Parallel Computing 39 (2013) 132–145

Contents lists available at SciVerse ScienceDirect

Parallel Computing

journal homepage: www.elsevier.com/locate/parco

A scalable infrastructure for the performance analysis of passive target synchronization

Marc-André Hermanns [a,b,*], Sriram Krishnamoorthy [d], Felix Wolf [a,b,c]

[a] German Research School for Simulation Sciences, 52062 Aachen, Germany
[b] Dept. of Computer Science, RWTH Aachen University, 52056 Aachen, Germany
[c] Jülich Supercomputing Centre, Forschungszentrum Jülich, 52425 Jülich, Germany
[d] Computer Science and Mathematics Division, Pacific Northwest National Laboratory, Richland, WA, USA

ARTICLE INFO

Article history:
Available online 8 October 2012

Keywords:
Performance analysis
Event tracing
One-sided communication
Remote memory access

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 32,768 processes.

1. Introduction

The evolution of high-performance computing (HPC) systems in the last decade has led to an exponential increase in parallelism. Computing systems in the top ten of the world's 500 fastest supercomputers today feature an average of more than 180,000 cores [1]. In 2011, the largest system in terms of the number of cores (RIKEN's K Computer) offers a total of 548,352 cores on 68,544 distributed-memory nodes. At larger scales, even small waiting times can propagate and accumulate throughout the application and significantly hinder acceptable application performance [2]. Performance-analysis tools for HPC platforms are designed to aid application and library developers, as well as compiler writers, in the often overwhelming task of investigating and understanding the application's behavior at such a large scale. However, they are often focused only on the predominant programming paradigm—message passing using the Message Passing Interface (MPI) [3].

With the advent of partitioned global address space (PGAS) languages, purely one-sided communication libraries gained 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

* Corresponding author at: German Research School for Simulation Sciences, 52062 Aachen, Germany. Tel.: +49 241 80 99753; fax: +49 241 80 6 99753.
E-mail addresses: m.a.hermanns@grs-sim.de (M.-A. Hermanns), sriram@pnnl.gov (S. Krishnamoorthy), f.wolf@grs-sim.de (F. Wolf).

0167-8191/$ - see front matter © 2012 Elsevier B.V. All rights reserved.
http://dx.doi.org/10.1016/j.parco.2012.09.002

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 are completed without active participation of the target. One of these one-sided communication libraries is the Aggregate Remote Memory Copy Interface (ARMCI) [4], used as the communication back-end of Global Arrays [5], a PGAS-style library. The efficiency of the communication relies greatly 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 this component can sometimes 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 [2]. The impact of the lack of remote communication progress on application performance has not been studied before, although this knowledge is crucial to assess the costs of alternatives such as extra threads or interrupts.

To assist in performance tuning at a 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 [6–10] available on typical HPC platforms. We have shown in previous work that trace-based performance analysis can be successfully employed at a large scale [11]. The main advantage of event tracing is the richness of the inter-process information that can be captured, allowing the analysis of extremely complex inter-process relationships.

Waiting time induced by 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 the 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) [10] supports the analysis of general one-sided communication constructs. It relies on the GASP interface [12], a callback interface specifically designed for the analysis of PGAS applications and one-sided communication. Although now supported by several Unified Parallel C (UPC) compilers, it is unfortunately not yet widely supported by current one-sided communication libraries. To the best of our knowledge, only GASNet [13] and Quadrics SHMEM [14] support this measurement interface so far.

The asynchronous parallel-programming framework Charm++ [15] supports the investigation of one-sided communication through its proprietary performance tool Projections. MPI Peruse [16] allows implementation-internal events related to MPI one-sided implementations to be captured, and could be used to measure the necessary internal information. However, it is limited to MPI and to the best of our knowledge is only supported by OpenMPI [17]. The Cray Pat and Apprentice performance tools [18] support measurement of Cray SHMEM [19] using a mixture of instrumentation and sampling. The TAU performance toolkit [9] has recently been extended to support measurement and analysis of Global Arrays and ARMCI calls [20], however, it records only time profiles and the communication matrix. In their study [21], Balaji and colleagues show that system-specific waiting times can be an important factor when analyzing application performance. They investigated overheads in the MPI implementation on Blue Gene/P due to computations done by the implementation itself, focusing 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 [22], we showed how large-scale parallel trace analysis can be facilitated using parallel message replay. So far, supported communication constructs include MPI point-to-point, collective, and one-sided operations with active target synchronization. The latter can be easily accomplished [23] 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. This 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 replay 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 three different applications based on either Global Arrays or ARMCI directly across multiple scales on up to 32,768 processes.

The remainder of this paper is organized as follows. Section 2 gives an overview of the Aggregate Remote Memory Copy Interface (ARMCI), the one-sided library which is the subject of 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 in Section 6 by the results of analyzing three different applications. 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 Rice University's Co-Array Fortran compiler.

ARMCI is compatible with the Message Passing Interface (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 return after the operation is 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 designed in close collaboration with developers from different 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 requests. This simplifies programming since the user does not have 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, effectively 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 reduced 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 paper [24], 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. Fig. 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 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 between *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 enclosed in the same region instance (e.g., get from D to C in Fig. 1).

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

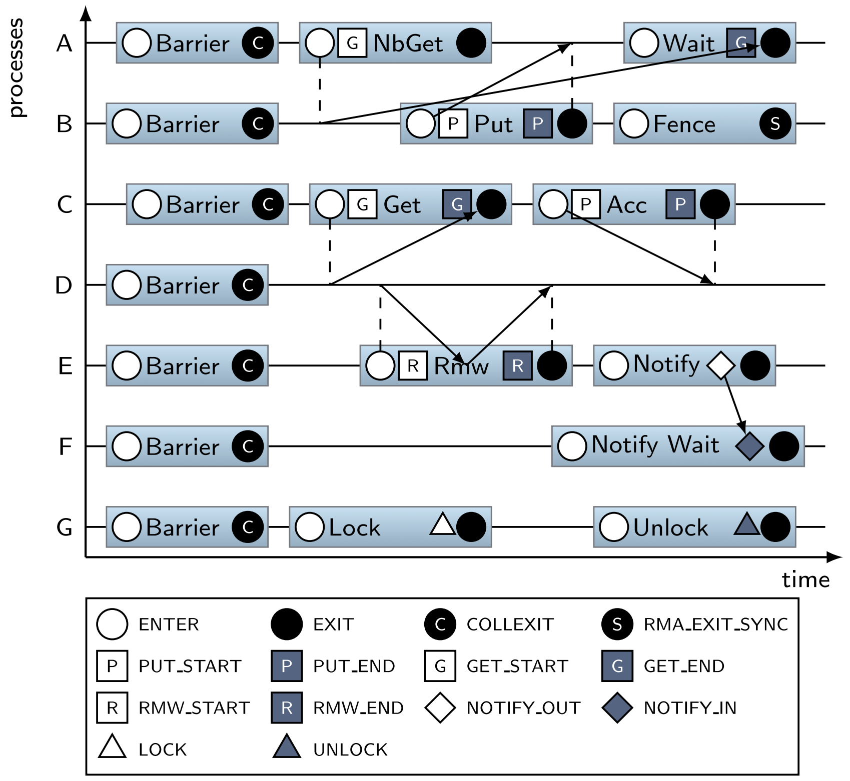


Fig. 1. Type and location of events used to model ARMCI communication.

For the non-blocking interface, the start event is recorded during the initiating call, 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 in Fig. 1). This effectively models the operation as occuring 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 considering 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.

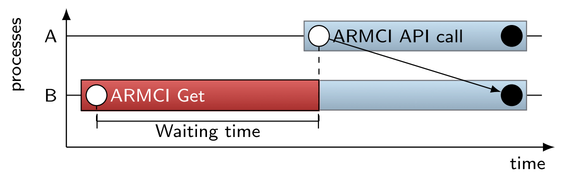
We instrumented the calls to the ARMCI library using a performance tool interface similar to that available for the Message Passing Interface (MPI) [25]. With the introduction of Global Arrays 5.0, the underlying ARMCI communication library provides two symbols for every API call.[2] The weak symbol with the prefix ARMCI can be overridden by an interposed measurement library that provides the measurement instrumentation. A strong symbol with the prefix PARMCI is available to be used in the interposed wrappers to call the original library routines. This interface enables software tools to create wrappers for each ARMCI API call to track the individual parameters the application uses when calling ARMCI.

4. The Wait for Progress pattern

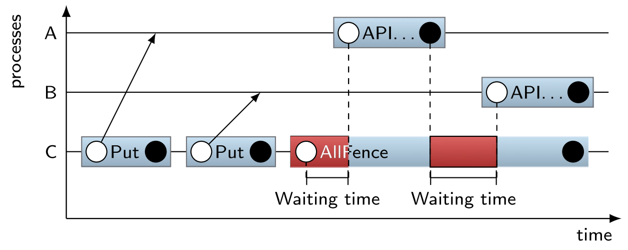
The *Wait for Progress* pattern describes the waiting time on the origin process due to untimely progress on the target process. We define the waiting time for calls with a single target, as shown in Fig. 2(a), as the timespan between the enter event of the ARMCI call on the origin and the enter event 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 put operations the situation is slightly different. Here, local completion can be achieved through buffering, or injection of the message into the network, even if the put operation itself is not necessarily complete at the target location. Therefore, with the given inquiry methods waiting time for the put operation cannot be inferred. However, if the user waits for the completion of puts at the target using the non-collective synchronization call `ARMCI_AllFence` (as shown in Fig. 2(b)), progress-related waiting times can naturally occur.[3] 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. While this may underestimate the real waiting time, it ensures the time needed for the actual transfer is never assessed as waiting time.

[2] Prior to Global Arrays 5.0, we used the GNU binutils to create a similar effect, renaming existing symbols and adding new symbols to the library.
[3] Note that the fence call is not mandatory for the put completion in ARMCI.



(a) Pattern with a single target process.



(b) Pattern with two or more target processes.

Fig. 2. Wait for Progress inefficiency pattern.

Neither the origin nor the targets have the complete view of the timespans involved to compute this pattern directly. All necessary timespans are therefore collected at the origin, which then performs the severity computation. We assume that the target completes all pending communication whenever it reaches a progress-enabling function. That is, the target process will not split the advancement of a communication between several calls to the higher-level API. This is absolutely required in polling mode, which we consider here, because it would be risky to delay pending remote communication requests more than necessary. After all, the target cannot guarantee a timely re-entering of the communication layer to continue the progress. Thus, for our progress detection heuristic, only the first occurrence of a region potentially advancing the communication is included in the evaluation. In situations where the implementation chooses to split the advancement of a message between several calls on the target side, our heuristic will therefore underestimate the waiting time incurred.

Without measurement data from lower communication layers the waiting time inside the ARMCI and MPI layer has to be estimated. Our initial assumption is that the origin process is idle during waiting time not overlapped with remote regions of its targets. However, we concede that the timespan attributed to waiting time could in principle be used to progress an on-going remote call. Although this is currently not taken into account by our analysis, it could be addressed by two extensions. First, in the presence of locally pending non-blocking calls, it can be checked whether the identified waiting time could possibly be used for progressing the pending messages. This can be done by an additional request to the target for every timespan spent in the communication layer, between the initiation of the non-blocking call and its completion. Second, with tracking of local waiting patterns as well as search requests by remote processes, local regions that may progress remote calls can be identified. In this way, timespans in which the implementation can potentially progress a call of a remote origin on the target can be found. Nevertheless, since it is much less complicated, we think that even in its current form our heuristic will give a valuable indication of the inter-process dependencies and the amount of waiting time induced by them.

5. Replay methodology

The Scalasca trace analyzer [8] searches an event trace, which is distributed across multiple files (i.e., one file per process), 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 [26], 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 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 re-enact 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 associated with this communication is exchanged using a communication operation of the same class. 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.

Using similar communication operations for the exchange of event data enables efficient communication, as the communication specific semantics can be leveraged. Collective communication can be used efficiently in the replay, because in the presence of a correct communication trace the local process can safely call the collective calls, as all other participating processes will also call the operation at the appropriate time. Likewise, with point-to-point communication, a receiving process can issue a receive call with the communication parameters obtained from the application trace, as it can rely on the matching message being sent by the corresponding communication partner. For one-sided communication with passive target synchronization, this scheme needed modification, as we will explain later in more detail.

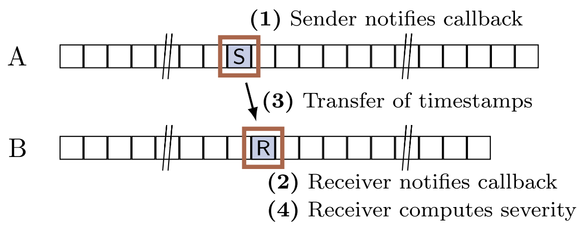

(1) Sender notifies callback
A
S
(3) Transfer of timestamps
B
R
(2) Receiver notifies callback
(4) Receiver computes severity

Fig. 3. Data exchange for point-to-point communication during trace analysis.

For all communication constructs previously supported by the pattern search [8,23], 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, Fig. 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 a 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. This is due to the fact that only the communication parameters stored at 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 memory-footprint reduction that is possible if the process coordinate of the severity data is implicitly stored. 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 inter-process behavior, however, are 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 lightweight 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 Fig. 2(a)), accumulate, read-modify-write or fence calls to a single target, the computation of the pattern severity can be split into two parts: the request and an aggregated response. As shown in Fig. 4(a), the origin process (1) deposits a request with

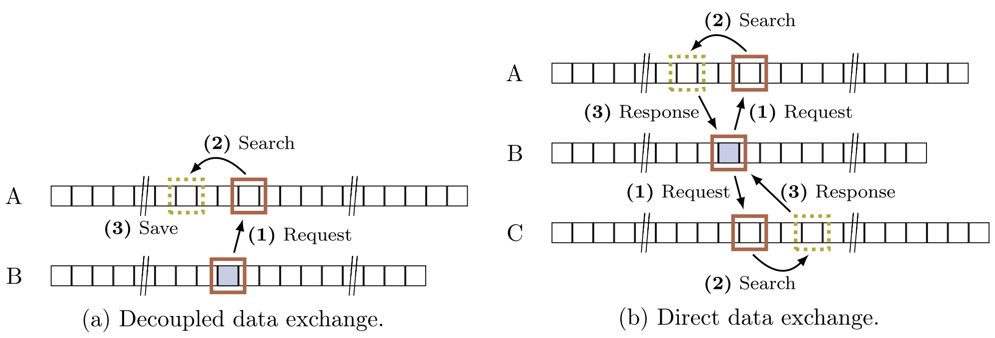


Fig. 4. The two one-sided data exchange schemes. The decoupled data exchange is used for a single target. The direct exchange is used for multiple targets.

the enter and exit timestamps of the RMA operation in the memory of the target process. Once the request is discovered by the target process, it (2) searches for a local region capable of advancing the message 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.

Local progress regions can be any call into the communication layer of the specific platform. For ARMCI, and the presented measurements, we assume all ARMCI and MPI calls to possibly advance the communication. In future developments instrumentation of the DCMF layer on Blue Gene may also become an option for identification of progress regions.

5.2. Direct data exchange

Fig. 2(b) 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 Fig. 4(b), 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 below, tracks 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 of such a combination would be a one-sided library layered on top of another. Implementation of the progress callback needs to fulfill the following tasks: (1) provide progress for the one-sided library it analyzes to ensure requests from remote processes are advanced properly, (2) check for and process requests and responses available in 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. Unfortunately, the optimal buffer sizes cannot be computed in advance. Moreover, it is not trivial to reallocate additional communication buffers once the local replay has started, as this can only be facilitated at global synchronization points in the parallel replay. The presence of additional global synchronization points other that its start and end, however, cannot be guaranteed in the general case.

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. Each process therefore obtains the address of an exclusive memory location to exchange data by performing a single atomic read-modify-write operation on its first communication with a specific communication partner. This minimizes the communication and synchronization overhead among the processes during the data exchange. Using exclusive memory locations, expensive locking can be avoided completely. In the current implementation, it is guaranteed that each process has an exclusive memory location on every remote process to hold a single request. However, this implies that the memory consumption is bounded by $O(p)$ on p processes. To reduce memory consumption of our implementation, we plan to reduce the memory footprint to a user-configurable size. The user can then specify the maximum number of request slots to be reserved for communication. Depending on the application's communication pattern, this may then consequently amount to an $O(1)$ memory footprint (e.g., nearest neighbor communication). In cases where it cannot be guaranteed that processes can exclusively access the request slots, the latter need to be protected by locks. The user will then have to trade performance for lower memory consumption, but can however use an application-dependent sweet spot.

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.

5.4. Backward replay

Identifying wait states in an application is a first step toward understanding its performance. Automatic identification of their root causes is an important next step. In our earlier work, we demonstrated how such a root-cause analysis can be facilitated for MPI applications using Scalasca's parallel replay infrastructure [2]. However, to do this the process-local traces have to be traversed in reverse direction—in a so-called backward replay. The analysis techniques discussed so far in this paper all employed a forward replay: the parallel traversal of the individual local traces from their first to their last event. Messages are then replayed in the same direction as they were recorded. During a backward replay, the individual local traces are traversed from their last to their first event, also reversing the flow of messages. That is, a receive event will trigger the sending of data, while a send event will trigger their receipt.

However, with passive target synchronization, there are no events available to trigger a data transfer in the reverse direction. A backward replay is therefore not directly supported in the current state of implementation. However, in the light of the enhanced analysis options it could offer, a backward replay would be highly desirable. To enable this, additional events would need to be sent to the target during the forward replay and inserted into the target trace as triggers for backward communication. While the first requirement could be handled by the decoupled data exchange presented here, the efficient event insertion in data structures optimized for space efficiency and fast serial traversal remains the main obstacle to a one-sided backward replay. As this would imply critical changes to the runtime system of the parallel analyzer, it was unfortunately beyond the scope of this work. Nevertheless, it may be a direction for further research.

5.5. Further communication approaches

Instead of replaying the communication step-by-step, re-enacting every communication operation separately, one could introduce a bulk replay that accumulates the event data to be exchanged locally and then facilitate the actual exchange only in certain intervals using an all-to-all operation. However, this would require well-defined global synchronization points at which a self-contained exchange is possible and every process knows that now the exchange must be pursued. While some applications do perform global synchronization such as `ga_sync` in regular intervals, this cannot be assumed in the general case. Therefore, only the end of the parallel replay is a guaranteed safe point for such a collective data exchange. Aggregating all communication requests to individual targets across the complete local trace would imply an increased memory footprint roughly proportional to the number of local events. The data exchange techniques presented here are not only architecturally simpler, but also much less memory-intensive. With memory requirements independent of the number of events, they offer a much better balance between communication and memory costs.

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 second-highest number of cores and ranks 12th in the Top500 list of June 2011 [1]. 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 above, 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 one 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 on application behavior in the absence of additional progress threads. A comparison with a version with interrupts enabled was impossible because the support for interrupts in the current ARMCI version was broken.

6.1. Matrix multiplication benchmark

We used the SRUMMA algorithm [27] 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 to avoid contention from numerous simultaneous get requests directed at a target process.

We performed strong scaling experiments for our measurements, multiplying two 4096×4096-element matrices on different numbers of 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 a great Wait for Progress penalty. Fig. 5 shows the overall strong scaling behavior of the benchmark. It can be seen that beyond 8,192 processes, the application fails to scale any further. This is due to the increasing lack of computational work performed by the individual processes and the benchmark-internal synchronization using `ARMCI_AllFence` starting to dominate the total time of the benchmark run. As the synchronization time of the benchmark is not the primary focus of this work, we will not investigate it any further here.

Another interesting aspect of Fig. 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. Fig. 6 reveals that the scaling behavior of the analysis depends on the number of one-sided operations performed by the application, or in other words, the number of requests that need to be handled in the system. In the data-decomposition scheme present in the benchmark, the number of communication calls grows linearly with the number of processes with an estimated coefficient of 2.5, leading to the observed growth in analysis time as the local request-response handling of the local processes is saturated. As the parallel analyzer uses the same communication layer as the measured application, its performance may also be degraded due to the lack of remote progress. The observed slowdown is a direct artifact of the analyzer's increased work load per process and is not expected with weak scaling experiments, even on configurations with a larger number of processes, as we show in Section 6.2.

The benchmark application has a balanced load of remote-memory-access operations. For the analyzer, this means the number of requests necessary to analyze the complete behavior are also quite balanced across all processes. With the largest measurement using 32,768 processes the analyzer was able to maintain a processing speed of 11 million get operations per second thus handling an overall workload of almost 59 million get operations. In general, communication patterns that overburden a single process with data accesses tend to reduce the overall performance of the parallel analysis, as the overburdened process will dominate the overall analysis time. We plan to further optimize the local processing speed (e.g., request coalescing to reduce latency costs), however, pathological communication patterns may continue to influence the analysis performance.

Fig. 7 shows an excerpt from 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 by the complexity of the all-fence operations. Additionally, the figure shows a significant amount of waiting time in the *Wait for Progress*

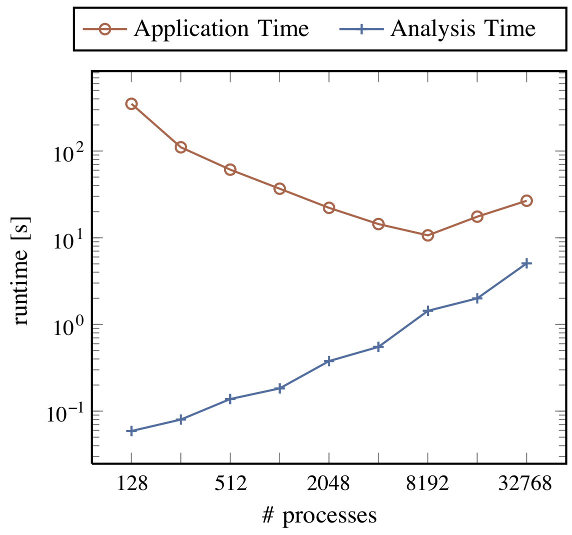


Fig. 5. Strong scaling behavior of the benchmark and the corresponding analysis. With larger scales, the benchmarks uses more communication, resulting in larger event streams to be processed by the analysis.

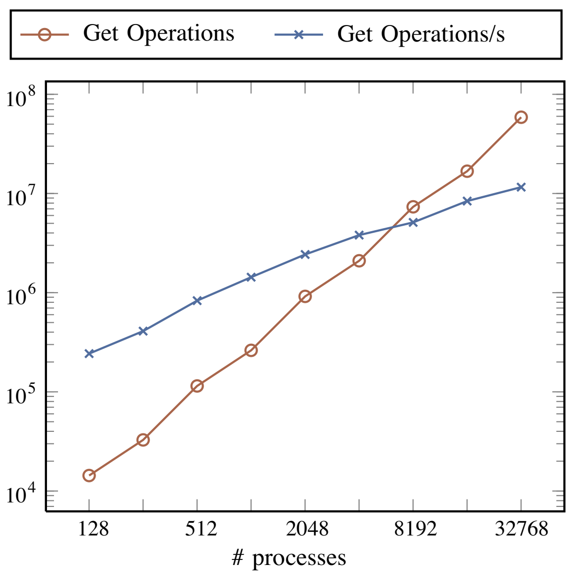


Fig. 6. Total number of get operations in the trace and the processing speed of the analyzer at different scales.

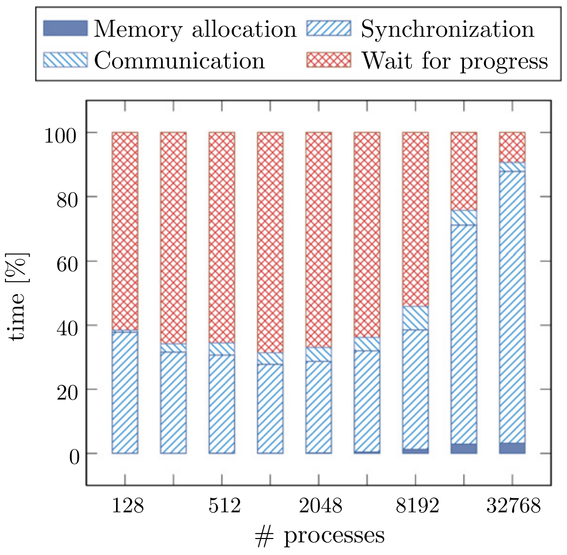


Fig. 7. Breakdown of the time spent within ARMCI during the execution of the SRUMMA benchmark.

pattern, which we contrast with actual communication in Fig. 8 at different scales. The dominance of waiting time in comparison to actual communication is significant. Nonetheless, the overall fraction of waiting time is still low enough that the use of a dedicated core to run a helper thread is not justified. The severity of the pattern is attenuated slightly at larger scales, as more overall communication increases the probability of a target process immediately providing progress.

Although we expected the impact of remote progress to be visible in our runtime configurations, we were surprised by its dominance in the ARMCI communication profile. As the application performs a quite regular and equalized amount of work, this is even more surprising, and reinforces our plan to study more irregular applications.

6.2. SOR benchmark

Whereas the matrix multiplication benchmark demonstrated the performance of our analysis under strong scaling, we conducted experiments with the SOR benchmark to investigate its performance under weak scaling. SOR solves the Poisson equation using a red–black successive over-relaxation method. The two main communication steps are halo exchange and

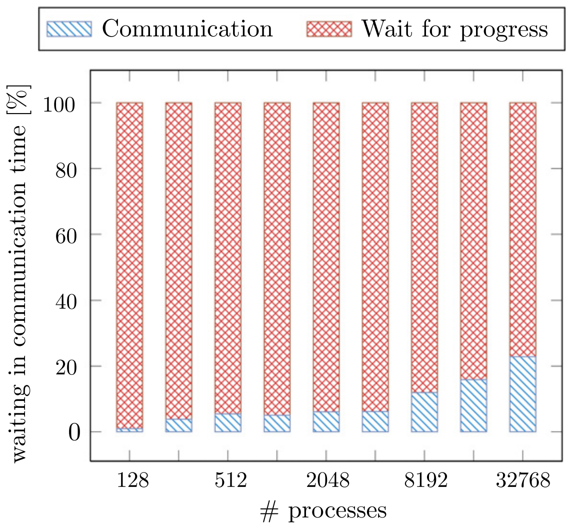


Fig. 8. Percent of waiting time in ARMCI communication.

scalar reduction. The former was adapted to use ARMCI get operations 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 of grid-point codes used in earth and environmental sciences.

The solver was configured to create measurements with roughly the same number of events per process, and specifically not to converge within the predefined maximum number of 1000 iterations. This was needed to reliably evaluate the weak-scaling behavior of our analysis approach.

The time exclusively needed for the replay analysis (i.e., without loading the traces and writing the results, which together took less than 90 s for the 32,768-core run) is reasonably low. As can be seen in Fig. 9, it roughly mimics the overall scaling behavior of the application itself, which is to be expected when using a replay-based approach. At this point, it is worth noting that the time of the replay does not depend on the work load of the application during measurement, but rather directly correlates with the number of communication operations employed during the data exchange. In combination with the benchmark results of the matrix multiplication, it can be seen that our replay analysis strategy is not bound by the number of processes in the measurement, but rather the number of events generated by each process.

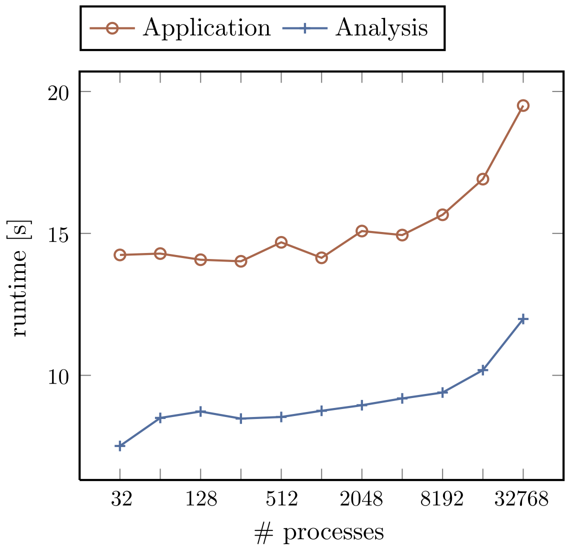
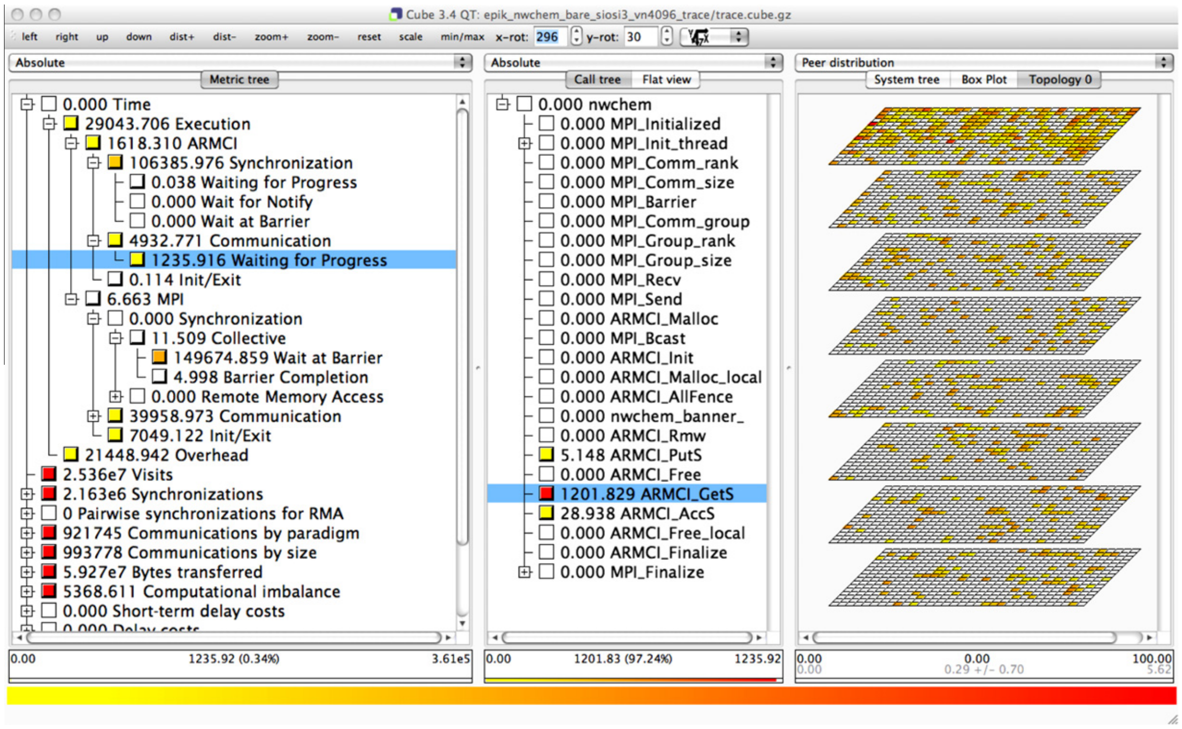


Fig. 9. Weak scaling experiments with an ARMCI-based SOR solver. Due to the communication constructs employed in the benchmark the analyzer only uses the decoupled exchange algorithm.



```
Cube 3.4 QT: epik_nwchem_bare_siosi3_vn4096_trace/trace.cube.gz
left  right  up  down  dist+  dist-  zoom+  zoom-  reset  scale  min/max  x-rot: 296  y-rot: 30

Absolute                                Absolute                          Peer distribution
Metric tree                             Call tree | Flat view             System tree | Box Plot | Topology 0

0.000 Time                              0.000 nwchem
  29043.706 Execution                     0.000 MPI_Initialized
    1618.310 ARMCI                        0.000 MPI_Init_thread
      106385.976 Synchronization          0.000 MPI_Comm_rank
        0.038 Waiting for Progress        0.000 MPI_Comm_size
        0.000 Wait for Notify             0.000 MPI_Barrier
        0.000 Wait at Barrier             0.000 MPI_Comm_group
      4932.771 Communication              0.000 MPI_Group_rank
        1235.916 Waiting for Progress     0.000 MPI_Group_size
      0.114 Init/Exit                     0.000 MPI_Recv
    6.663 MPI                             0.000 MPI_Send
      0.000 Synchronization               0.000 ARMCI_Malloc
        11.509 Collective                 0.000 MPI_Bcast
          149674.859 Wait at Barrier      0.000 ARMCI_Init
          4.998 Barrier Completion        0.000 ARMCI_Malloc_local
        0.000 Remote Memory Access        0.000 ARMCI_AllFence
      39958.973 Communication             0.000 nwchem_banner_
      7049.122 Init/Exit                  0.000 ARMCI_Rmw
    21448.942 Overhead                    5.148 ARMCI_PutS
  2.536e7 Visits                          0.000 ARMCI_Free
  2.163e6 Synchronizations                1201.829 ARMCI_GetS
  0 Pairwise synchronizations for RMA     28.938 ARMCI_AccS
  921745 Communications by paradigm       0.000 ARMCI_Free_local
  993778 Communications by size           0.000 ARMCI_Finalize
  5.927e7 Bytes transferred               0.000 MPI_Finalize
  5368.611 Computational imbalance
  0.000 Short-term delay costs
  0.000 Delay costs

0.00   1235.92 (0.34%)   3.61e5         0.00   1201.83 (97.24%)   1235.92       0.00   0.00   100.00
                                                                                0.00   0.29 +/- 0.70   5.62
```

Fig. 10. Performance analysis report on a simulation run of NWChem using the SiOSi3 input conducted on 4096 cores.

6.3. NWChem

As a third test case, we investigated the NWChem [28] framework for computational chemistry. We performed a density functional theory (DFT) calculation on the SiOSi3 input. DFT is a widely used single-determinant approach to the many-electron problem [29–31]. All integral evaluations were performed using the direct method, and the Fock matrix was constructed using the distributed data approach [32].

The NWChem framework exhibits a vast amount of call paths in this experiment. To keep the overhead and trace size in a manageable range, we disabled all user function instrumentation at runtime, recording a rather flat call tree, as can be seen in the middle pane of the report browser, as shown in Fig. 10. In the left pane, the major source of waiting time is a computational imbalance at global synchronization points. Although the Wait for Progress inefficiency pattern described in this work is not a major part of the overall waiting time in the presented NWChem measurement, it still has a significant influence on the one-sided communication employed in Global Arrays. In this example, it accounts for 20 percent of the ARMCI communication time, which is used for the one-sided data accesses. Fig. 10 also reveals that the attributed waiting time is sparsely scattered across the processes. It is likely that these highly irregular waiting times induce imbalances which themselves lead to further waiting times—a possibility that should be subject of a more detailed investigation of NWChem performance.

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 for the efficient exchange of 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 furthermore demonstrated the usability and scalability of our extended infrastructure using three applications based on Global Arrays, a global-address-space library, and its one-sided communication substrate ARMCI, respectively. With up to 32,768 processes, 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. Our findings revealed a significant impact of stalled remote progress on the one-sided communication of the measured applications, both in smaller benchmarks as well as in the NWChem computational chemistry application [28].

Our results encourage us to study this phenomenon in NWChem with further data sets in pursuit of a generic optimization potential which is independent of the input. On a general level, our techniques can be helpful in examining the communication behavior of other one-sided communication libraries used in the runtime components of partitioned global-address-space languages such as Unified Parallel C [33] or Co-Array Fortran [34]. Combined with measurement data obtained through the GASP interface [12], we would like to enable the investigation of such languages on a very large scale. 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 the circumstances under which alternative progress mechanisms such as a thread running on a dedicated core or interrupts will deliver better or poorer performance.

Acknowledgments

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.

References

[1] H. Meuer, E. Strohmaier, J. Dongarra, H. Simon, The Top500 list, 2011, Electronically published at <http://www.top500.org/lists/2011/06>.
[2] D. Böhme, M. Geimer, F. Wolf, 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), IEEE Computer Society, San Diego, CA, USA, 2010, pp. 90–100.
[3] MPI Forum, MPI: A Message-Passing Interface Standard, Version 2.2, MPI Forum, 2009, Available at: <http://www.mpi-forum.org/>.
[4] J. Nieplocha, V. Tipparaju, M. Krishnan, D.K. Panda, High performance remote memory access communication: the ARMCI approach, Int. J. High Perform. Comput. Appl. 20 (2006) 233–253.
[5] J. Nieplocha, R.J. Harrison, R.J. Littlefield, Global Arrays: a nonuniform memory access programming model for high-performance computers, J. Supercomput. 10 (1996) 169–189.
[6] J. Labarta, S. Girona, V. Pillet, T. Cortes, L. Gregoris, Dip: a parallel program development environment, in: L. Bougé, P. Fraigniaud, A. Mignotte, Y. Robert (Eds.), Euro-Par'96 Parallel Processing, Lecture Notes in Computer Science, vol. 1124, Springer, Berlin/Heidelberg, 1996, pp. 665–674, http://dx.doi.org/10.1007/BFb0024763.
[7] W.E. Nagel, A. Arnold, M. Weber, H.C. Hoppe, K. Solchenbach, VAMPIR: visualization and analysis of MPI resources, Supercomputer 12 (1996) 69–80.
[8] M. Geimer, F. Wolf, B.J.N. Wylie, B. Mohr, Scalable parallel trace-based performance analysis, in: Proceedings of the 13th European PVM/MPI Users' Group Meeting (EuroPVM/MPI), Lecture Notes in Computer Science, vol. 4192, Springer, Bonn, Germany, 2006, pp. 303–312.
[9] S.S. Shende, A.D. Malony, The TAU parallel performance system, International Journal of High Performance Computing Applications 20 (2006) 287–311.
[10] A. Leko, H.-H. Su, D. Bonachea, B. Golden, M. Billingsley III, 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, ACM, New York, NY, USA, 2006, p. 186.
[11] B.J.N. Wylie, M. Geimer, B. Mohr, D. Böhme, Z. Szebenyi, F. Wolf, Large-scale performance analysis of Sweep3D with the Scalasca toolset, Parallel Processing Letters 20 (2010) 397–414.
[12] A. Leko, D. Bonachea, H.-H. Su, A.D. George, GASP: a performance analysis tool interface for global address space programming models, Technical Report LBNL-61606, Lawrence Berkeley National Lab, 2006, Version 1.5 (09/14/2006).
[13] D. Bonachea, GASNet specification, v1.1, Technical Report, University of California, Berkeley, 2002.
[14] The Quadrics SHMEM manual, 2004, Electronically available at <http://downloads.hpc.vega.co.uk/documentation/ShmemMan6.pdf>.
[15] L.V. Kalé, S. Kumar, G. Zheng, C. Lee, Scaling molecular dynamics to 3000 processors with projections: a performance analysis case study, in: P. Sloot, D. Abramson, A. Bogdanov, Y. Gorbachev, J. Dongarra, A. Zomaya (Eds.), Computational Science — ICCS 2003, volume 2660 of Lecture Notes in Computer Science, Springer, Berlin, Heidelberg, 2003,pp. 23–32.
[16] T. Jones, R. Dimitrov, K. Rajaram, K. Mohror, MPI PERUSE: an MPI extension for revealing unexposed implementation information, Technical Report, Lawrence Livermore National Laboratory, 2006.
[17] B. Mohr, J.L. Träff, J. Worringen, J. Dongarra (Eds.), Recent Advances in Parallel Virtual Machine and Message Passing Interface, 13th European PVM/MPI User's Group Meeting, Lecture Notes in Computer Science, vol. 4192, Springer, Bonn, Germany, 2006.
[18] W. Williams, T. Hoel, D. Pase, The MPP apprentice performance tool: delivering the performance of the Cray T3D, in: Programming Environments for Massively Parallel Distributed Systems, Birkhäuser Verlag, 1994, pp. 334–345.
[19] Cray Inc. Cray application developer's environment user's guide. Manual, Cray Inc., May 2012.
[20] J.R. Hammond, S. Krishnamoorthy, S. Shende, N.A. Romero, A.D. Malony, Performance characterization of global address space applications: a case study with NWChem., Concurrency and Computation: Practice and Experience 24 (2012) 135–154.
[21] P. Balaji, A. Chan, W. Gropp, R. Thakur, E. Lusk, The importance of non-data-communication overheads in MPI, International Journal of High Performance Computing Applications 24 (2010) 5–15.
[22] M. Geimer, F. Wolf, B.J.N. Wylie, B. Mohr, A scalable tool architecture for diagnosing wait states in massively parallel applications, Parallel Computing 35 (2009) 375–388.
[23] M.-A. Hermanns, M. Geimer, B. Mohr, 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), Lecture Notes in Computer Science, vol. 5759, Springer, Espoo, Finland, 2009, pp. 31–41.
[24] M.-A. Hermanns, B. Mohr, F. Wolf, Event-based measurement and analysis of one-sided communication, in: Proceedings of the 11th Euro-Par Conference, Lecture Notes in Computer Science, vol. 3648, Springer, Lisboa, Portugal, 2005, pp. 156–165.
[25] MPI Forum, Profiling Interface, in: [3], pp. 453–458, Dec. 2009, Available at: <http://www.mpi-forum.org/>.
[26] M. Geimer, F. Wolf, A. Knüpfer, B. Mohr, 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), Lecture Notes in Computer Science, vol. 4699, Springer, Umeå, Sweden, 2006, pp. 398–408.
[27] M. Krishnan, 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, p. 70.
[28] M. Valiev et al, NWChem: a comprehensive and scalable open-source solution for large scale molecular simulations, Computer Physics Communications 181 (2010) 1477–1489.
[29] W. Kohn, L.J. Sham, Self-consistent equations including exchange and correlation effects, Physical Review 140 (1965) A1133–A1138.
[30] R.G. Parr, W. Yang, Density-Functional Theory of Atoms and Molecules, Oxford University Press, Inc., New York, 1989.

[31] J.P. Perdew, K. Schmidt, Jacob's ladder of density functional approximations for the exchange-correlation energy, AIP Conference Proceedings 577 (2001) 1–20.
[32] R.J. Harrison et al, Toward high-performance computational chemistry: II. A scalable self-consistent field program, Journal of Computational Chemistry 17 (1996) 124–132.
[33] UPC Language Specification, 1.2 edition, 2005.
[34] J. Reid, Coarrays in the next fortran standard, SIGPLAN Fortran Forum 29 (2010) 10–27.