# A scalable replay-based infrastructure for the performance analysis of one-sided communication

Marc-André Hermanns
German Research School for Simulation Sciences
Dept. of Computer Science
RWTH Aachen University
m.a.hermanns@grs-sim.de

Sriram Krishnamoorthy
Comp. Science and Math. Div.
Pacific Northwest National Laboratory
Richland, WA
sriram@pnl.gov

Felix Wolf
German Research School for Simulation Sciences
Dept. of Computer Science
RWTH Aachen University
Jülich Supercomputing Centre
Forschungszentrum Jülich
f.wolf@grs-sim.de

## ABSTRACT

Partitioned global address space (PGAS) languages combine the convenient abstraction of shared memory with the notion of affinity, extending multi-threaded programming to large-scale systems with physically distributed memory. However, in spite of their obvious advantages, PGAS languages still lack appropriate tool support for performance analysis, one of the reasons why their adoption is still in its infancy. Some of the performance problems, for which tool support is needed, occur at the level of the underlying one-sided communication substrate, such as the Aggregate Remote Memory Copy Interface (ARMCI). One such example is the waiting time in situations where asynchronous data transfers cannot be completed without software intervention at the target side. This is not uncommon on systems with reduced operating-system kernels such as IBM Blue Gene/P where the use of progress threads would double the number of cores necessary to run an application. In this paper, we present an extension of the Scalasca trace-analysis infrastructure aimed at the identification and quantification of progress-related waiting times at larger scales. We demonstrate its utility and scalability using a benchmark running with up to 16,384 processes.

## Keywords

performance analysis, event tracing, one-sided communication, remote memory access

## 1. INTRODUCTION

The evolution of high-performance computing (HPC) systems in the last decade has shown an exponential increase in parallelism. Computing systems among the top ten of the 500 fastest supercomputers in the world today feature an average of more than 150,000 cores [18]. Currently the largest system in terms of the number of cores offers a total of 294,912 cores on 72,728 distributed-memory nodes. At larger scales, even small waiting times can propagate and accumulate throughout the application and significantly prevent acceptable application performance [4]. Performance analysis tools for HPC platforms are designed to aid the developer in the often overwhelming task of investigating and understanding the application's behavior at such large scale. However, they are often focussed only on the prevalent programming paradigm—message passing using the Message Passing Interface (MPI) [20].

With the advent of partitioned global address space (PGAS) languages, purely one-sided communication libraries gain more momentum, as these are employed in the communication runtime of those languages. In one-sided communication, all communication parameters, such as source and destination memory locations, are provided by one of the communication partners only—the origin. The second communication partner—the target—does not explicitly call a communication function to match the origin's communication call. Seen from the programmer's view, one-sided data transfers complete without active participation of the target. Among such one-sided communication libraries is the Aggregate Remote Memory Copy Interface (ARMCI) [23], used as the communication back-end of Global Arrays [22], a PGAS-style library. The efficiency of the communication relies much on whether the data exchange can be completed without the active participation of the other process. This is often provided through the communication hardware's remote direct memory access (RDMA) support. When this support is unavailable either for the entire platform or only for a specific type of communication construct, a software component provides this progress. While sometimes this component can be executed by a helper thread, large-scale architectures with reduced kernels such as IBM's Blue Gene/P require an extra core to run it, effectively doubling the required hardware. Interrupt-driven progress, an alternative to a dedicated thread, on the other hand, introduces the cost of an interrupt for every communication call and may pollute the cache. Without a separately scheduled progress engine, however, progress can only occur when the application calls the communication library directly. Yet, one of the inherent characteristics of PGAS applications is that individual processes do not necessarily communicate at the same time. Significant waiting times can therefore occur at the origin of a one-sided operation, while it is waiting for progress at the target side. In addition, inter-process dependencies may induce further waiting times on remote processes via propagation, even if the original waiting times are small [4]. The impact of absent remote communication progress on application performance has not been studied before, but knowing it is crucial to assess the costs of alternatives such as extra threads or interrupts.

To assist in performance tuning at larger scale, performance-analysis tools must be scalable as well. Event tracing is a widely-used method for performance analysis of parallel applications, and it has been successfully applied by several performance-analysis tools [7, 15, 17, 21, 24] available on typical HPC platforms. We have shown in previous work that trace-based performance analysis can be successfully employed at large scale [26]. The main advantage of event tracing comes from the richness of the inter-process information that can be captured, allowing the analysis of extremely complex inter-process relationships.

Waiting time implied through insufficient message progress on the remote side is an example of such an inter-process relationship, where event data from multiple processes have to be taken into account. The waiting time on the remote process can only be quantified by knowing start and end time of the communication call on the origin, as well as of the progress function on the target.

The number of performance analysis tools supporting one-sided communication libraries is currently rather small. The Parallel Performance Wizard (PPW) [17] supports the analysis of general one-sided communication constructs. However, it relies on the GASP interface [16], which, although specifically designed for the analysis of PGAS applications and one-sided communication, is unfortunately not yet widely supported by current one-sided communication libraries. To the best of our knowledge, only GASNet [5] and Quadrics SHMEM [2] support this measurement interface so far. The Charm++ parallel-programming framework [13] supports the investigation of one-sided communication through its proprietary performance tool Projections. MPI PERUSE [12] allows implementation-internal events related to MPI one-sided implementations to be captured, and could be used to obtain the neceassry internal information. Yet, it is limited to MPI and to the best of our knowledge is only supported by OpenMPI [19]. The Cray Pat and Apprentice performance tools [25] support measurement of Cray SHMEM [1] using a mixture of instrumentation and sampling. The TAU performance toolkit [24] has recently been extended to support measurement and analysis of Global Arrays and ARMCI calls [9], however, it records only time profiles and the communication matrix. In their study of system-specific waiting times, Balaji and colleagues have investigated overheads related to the communication of non-data in the MPI implementation on Blue Gene/P [3], focussing on another architecture characteristic of these systems—the comparatively low clock rate of the compute elements.

In our earlier work in the context of the Scalasca performance analysis tool [8], we have shown how large-scale parallel trace analysis can be facilitated using parallel message replay. Until now, supported communication constructs include MPI point-to-point, collective, and one-sided operations with active target synchronization. The latter can be easily accomplished [10] because the active target synchronization following the one-sided exchange, which involves both parties, provides a welcome opportunity to exchange relevant information during the replay.

However, ARMCI one-sided communication provides only passive target synchronization, which does not actively involve the target process. During the replay, the origin process, where the progress-related waiting time occurs, would not know the location of relevant information on the target processes, and the target process would not know how to locate this information on behalf of the origin process. The missing opportunity for data exchange poses serious challenges for Scalasca's trace-based performance-analysis approach. In this work, we present two advanced techniques for data exchange during the reply of one-sided communication that overcome the absence of triggering events on the target side. We describe how we use these techniques to detect and quantify the waiting times caused by untimely remote progress in one-sided communication. We demonstrate this functionality using a Global Arrays matrix-multiplication benchmark on multiple scales up to 16,384 processes.

The remainder of this paper is organized as follows. Section 2 gives an overview of the Aggregate Remote Memory Copy Interface (ARMCI), which is the one-sided library subject to our investigation. We present the event model that we use to model ARMCI communication in Section 3. Based on this model, we define the *Wait for Progress* inefficiency pattern in Section 4. Section 5 gives a short introduction to Scalasca's message-replay driven analysis and presents our extension to the replay-mechanism in detail, followed by results of analyzing a Global-Arrays-based application in Section 6. Concluding this paper, Section 7 summarizes our work and makes a suggestion for future applications of our technique.

## 2. ARMCI

The Aggregate Remote Memory Copy Interface (ARMCI) is a library that provides one-sided communication functionality on distributed memory architectures. It is a portable library optimized for most major communication substrates, including Sockets, Infiniband, Portals, Gemini, DCMF, and the MPI two-sided API. It forms the basis of the Global Arrays library, GPSHMEM, and an earlier version of the Rice University's Co-Array Fortran compiler.

ARMCI is compatible with MPI and shares its process model. The remote memory used as a target of communication is collectively allocated by all processes through the ARMCI API. ARMCI supports a variety of communication idioms that correspond to copying data between local and remote memory regions. This includes blocking and non-blocking communication of contiguous, strided, and vector data, remote atomic operations, and memory synchronization primitives. In addition to the *put* and *get* primitives, ARMCI supports the *accumulate* primitive to atomically add a value to a remote location.

ARMCI provides blocking and non-blocking variants for a subset of communication primitives. The blocking variants

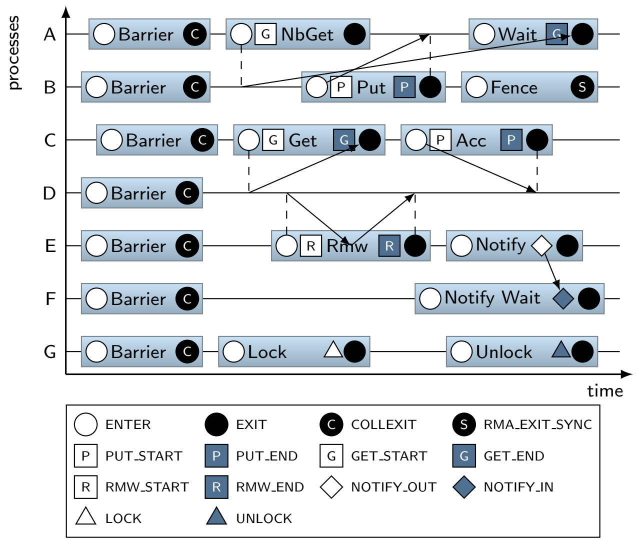


**Figure 1: Type and location of events used to model ARMCI communication.**

return after the operation completed locally at the origin. The non-blocking variants return to the application as soon as possible after initiating the operation. To ensure local completion at the origin, a separate test or wait function has to be called. It is the developer's responsibility to ensure that the communication buffer remains valid between initiation and completion of the operation.

ARMCI has been developed in close collaboration with application domains, and the design of its functionality has been directed by usage modes in higher-level libraries employed in applications. While the supported base functionality, such as contiguous put and get operations, can be handled on many systems by the network interface card (NIC), extended functionality, such as accumulate operations, often requires computation at the remote side for efficient implementation. ARMCI has been designed to support a partitioned global address space view that is closely aligned with distributed shared memory (DSM). In the spirit of DSM systems, one-sided access to remote data does not require any participation from the remote process. For operations that cannot be supported by the NIC, a data server thread is launched on each SMP node to satisfy incoming request. This simplifies programming with the user not having to reason about periodic invocation of calls to the runtime to ensure progress of incoming communication. However, this additional data server thread incurs a performance overhead by consuming computational resources. Architectures with reduced threading support, such as IBM Blue Gene/P, either do not support data server threads in certain configurations or require an extra core to be reserved for every progress thread, doubling the required number of cores per process.

## 3. EVENT MODEL

The current version of Scalasca is based on direct instrumentation, which means that extra code is inserted at specific points in the code—typically at routine entries and exits and inside wrappers around communication routines. The latter is necessary for the acquisition of communication metrics. Whenever the control flow passes one of these instrumentation points, an event is triggered and with it the associated measurement logic. The types and attributes of these events together with their usage constraints are defined in an *event model.* Direct instrumentation as opposed to statistical sampling not only ensures that all performance-relevant events are properly captured but also simplifies access to parameters of communication routines, an important ingredient of parallel performance data. If needed, the resulting runtime dilation, which is highly application dependent, can be lowered by filtering irrelevant events such as those around many frequently called but otherwise very short functions (e.g., getters and setters). In tracing mode, Scalasca simply collects all encountered events along with a timestamp and their event-type-dependent attributes in a memory buffer, which is later flushed to disk. At the end of the execution, the events of every process are stored in a separate file. The whole set of files is then subjected to an automatic pattern search, which is outlined in Section 5.

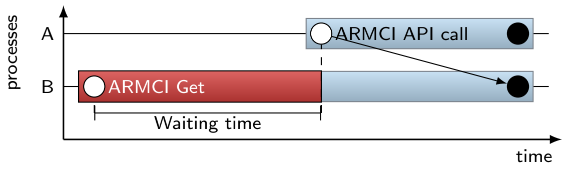
In our earlier work [11], we introduced an event model to record MPI one-sided communication. Here, we reuse this event model, and extend it to accommodate the additional features provided by ARMCI, namely the notify-wait[1] and read-modify-write constructs. Figure 1 illustrates the semantics of the different event types in a timeline diagram. Each call to the ARMCI application programming interface (API) creates an *enter* event after entering a function, and an *exit* event before leaving the function. Collective calls, such as `ARMCI_Barrier`, use the special collective exit event *collexit* to indicate that the function call has collective semantics, which can be exploited during analysis. The call to `ARMCI_Fence` uses a another special exit event called *RMA exit sync* to mark its explicit synchronization with another process.

For remote memory operations, the start of the individual operation is recorded directly after signaling the function entry. The model distinguishes among *put* events for put and accumulate, *get* events for get, and *RMW* events for read-modify-write operations. For each of these operations, two distinct events are defined to mark their beginning and their end. For the blocking interface, start and end event are both contained in the same region instance. For the non-blocking interface, the start event is recorded during the initiating call (e.g., get from D to C), whereas the event marking the end of the operation is recorded during the completion call that actually completes the operation (e.g., non-blocking get from B to A). This effectively models the operation to occur within its completion interval at the origin. For put and accumulate calls this might denote the time when the operation is also completed at the target. Completion at the target is currently not explicitly modeled. However, when reasoning about the performance of put and accumulate calls, only the completion at the origin is relevant. The call to fence generates its own events along with the necessary synchronization information.

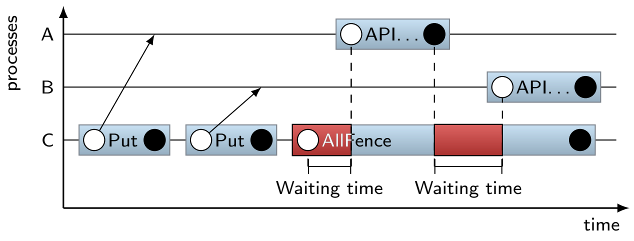
## 4. THE WAIT-FOR-PROGRESS PATTERN

The *Wait for Progress* pattern describes the waiting time on the origin process due to untimely progress on the target

[1]The analysis of waiting times using notify-wait synchronization is not the subject of this paper, and is mentioned here only for completeness's sake.



(a) Pattern with a single target process.



(b) Pattern with two or more target processes.

**Figure 2: Wait for Progress inefficiency pattern.**

process. We define the waiting time for calls with a single target (as shown in Figure 2a) as the timespan between the enter event of the ARMCI call on the origin and the enter event of the first potentially advancing function call on the target side. For the Blue Gene/P platform, we assume any call to ARMCI and MPI to be capable of advancing an ongoing communication.

For non-collective calls with multiple targets such as the synchronization call `ARMCI_AllFence`, as shown in Figure 2b, we define the waiting time to be the time on the origin process that has no overlap with the first potentially progressing call on one of the targets it communicates with. The waiting time is therefore not necessarily a contiguous interval at the beginning of the function call, but can consist of multiple parts.

We assume that the target performs a complete advance on the pending communication. That is, it will not split the advancement of a communication between several calls to the API on the target. Thus, for our progress detection heuristic, only the first occurrence of a region potentially progressing the communication is included in the evaluation.

## 5. REPLAY METHODOLOGY

The Scalasca trace analyzer [7] searches a distributed application event trace in parallel for predefined inefficiency patterns and quantifies their performance impact, such as the amount of waiting time incurred. This impact is called the *severity* of the pattern in Scalasca terminology. The pattern definition is usually based on the relationships between events on two or more processes. To ensure the scalability of the parallel analysis, it is conducted at the same scale (i.e., the same number of processes) as the measurement run. This enables each analyzing process to load a single local trace into memory and to traverse it simultaneously along with the other processes using a replay infrastructure called PEARL [6], a parallel C++ library for high-level event trace access. Forming the backbone of many of the parallel tools in the Scalasca toolset, it provides random access to the local event trace and abstractions such as links between related events through its event access API. Furthermore, it supplies a sophisticated callback subscription mechanism

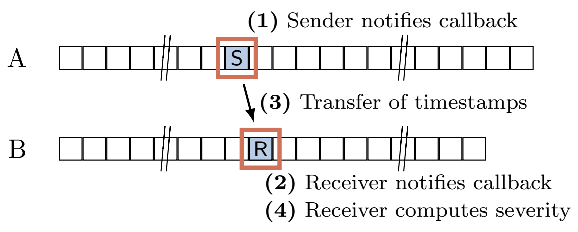

(1) Sender notifies callback
A
S
(3) Transfer of timestamps
B
R
(2) Receiver notifies callback
(4) Receiver computes severity

**Figure 3: Data exchange for point-to-point communication during trace analysis.**

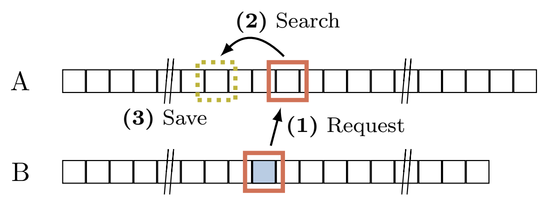
to trigger actions when seeing specific events during trace traversal. Such triggering events include basic events as defined in the event model as well as higher-level events triggered from within another callback.

The general idea of the replay-based analysis is to traverse the trace file in parallel and to reenact the recorded communication based on the information available in the trace. When reaching a communication event, information relevant to the detection and quantification of inefficiency patterns potentially associated with this communication is exchanged using a communication operation of similar type. That is, an MPI point-to-point communication is analyzed using an MPI point-to-point communication, but not necessarily using precisely the same call that was recorded in the trace. This is an important detail to keep in mind when we discuss the analysis of ARMCI operations.

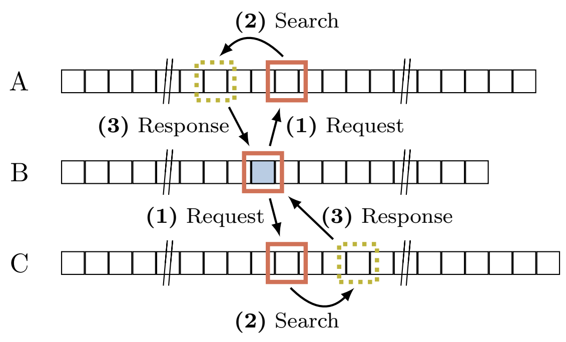
For all communication constructs previously supported by the pattern search [7, 10], both processes of the data exchange are able to rely on local events on both sides of the inter-process relation to trigger the communication needed for the analysis. Even the active target synchronization of MPI one-sided communication involves both processes in each RMA epoch, and therefore enables data exchange regarding the communication within such an epoch.

As an example, Figure 3 shows how the point-to-point communication pattern Late Sender is analyzed using this replay infrastructure. The Late Sender pattern describes waiting time at the receiver due to an belated start of the corresponding transfer at the sender. The send event (S) with the recorded parameters of this transfer is stored in the sender's local trace, while the receive event (R) is stored in the receiver's local trace. Both local traces are traversed from beginning to end (left to right in the figure). Upon reaching the send event, the analysis process representing the sender invokes a callback that sends local timestamp information to the receiver. Likewise, upon reaching the receive event, the analysis process representing the receiver invokes a callback that accepts the timestamp information from the sender. After the data has been exchanged (3), the receiver can compute the pattern severity (4) from remote and local timestamp information.

Unfortunately, in the case of one-sided communication constructs that do not involve active participation of the target process, the target trace does not contain any events related to the exchange that could be used to trigger the collection of local information required to compute the pattern severity. After all, only the communication parameters stored at



(a) Decoupled data exchange.



(b) Direct data exchange.

**Figure 4: The two one-sided data exchange schemes. The decoupled data exchange is used for a single target. The direct exchange is used for multiple targets.**

the origin of a one-sided operation are present in the trace.

Another notable characteristic of the Scalasca trace analyzer is that the measured pattern severity needs to be saved at the process where it occurred. This requirement is motivated by the reduced memory footprint possible if the process coordinate of severity data is merely implied. This means that waiting times, regardless of where they are calculated during the replay analysis, have to be communicated to the process where they occur before the analysis report is written. For one-sided communication constructs this is the origin process. The complete data needed to calculate the interprocess behavior, however, is not always locally available at the origin, and the target lacks events to efficiently trigger the exchange of data needed at the origin.

To enable this data exchange in the context of purely one-sided communication, we implemented two request-response schemes as an extension of Scalasca's replay engine, which are selected depending on the number of target processes involved in the pattern to be analyzed. In the case of a single target, the analyzer employs a light-weight approach, where a single put operation is needed to send the timestamp information of the origin to the target process. In the case of two or more target processes, the analyzer employs a full request-response message exchange with more than two communication partners. The latter imposes a stricter synchronization between the origin and its targets. In the following, we elaborate on these data exchange schemes and the necessary infrastructure to enact them.

## 5.1 Decoupled data exchange

For patterns that involve only a single target process, such as the Wait for Progress pattern inside get (as depicted in Figure 2a) and accumulate calls, the computation of the pattern severity can be split into two parts: the request and an aggregated response. As shown in Figure 4a, the origin process (1) deposits a request with the enter timestamp of the get and accumulate call in the memory of the target process. Once the request is discovered by the target process, it (2) searches for a local progress region overlapping with the get or accumulate call, and (3) directly computes the waiting time from the available information, which is then saved and aggregated on a per call-path basis. At the end of the replay, all aggregated severities are transferred back to their respective origin to be saved in the final analysis report.

## 5.2 Direct data exchange

Figure 2b in Section 4 shows the Wait for Progress pattern for calls with multiple target processes, such as the synchronization call `ARMCI_AllFence`. In this case, the pattern severity cannot be computed remotely, as no process can obtain the complete view of the pattern with a single data transfer. All necessary timespans are therefore collected at the origin, which then performs the severity computation. Thus, in the direct data exchange (as depicted in Figure 4b), the origin process (1) sends requests to all targets involved, which (2) conduct the local search as they would do in the decoupled case. However, instead of computing the severity, each target (3) sends its corresponding timestamp information to the requesting process. When all remote timestamps have been received by the origin, it can compute the overall waiting time.

To ensure timeliness of the overall analysis and to reduce propagation of waiting times in the analysis process itself, the origin process continues its local replay. The progress callback, discussed in greater detail later on, takes care of tracking whether all replies to a request have arrived and eventually computes the pattern severity.

## 5.3 Replay infrastructure

To enable the above-mentioned data exchange schemes, the existing replay infrastructure of our toolset had to be extended. First, timely and correct handling of requests and responses had to be ensured in the absence of appropriate triggers on the target side. Secondly, communication buffers had to be provided for use with one-sided communication. Finally, a finalization of the data exchange had to be implemented.

Analogous to the data server in ARMCI, a progress callback was added to Scalasca's replay infrastructure. This callback serves as the analysis's own progress engine—a virtual progress thread—and is multiplexed into the standard replay mechanism, by registering it for all available basic event types. This ensures that the progress callback is called at least once for each event encountered during the local replay. In case the local analysis process is required to wait, it can do so by repeatedly triggering the progress callback, to ensure request-response completion, and then checking the condition it is waiting for again. The main advantage of using a callback instead of periodically calling a specific function to ensure progress of the local analysis is that it enables the support of multiple one-sided communication libraries at the same time. As long as all of these libraries implement and register their progress callbacks appropriately, their progress is ensured. An example for such a combination would be a one-sided library layered on top of another. An implementation of the progress callback needs to fulfill the following tasks: (1) provide progress for the one-sided library it is used to analyze, (2) check for and process requests and responses available in the local buffers, and (3) check previously cached requests and send appropriate responses.

For point-to-point and collective communication, the buffer space can be created ad-hoc during the analysis. In ARMCI, as with many one-sided communication libraries, communication buffers need to be allocated collectively. This implies that all buffers used to exchange performance relevant data have to be allocated in advance. Alas, the optimal buffer sizes can not be computed in advance. Moreover, it is not trivial to reallocate additional communication buffers once the local replay is started, as all processes are potentially at different points in their local replay.

The performance advantage of one-sided communication can only be leveraged when all communication parameters are known to the origin process in advance. Additional queries by the origin process to obtain a remote address that can be used to exchange the relevant information are usually too costly. Likewise, using a single buffer on the target to be used by multiple remote processes will require locks to ensure data integrity. This will again result in degraded communication performance, as it serializes otherwise unrelated communication operations.

To minimize communication and synchronization among the processes, every process initially performs a single atomic read-modify-write operation per remote communication partner to obtain an exclusive memory location to exchange data. With exclusive memory locations, expensive locking can be completely avoided. In the current implementation, it is guaranteed that each process has an exclusive memory location on every remote process to hold a single request.

To ensure data integrity, a process may only send data to the remote buffer if it can be guaranteed that the remote process has finished processing the data currently residing in the buffer. Therefore, each process has a local array of flags registered for one-sided communication, with one entry for each remote process. As this flag resides on the origin, it can be queried with low cost prior to any data transfer. The remote process updates the flag through a one-sided operation (put) after it has processed the data in its buffers. Depending on the flag a request or response is sent or buffered (or remains buffered). Each call to the progress callback checks the flags of processes with pending communication, and eventually sends the data.

As soon as an analyzer process reaches the finalization of the ARMCI replay, it has to ensure that no other remote processes will send requests before it can start with its local finalization. This is facilitated through a non-blocking barrier, implemented using ARMCI communication. Instead of directly disabling request handling, it initiates the non-blocking barrier, and then continues to alternately notify the progress callback and check for barrier completion. Once the barrier is completed, every process is guaranteed to have reached the finalization phase, and no additional request will be made. Therefore, all processes can then collectively engage in the finalization of the replay.

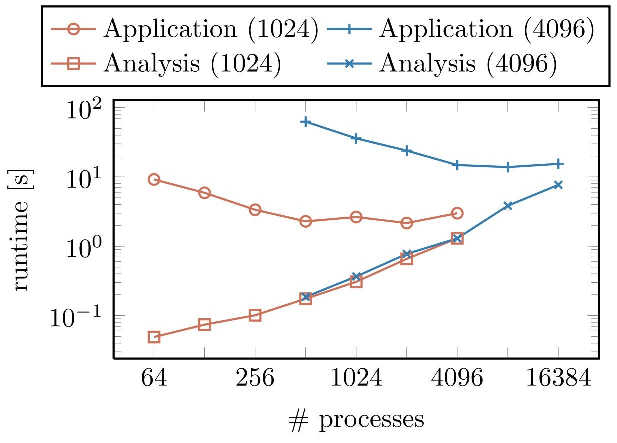


**Figure 5: Strong scaling behavior of the benchmark and the corresponding analysis with two different problem sizes.**

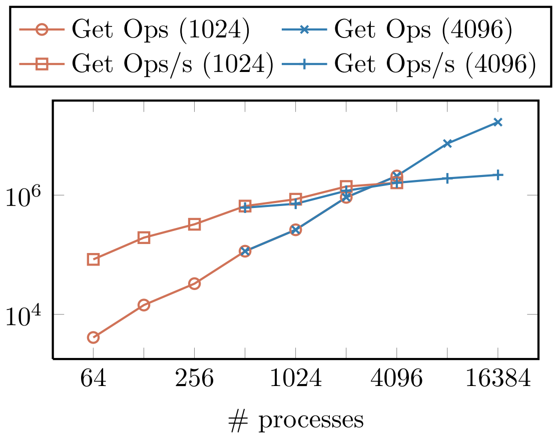


**Figure 6: Total number of get operations in the trace and the processing speed of the analyzer at different scales and problem sizes.**

## 6. RESULTS

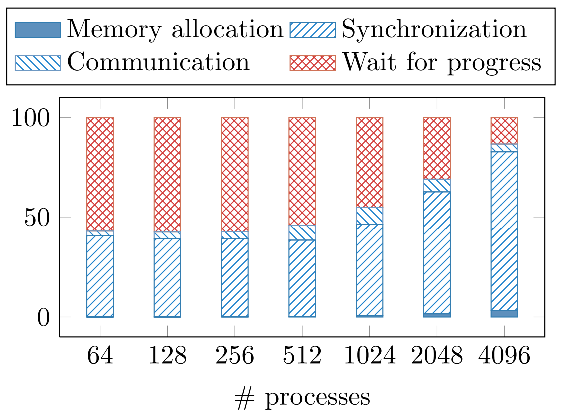
We evaluated our infrastructure and the impact of wait for progress on Jugene, a 72-rack IBM Blue Gene/P system at the Jülich Supercomputing Centre in Germany. It is the installation with the highest number of cores and ranks nine in the current Top500 list (Nov 2010) [18]. It is composed of 73,728 four-way SMP PowerPC 450d compute nodes, resulting in a total of 294,912 cores. The compute nodes run a reduced kernel—the compute node kernel (CNK)—with limited system call functionality. As mentioned previously, one of these limitations is that only a single thread per core can be executed. The Blue Gene/P system provides three different execution modes: virtual node, dual node, and SMP. The virtual node mode spawns one process per core, leaving no room for additional threads. The dual node mode spawns two processes, leaving room for two more threads, and the SMP mode spawns just a single process per compute node, and three additional threads can be spawned. We performed our test in the virtual node mode, with interrupts disabled, to investigate the influence of remote progress in the absence of additional progress threads on application behavior.

We used the SRUMMA algorithm [14] for scalable matrix-matrix multiplication as a test case. The procedure, which is based on remote memory access, is implemented in Global Arrays to support the multiplication of global arrays. This algorithm, invoked as the `ga_dgemm` call, employs an owner-computes model with each process computing a block of the output matrix. The relevant blocks of the input matrices are obtained through non-blocking get operations. The different block-block products are structured so as to avoid contention from numerous simultaneous get requests directed at a target process.

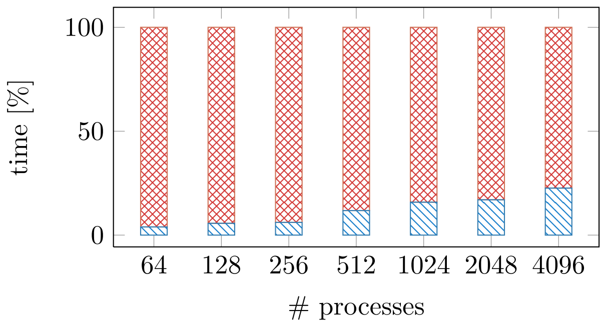
We performed strong scaling experiments for our measurements. The initial problem size for the application was the multiplication of two $1024 \times 1024$ matrices. When the application showed a slowdown above 1024 processes, hitting the point where further strong scaling ceases to produce benefits, as displayed in Figure 5, we increased the matrix size by a factor of 4 for further tests from 512 to 16,384 processes. The square property of the matrix, coupled with the blocked data distribution of the global arrays, results in all processes performing the same number of floating-pointing operations between communication calls. The symmetry is only broken by the differences in the cost of communication due to topological asymmetries. Such regular calculations with seemingly co-ordinated communication are typically not expected to incur much wait for progress penalty.

Another interesting aspect of Figure 5 is the scaling behavior of the analysis time, which seems (a) to be independent of the applications scaling behavior, and (b) to increase at larger scales. Figure 6 reveals that the scaling behavior of the analysis depends on the number of one-sided operations performed by the application, or to put in in other words, the number of requests that need to be handled in the system. The number of communication calls is growing exponentially as the scale is increased, leading to the observed growth in analysis time. Figure 6 also reveals that, even as the overall time to complete the analysis increases, the analysis performance in terms of analyzed operations per second improves at higher scales. This speedup is owed to the fact that the more communication events need to be processed, the higher is the probability of the target process of a request currently performing communication—possibly requesting data from yet another process. As the analyzer itself is also influenced by the absence of remote progress, the increased probability of a communication partner performing communication will lead to reduced waiting times.

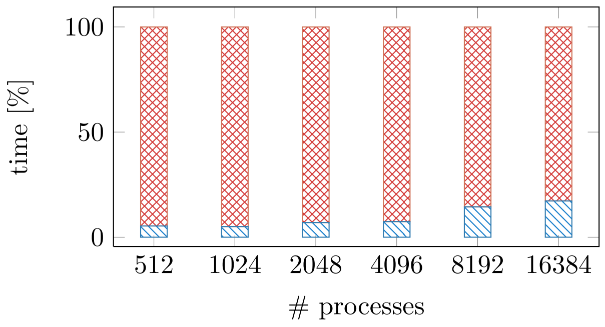
The benchmark application has a balanced load of remote-memory-access operations. For the analyzer, this means the requests it needs to post to analyze the complete behavior are also quite balanced, and even though the number of communication events is very high it reaches a processing speed of over 2 million requests per second in total and over 130 requests per second per process with 16,384 processes. Communication patterns that overburden a single process with data accesses are susceptible to have an impact on the overall performance of the parallel analysis, as the overburdened process will dominate the overall analysis time. We plan on



(a) Distribution of time in ARMCI (1024×1024)



(b) Communication vs. wait for progress with 1024×1024 matrices.



(c) Communication vs. wait for progress with 4096×4096 matrices.

**Figure 7: Breakdown of the time spent within ARMCI during the execution of the SRUMMA benchmark.**

further optimizing the local processing speed, yet, pathological communication patterns may continue to influence the analysis performance.

Figure 7a shows an excerpt of the application's performance metrics, indicating that with the decomposition of the matrices across more processes, the execution is increasingly dominated by synchronization. This can be explained with the complexity of the all-fence operations. However, since our main focus is the waiting time inside the communication, we abstained from plotting this graph beyond 4096 processes. Additionally, the figure shows a significant amount of waiting time in the *Wait for progress* pattern, which we contrast with actual communication in Figure 7b at different scales. The dominance of waiting time in comparison to true communication is significant. Nonetheless, the overall fraction of waiting time is still low enough not to justify the use of an dedicated core to run a helper thread. The severity of the pattern is attenuated slightly at larger scales, as more overall communication increases the probability of a target process to immediately provide progress. Figure 7c confirms this observation also for the larger matrix size with up to 16,384 processes.

## 7. CONCLUSION

We extended the Scalasca trace-analysis infrastructure to investigate the performance of purely one-sided applications using a scalable trace replay methodology. We presented two novel techniques to efficiently exchange relevant information during the replay of one-sided communication traces, overcoming the problem of communication operations not being reflected in the target-local trace.

We demonstrated the usability and scalability of our extended infrastructure using an application benchmark implemented with Global Arrays, a global address space library based on the ARMCI one-sided communication substrate. We were able to measure a previously unstudied inefficiency pattern related to the absence of remote progress, which can occur in some configurations of today's massively parallel systems, with up to 16,384 processes.

Our findings revealed a significant impact of the absence of remote progress on the overall application behavior. They encourage us to intensify the study of this phenomenon with larger applications, such as NWChem, where the severity is expected to be even greater due to more irregular communication patterns. Furthermore, we plan to optimize our implementation, focusing on higher throughput of analyzed one-sided operations to compensate for the effects of uneven analysis workloads. Finally, we intend to use our measurement technique to better understand under which circumstances alternatives such as a progress thread running on a dedicated core or interrupt-driven progress will deliver better or worse performance.

## Acknowledgment

This work was supported in part by the Extreme Scale Computing Initiative, a Laboratory Directed Research and Development Program at Pacific Northwest National Laboratory. The Pacific Northwest National Laboratory is operated for the U.S. Department of Energy by the Battelle Memorial Institute under Contract DE-AC06-76RLO-1830.

## 8. REFERENCES

[1] The Cray SHMEM man pages. Electronically available at `http://docs.cray.com/`.

[2] The Quadrics SHMEM manual. Electronically available at `http://downloads.hpc.vega.co.uk/documentation/ShmemMan_6.pdf`, 2004.

[3] P. Balaji, A. Chan, W. Gropp, R. Thakur, and E. Lusk. The importance of non-data-communication overheads in MPI. *International Journal of High Performance Computing Applications*, 24:5–15, February 2010.

[4] D. Böhme, M. Geimer, F. Wolf, and L. Arnold. Identifying the root causes of wait states in large-scale parallel applications. In *Proceedings of the 39th International Conference on Parallel Processing (ICPP), San Diego, CA, USA*, pages 90–100. IEEE Computer Society, Sept. 2010.

[5] D. Bonachea. GASNet Specification, v1.1. Technical report, University of California, Berkeley, Oct. 2002.

[6] M. Geimer, F. Wolf, A. Knüpfer, B. Mohr, and B. J. N. Wylie. A parallel trace-data interface for scalable performance analysis. In *Proceedings of the 8th International Workshop on State-of-the-Art in Scientific and Parallel Computing (PARA), Umeå, Sweden*, volume 4699 of *Lecture Notes in Computer Science*, pages 398–408. Springer, June 2006.

[7] M. Geimer, F. Wolf, B. J. N. Wylie, and B. Mohr. Scalable parallel trace-based performance analysis. In *Proceedings of the 13th European PVM/MPI Users' Group Meeting (EuroPVM/MPI), Bonn, Germany*, volume 4192 of *Lecture Notes in Computer Science*, pages 303–312. Springer, September 2006.

[8] M. Geimer, F. Wolf, B. J. N. Wylie, and B. Mohr. A scalable tool architecture for diagnosing wait states in massively parallel applications. *Parallel Computing*, 35(7):375–388, July 2009.

[9] J. R. Hammond, S. Krishnamoorthy, S. S. Shende, N. A. Romero, and A. D. Malony. Performance characterization of global address space applications: A case study with NWChem. *Concurrency and Computation: Practice and Experience*, 2011. (under submission).

[10] M.-A. Hermanns, M. Geimer, B. Mohr, and F. Wolf. Scalable detection of MPI-2 remote memory access inefficiency patterns. In *Proceedings of the 16th European PVM/MPI Users' Group Meeting (EuroPVM/MPI), Espoo, Finland*, volume 5759 of *Lecture Notes in Computer Science*, pages 31–41. Springer, September-October 2009.

[11] M.-A. Hermanns, B. Mohr, and F. Wolf. Event-based measurement and analysis of one-sided communication. In *Proceedings of the 11th Euro-Par Conference, Lisboa, Portugal*, volume 3648 of *Lecture Notes in Computer Science*, pages 156–165. Springer, August-September 2005.

[12] T. Jones, R. Dimitrov, K. Rajaram, and K. Mohror. MPI PERUSE: An MPI extension for revealing unexposed implementation information. Technical report, Lawrence Livermore National Laboratory, May 2006.

[13] L. V. Kalé, S. Kumar, G. Zheng, and C. W. Lee. Scaling molecular dynamics to 3000 processors with projections: A performance analysis case study. In *Terascale Performance Analysis Workshop, International Conference on Computational Science(ICCS)*, Melbourne, Australia, June 2003.

[14] M. Krishnan and J. Nieplocha. SRUMMA: a matrix multiplication algorithm suitable for clusters and scalable shared memory systems. In *Parallel and Distributed Processing Symposium, 2004. Proceedings. 18th International*, page 70, 2004.

[15] J. Labarta, S. Girona, V. Pillet, T. Cortes, and L. Gregoris. Dip: A parallel program development

environment. In L. Bougé, P. Fraigniaud, A. Mignotte, and Y. Robert, editors, *Euro-Par'96 Parallel Processing*, volume 1124 of *Lecture Notes in Computer Science*, pages 665–674. Springer Berlin / Heidelberg, 1996. 10.1007/BFb0024763.

[16] A. Leko, D. Bonachea, H.-H. Su, and A. D. George. GASP: A performance analysis tool interface for global address space programming models. Technical Report LBNL-61606, Lawrence Berkeley National Lab, Sept. 2006. Version 1.5 (09/14/2006).

[17] A. Leko, H.-H. Su, D. Bonachea, B. Golden, M. Billingsley, III., and A. D. George. Parallel performance wizard: a performance analysis tool for partitioned global-address-space programming models. In *SC '06: Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, page 186, New York, NY, USA, 2006. ACM.

[18] H. Meuer, E. Strohmaier, J. Dongarra, and H. Simon. The Top500 list. Electronically published at http://www.top500.org/lists/2010/11, Nov. 2010.

[19] B. Mohr, J. L. Träff, J. Worringen, and J. Dongarra, editors. *Recent Advances in Parallel Virtual Machine and Message Passing Interface, 13th European PVM/MPI User's Group Meeting, Bonn, Germany, September 17-20, 2006, Proceedings*, volume 4192 of *Lecture Notes in Computer Science*, Bonn, Germany, Sept. 17-20 2006. Springer.

[20] MPI Forum. MPI: A Message-Passing Interface Standard. Version 2.2, Sept. 4th 2009. available at: `http://www.mpi-forum.org/` (Dec. 2009).

[21] W. E. Nagel, A. Arnold, M. Weber, H. C. Hoppe, and K. Solchenbach. VAMPIR: Visualization and analysis of MPI resources. *Supercomputer*, 12:69–80, 1996.

[22] J. Nieplocha, R. J. Harrison, and R. J. Littlefield. Global Arrays: a nonuniform memory access programming model for high-performance computers. *J. Supercomput.*, 10:169–189, June 1996.

[23] J. Nieplocha, V. Tipparaju, M. Krishnan, and D. K. Panda. High performance remote memory access communication: The ARMCI approach. *Int. J. High Perform. Comput. Appl.*, 20:233–253, May 2006.

[24] S. S. Shende and A. D. Malony. The TAU parallel performance system. *International Journal of High Performance Computing Applications*, 20(2):287–311, 2006.

[25] W. Williams, T. Hoel, and D. Pase. The MPP apprentice performance tool: Delivering the performance of the Cray T3D. In *Programming Environments for Massively Parallel Distributed Systems*, pages 334–345. Birkhäuser Verlag, 1994.

[26] B. J. N. Wylie, M. Geimer, B. Mohr, D. Böhme, Z. Szebenyi, and F. Wolf. Large-scale performance analysis of Sweep3D with the Scalasca toolset. *Parallel Processing Letters*, 20(4):397–414, Dec. 2010.