet been widely adopted among the MPI user community. On the other hand, 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 as well in order 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 present severe challenges to achieving good performance. In our earlier work [8], we have shown how wait states related to MPI-2 one-sided communication 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 the pattern search in event traces more scalable [3]. Instead of sequentially analyzing a single global trace file, the framework analyzes multiple process-local trace files in parallel while performing a replay of the target application's communication behavior. In this article, 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 [12], a performance analysis toolset specifically designed for large-scale systems.

The remainder of this article 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 one-sided 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 32,768 cores demonstrate the scalability of our method and its usefulness for the optimization cycle in Section 5. Finally, Section 6 concludes this article 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 [11]. 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 [13] supports profiling and tracing of MPI-2 one-sided communication, though only by monitoring the entry and exit of RMA functions. Therefore, it neither provides RMA transfer statistics nor does it record the transfers in tracing mode. Recently, the trace collection and visualization toolset VampirTrace/Vampir [7] was extended to provide experimental support for MPI-2 one-sided communication [6].

In our previous work, we defined a formal event model [5] as well as a number of characteristic patterns of inefficient behavior that can arise in the context of MPI-2 RMA communication [8]. The detection of these patterns was implemented as an extension of

the serial trace analyzer KOJAK [15] and constitutes the foundation for our new, scalable bottleneck detection algorithms.

One-sided communications are also closely related to *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. In this context, the *Parallel Performance Wizard* (PPW) [14] is an automatic performance tool specifically designed for PGAS languages. PPW supports the performance 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 particular 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` – a variant of `MPI_Put` with the possibility of using a reduction operator – 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 explicitly supplied as a parameter to the synchronization calls. A so-called *access epoch* is started on 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 on 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, which enclose the access epoch for this window on 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 article, we exclusively focus on active target synchronization. However, as part of our future work, we plan to address also passive target synchronization.

4 Automatic Detection of RMA Inefficiency Patterns

In this section, we describe how the MPI RMA-related inefficiency patterns defined in [8] as well as three new patterns, two of them time-based and one of them counter based, 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 by searching event traces for characteristic patterns. Such wait states occur, for example, as a result of unevenly distributed workloads. To make the trace analysis scalable, process-local traces are analyzed in parallel without prior merging. This implies that there is no knowledge about when in time a specific remote event occurred locally available. This information is transferred to the location where it is needed during the analysis process. The central idea behind Scalasca's parallel trace analyzer is to reenact the application's communication and synchronization behavior recorded in the trace, analyzing communication operations using operations of similar 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 technique relies on reasonably synchronized timestamps between the different processes. On platforms without synchronized clocks, a software correction mechanism is applied post mortem [2]. The scalability of the parallel replay mechanism has already been demonstrated for up to 294,912 cores [4].

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, for each window tracked during measurement of the original application, our analysis creates a window exposing a small memory buffer during replay. The buffers are used by origin and target processes to exchange data relevant to the specific performance metrics. Specifically, these buffers comprise four double-precision floating-point entries for timestamps, as well as a bitfield large enough to accommodate a bit for every process having this window defined. The *i*th bit in this bitfield being set indicates at least one RMA access (put or get) by the *i*th process in the corresponding communicator during the ongoing epoch. 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

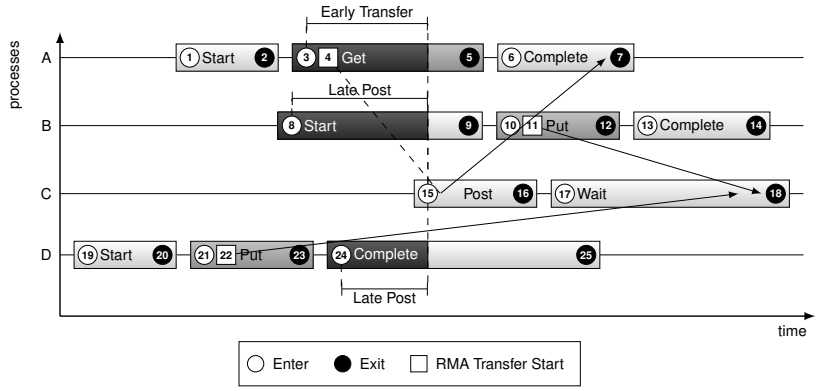

Fig. 1: The Early Transfer and Late Post (in two variants) inefficiency patterns. The waiting time attributed to each pattern is marked in dark gray. Origin and target roles are isolated in different processes—process C is the target for processes A, B, and D.

these windows. When the replay is performed during the analysis step, all those windows can be recreated using the same set of processes based on the recorded window definitions. The information needed for the analysis is 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 with the exit event of `MPI_Win_post` and closed with the exit events of `MPI_Win_wait` or `MPI_Win_test`. 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, not to transfer the same data twice, our implementation actually combines all these actions using a sophisticated notification and call-back mechanism, 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` (variant involving process B and C) or `MPI_Win_complete` (variant involving process

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 by 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. On 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 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 blocks 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 and therefore appears in the communication subtree of the time metrics in the analysis report (Fig. 2). As before, it can not easily be determined whether the original 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 local 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 the original run of the application, 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. 3). 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

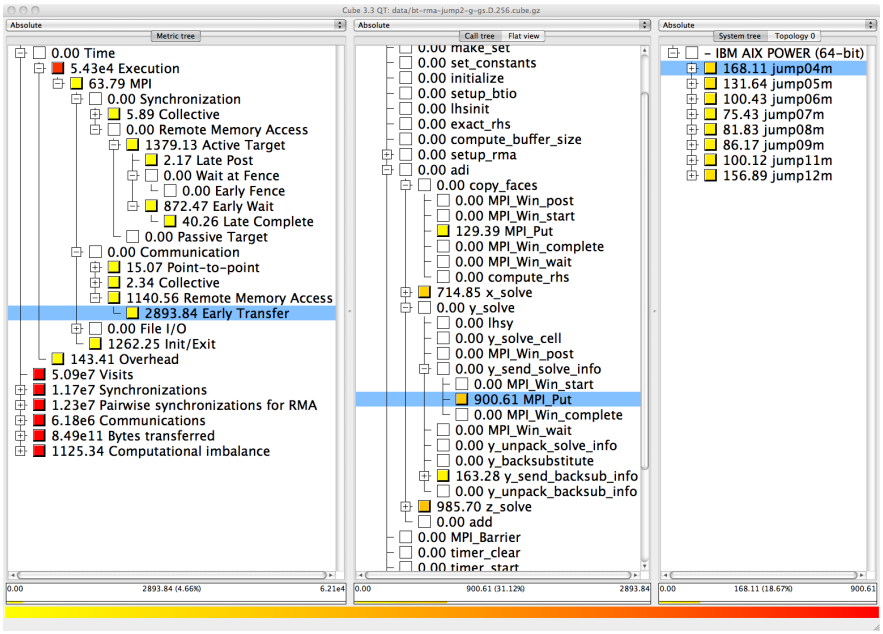

Fig. 2: Screenshot of the CUBE analysis result browser. The Early Transfer inefficiency pattern is selected. This pattern is similar to the Late Post pattern. However, it appears in the communication subtree of the metric tree, as it indicates waiting time of RMA operations, here `MPI_Put`.

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 enter timestamp of a corresponding complete call 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. Depending on whether an MPI implementation can achieve communication/computation overlap or not, access epochs should be as compact as possible in the latter case. 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. 3, hatched area), a sub-pattern of Early Wait. As this waiting time occurs on the target of the access epoch, one solution to reduce waiting time can be moving the call of `MPI_Win_complete` closer to the last RMA operation. This may, however, prevent communication/computation overlap on the ori-

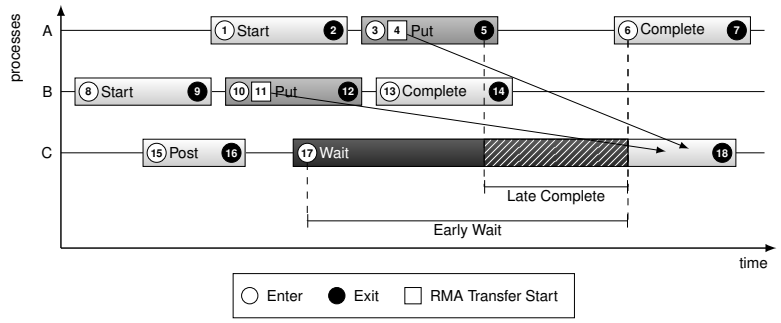

Fig. 3: The Early Wait (dark gray+hatched) and Late Complete (hatched) inefficiency patterns.

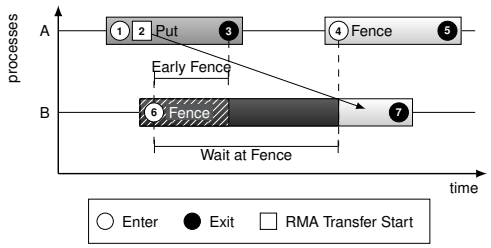

Fig. 4: The Wait-at-Fence (dark gray+hatched) and Early Fence (hatched) inefficiency patterns.

gin. The user must therefore weight the benefits of reducing waiting time on the target against loosing overlap on the origin. Alternatively, the user can also reduce the encapsulating Early Wait pattern by moving the call to `MPI_Win_wait` on the target to a later point in time, which would address both the Early Wait and the Late Complete 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 4. 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 remote processes that their access epoch

on this window has ended. This could be prevented, if the implementation supports it, during some calls where an assertion is given that no put or accumulate calls have to be handled. 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.

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 across all participating processes (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. 4, hatched area). 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 latest RMA operation (3) and the local enter event of the fence (6).

Unneeded Pairwise Synchronizations. In MPI-2 RMA active target synchronization, the user explicitly synchronizes with a set of processes. The results of RMA operations issued before this synchronization become visible only thereafter. Logically, every potential origin for a target process has to inform the target process that no further RMA operation will be issued for the current epoch. In calls to `MPI_Win_fence`, an MPI implementation needs to synchronize each process internally with every other process in the communicator corresponding to the window the call is issued on. There is no possibility for a process to derive this information from local data other than the above-mentioned assertion. This creates an internal synchronization between origin and target, where the target has to wait for an acknowledgement from potential origin processes for the current exposure epoch. In cases where the origin process issues no RMA operation for a target, this synchronization still has to be done and will consume application time. How much time is spent on these synchronizations cannot be explicitly measured, so the costs of unneeded synchronizations can only be estimated by the user interpreting the performance data. The Unneeded Pairwise Synchronizations pattern provides a count for all synchronizations in MPI-2 RMA active target synchronization without preceding RMA operation. It is a subset of all synchronizations done during those synchronization calls. In this way, the user can then investigate this pattern if MPI-2 RMA synchronization in general consumes a major fraction of the application time.

To calculate the number of unneeded synchronizations, the exchange buffer associated with every window contains a bitfield, where the *i*th bit represents a remote access of the process with rank *i* in the communicator associated with the window to the local process. This bitfield is initialized with all bits set to zero before each exposure epoch is started, and then set by the accessing processes using `MPI_Accumulate` with the binary-or operator `MPI_BOR` to set the bit corresponding to its rank on the target process. At the

Table 1: Event statistics and analysis times for the red-black SOR Poisson solver measured on the IBM Blue Gene/P system "Jugene". The last column shows the analysis time in percent of the application runtime.

# cores	# events	execution time [s]	analysis time [s]	analysis time [%]
128	12.682.656	40.7	2.37	5.82
256	26.133.376	42.7	2.41	5.64
512	53.034.816	83.9	2.48	2.96
1,024	107.605.760	86.9	2.61	3.30
2,048	216.747.648	204.0	2.91	1.43
4,096	436.567.552	230.0	3.38	1.47
8,192	876.207.360	375.0	4.47	1.19
16,384	1.758.559.232	397.0	8.61	2.17
32,768	3.523.262.976	793.0	14.73	1.86

end of the exposure epoch, the target process then evaluates the bitfield, counting the number of bits set and storing the difference of the expected origin count and the actual origin count in this epoch as the severity of this pattern. The origin processes accumulating the bit count cache each target location and perform the actual accumulation only once at the end of the access epoch.

5 Results

In this section, we present results for two different MPI-2 RMA codes. We took our measurements on the IBM Power6 575 cluster "Jump" and the IBM Blue Gene/P system "Jugene" located at the Jülich Supercomputing Centre. The results collected with up to 32,768 processes using 8 racks of the 72-rack Blue Gene/P so far confirm that our approach scales well even at very large processor configurations.

5.1 SOR Solver

With the first code, SOR, we verified 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 the 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.

The solver was configured to create measurements with roughly the same number of events per process, and specifically not to converge within the defined maximum number of 1000 iterations. This enabled us to evaluate the weak-scaling behavior of our analysis approach. The key numbers are given in Table 1. As can be seen, the total number of events increases linearly with the number of cores. The jumps in execution

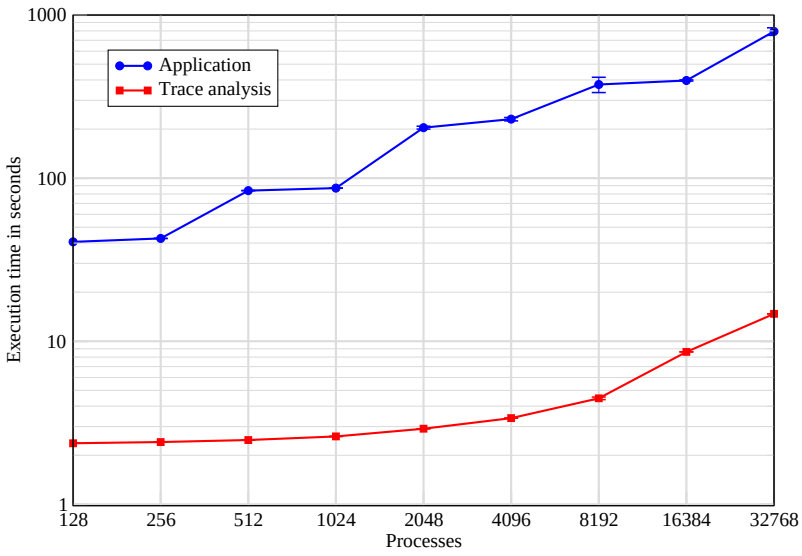

Fig. 5: Scaling behavior of the SOR solver from 128 to 32,768 processes on the IBM Blue Gene/P system "Jugene". The analysis time (red line with squares) stays within one order of magnitude lower than the measured application (blue line with circles).

time of the application reflects different numbers of grid points per process in each dimension. Increasing the workload in the horizontal dimension therefore had a different impact on overall computation than increasing the workload in the vertical dimension. However, as this only influenced the communication/computation ratio of the measurement, and neither the number of per-process events nor the applications communication patterns, these effects are irrelevant for our evaluation.

The time exclusively needed for the replay analysis (i.e., without loading the traces and writing the results, which together took less than 55 seconds for the 32,768-core run) is reasonably low. As seen in Figure 5 it roughly mimics the overall scaling behavior of the application itself, which is to be expected using our replay approach.

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 [1], which we called BT-RMA, that uses one-sided instead of non-blocking point-to-point

Table 2: Performance metrics for different variants of BT-RMA running on 256 cores of the IBM Power6 575 system "Jump". The first number in each column shows time in CPU seconds or a count, respectively. The second number shows the percentage of the total time or total count. 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
P.w. sync. for RMA	$5.98789e^9$	100.0	$7.76264e^7$	100.0	$1.23402e^7$	100.0	$1.23402e^7$	100.0
Unneeded	$5.97555e^9$	99.8	$6.52861e^7$	84.1	0	0.0	0	0.0

communication. 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 the original BT. Initial evaluations were conducted on the 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.

From a user's perspective, the simplest form of synchronization with the MPI one-sided interface is using fences. Thus we developed our initial version of BT-RMA using 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, that is, a major fraction of synchronization time was actually spent synchronizing the individual processes and not in any particular inefficiency pattern.

Further investigation of the initial measurement revealed that most of the synchronization time was spent in synchronizing the solver exchanges. Additionally, the performance metric *Pairwise synchronizations for* RMA showed that 98.1% of all pairwise synchronizations counted are in the same synchronization calls that exhibit the excessive use of time. Even more, 99.8% of those pairwise synchronizations were unneeded as no data is exchanged between the processes involved. We therefore subsequently modified the code to use GATS synchronization in the solver, while still using fences in the face exchange. This version showed a dramatic reduction of the overall execution time to only 57% of the runtime of the fence-only variant. Although significantly faster,

Table 3: Performance metrics for different variants of BT-RMA running on 1,024 cores of the IBM Blue Gene/P system "Jugene". The first number in each column shows time in CPU seconds or a count respectively. The second number shows the percentage of the total time or total count. 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	862,852.5	100.0	235,560.8	100.0	233,595.0	100.0	234,458.7	100.0
MPI time	642,459.6	74.4	21,356.8	9.1	19,726.4	8.4	20,399.2	8.7
Collective sync.	246.8	0.0	1,686.3	0.7	2,088.6	0.9	2,938.6	1.3
Wait at Barrier	238.6	0.0	1,674.8	0.7	2,076.3	0.9	2,926.0	1.3
RMA sync.	639,439.8	74.1	16,901.9	7.2	13,675.9	5.9	14,115.9	6.0
Wait at Fence	16,302.5	1.9	187.8	0.1	0.0	0.0	0.0	0.0
Early Wait	0.0	0.0	2,656.2	1.2	2,661.1	1.1	2,824.4	1.2
Late Compl.	0.0	0.0	43.5	0.0	40.9	0.0	41.9	0.0
Late Post	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
RMA comm.	1920.8	0.2	3,246.6	0.8	1,972.2	0.8	1,981.5	0.9
Early Transfer	0.0	0.0	980.5	0.0	0.0	0.0	0.0	0.0
P.w. sync. for RMA	$1.96872e^{11}$	100.0	$1.15257e^{09}$	100.0	$9.87095e^{07}$	100.0	$9.87095e^{07}$	100.0
Unneeded	$1.96822e^{11}$	99.9	$1.05386e^{09}$	91.5	0	0.0	0	0.0

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 times 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. This indicates that in the version using fence synchronization the MPI implementation is progressing more of the overall RMA communication during the fence calls themselves.

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.

In addition to our initial evaluation of the BT-RMA code on the Power6 575 cluster "Jump", we also investigated its behavior at a slightly larger scale of 1,024 cores on the Blue Gene/P system "Jugene" at the Jülich Supercomputing Centre. Unfortunately, we encountered an issue with general active target synchronization, which is currently under investigation by IBM. The skew in processes moving from one to the other dimension in the solver steps left the runtime system exiting unexpectedly. As a workaround, we inserted a barrier call after each solver step in the dimension x, y, and z when chang-

ing the synchronization mechanism to GATS, yet knowing that this might impair overall performance of the solver. When using fence synchronization in the solver step (fence only) the inserted barrier calls hardly have any effect on the application behavior, as the fence calls implicitly synchronize the processes.

As can be seen in Table 3, the skew in processes between the dimensions while using GATS in the solver part is now consumed by the *Wait at Barrier* pattern, as expected. However, we still observed a dramatic decrease in time spent in active target synchronization. This insight adds to the overall hypothesis that the synchronization itself is not only costly in terms of waiting time, as these costs are attributed to the Wait at Barrier pattern. It also involves, at least in the measurement under consideration, a significant amount of CPU time to execute the synchronization mechanism itself. It can be seen that while the amount of needed pair-wise synchronizations increases only by the expected factor of four going from 256 to 1,024 processes, the amount of unneeded pair-wise synchronizations has increased much more dramatically.

Another interesting aspect of our performance investigation on a second platform is that one can see whether one or the other performance property shows up in the overall application behavior also depends on the MPI implementation. For example, the *Late Post* pattern is non-existent in our measurements of BT-RMA on the Blue Gene/P, which indicates different progress strategies compared to the MPI implementation on the Power6 575 cluster.

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 32,768 cores and incrementally optimized a second and more complex code guided by results of our analysis.

Future research will incorporate additional information into the pattern search, such as the assertions given to various MPI calls by the application. In addition, we plan to investigate further inefficiency patterns for MPI-2 RMA such as passive target lock competition. Moreover, we also consider leveraging our method for the scalable automatic analysis of applications written in PGAS languages such as UPC.

Acknowledgements

This work has been supported by the German Ministry of Education and Research (BMBF) under Grant No. 01IS07005C ("ParMA") and the Helmholtz Association of German Research Centers under Grant No. VH-NG-118.

References

1. Davd H. Bailey, Eric Barzcz, Leonardo Dagum, and Horst D. Simon. NAS parallel benchmark results. *IEEE Parallel Distrib. Technol.*, 1(1):43–51, 1993.
2. Daniel Becker, Rolf Rabenseifner, Felix Wolf, and John Linford. 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)*.
3. Markus Geimer, Felix Wolf, Brian J. N. Wylie, and Bernd Mohr. A scalable tool architecture for diagnosing wait states in massively parallel applications. *Parallel Computing*, 35(7):375–388, July 2009.
4. Markus Geimer, Felix Wolf, Brian J.N. Wylie, Erika Ábrahám, Daniel Becker, and Bernd Mohr. The Scalasca performance toolset architecture. *Concurrency and Computation: Practice and Experience*, 22(6):702–719, April 2010.
5. Marc-André Hermanns, Bernd Mohr, and Felix Wolf. Event-based measurement and analysis of one-sided communication. In *Proc. of the 11th Euro-Par Conference*, volume 3648 of *Lecture Notes in Computer Science*, pages 156–165, Lisboa, Portugal, August-September 2005. Springer.
6. Andreas Knüpfer. Personal communication, 2009.
7. Andreas Knüpfer, Holger Brunst, Jens Doleschal, Matthias Jurenz, Matthias Lieber, Holger Mickler, Matthias S. Müller, and Wolfgang E. Nagel. The Vampir performance analysis tool set. In Michael Resch, Rainer Keller, Valentin Himmler, Bettina Krammer, and Alexander Schulz, editors, *Tools for High Performance Computing*, pages 139–155. Springer, July 2008.
8. Andrej Kühnal, Marc-André Hermanns, Bernd Mohr, and Felix Wolf. Specification of inefficiency patterns for MPI-2 one-sided communication. In Wolfgang E. Nagel, Wolfgang V. Walter, and Wolfgang Lehner, editors, *Euro-Par*, volume 4128 of *LNCS*, pages 47–62. Springer, 2006.
9. Message Passing Interface Forum. MPI-2: Extentions to the Message-Passing Interface, 1997. Version 2.0, http://www.mpi-forum.org/.
10. Arthur A. Mirin and William B. Sawyer. 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.
11. Kathryn Mohror and Karen L. Karavanic. Performance tool support for mpi-2 on linux. In *Proceedings of the 2004 ACM/IEEE conference on Supercomputing*, SC '04, pages 28–40, Washington, DC, USA, 2004. IEEE Computer Society.
12. Scalasca, 2010. http://www.scalasca.org/.
13. Sameer S. Shende and Allen D. Malony. The TAU parallel performance system. *International Journal of High Performance Computing Applications*, 20(2):287–331, 2006.
14. Hung-Hsun Su, M. Billingsley, and A.D. George. Parallel performance wizard: A performance analysis tool for partitioned global-address-space programming. In *Parallel and Distributed Processing, 2008. IPDPS 2008. IEEE International Symposium on*, pages 1–8, april 2008.
15. Felix Wolf and Bernd Mohr. Automatic performance analysis of hybrid MPI/OpenMP applications. *Journal of Systems Architecture*, 49(10-11):421–439, 2003.