Leveraging MPI-3 Shared-Memory Extensions for Efficient PGAS Runtime Systems

Huan Zhou(✉), Kamran Idrees, and José Gracia

High Performance Computing Center Stuttgart (HLRS),
University of Stuttgart, Stuttgart, Germany
zhou@hlrs.de

Abstract. The relaxed semantics and rich functionality of one-sided communication primitives of MPI-3 makes MPI an attractive candidate for the implementation of PGAS models. However, the performance of such implementation suffers from the fact, that current MPI RMA implementations typically have a large overhead when source and target of a communication request share a common, local physical memory. In this paper, we present an optimized PGAS-like runtime system which uses the new MPI-3 shared-memory extensions to serve intra-node communication requests and MPI-3 one-sided communication primitives to serve inter-node communication requests. The performance of our runtime system is evaluated on a Cray XC40 system through low-level communication benchmarks, a random-access benchmark and a stencil kernel. The results of the experiments demonstrate that the performance of our hybrid runtime system matches the performance of low-level RMA libraries for intra-node transfers, and that of MPI-3 for inter-node transfers.

Keywords: MPI · One-sided communication · Remote-memory access · RMA · Partitioned global address space · PGAS

1 Introduction

The Message Passing Interface (MPI, [7]) is the de-facto communication standard for distributed-memory parallel programming. One particular advantage for parallel programmers is the portability of MPI performance across systems with different underlying network hardware: While HPC hardware vendors and the MPI community spend considerable effort to optimize MPI implementations for the latest HPC network infrastructure, other alternative communication libraries typically do not have optimized support for a wide range of network hardware. With the advent of the remote-memory access (RMA, also referred to as one-sided communication) functionalities in MPI-2 [6] and the significant improvement of the RMA in MPI-3 [7], MPI has become an adequate communication backend for the implementation of partitioned global address space (PGAS) programming models [11].

DASH [4] is a C++ template library which implements a PGAS-like programming model. Unlike other PGAS models, DASH acknowledges the multi-level

© Springer-Verlag Berlin Heidelberg 2015
J.L. Träff et al. (Eds.): Euro-Par 2015, LNCS 9233, pp. 373–384, 2015.
DOI: 10.1007/978-3-662-48096-0_29

hierarchical or compositional nature of today's supercomputing systems, e.g. cores, processors, nodes, racks, islands, full system, and thus does not classify data into remote and local only, but allows for various degrees of remoteness. The template library sits on top of a runtime system (DART), which is responsible for providing services to the DASH library, including the definition of semantics and the abstraction of the underlying hardware. In particular, DART provides functions for the management of teams (a concept similar to MPI communicators), one-sided communications, collective operations, and global memory management.

In an earlier paper [17], we have described DART-MPI, a portable implementation of the DASH runtime, that uses MPI-3 as low-level communication substrate. There, we showed, that the overhead of DART-MPI RMA operations on top of the corresponding MPI-3 operations is negligible in general. Most other PGAS implementations however, do not use MPI as communication substrate; UPC [2] for instance is frequently based on GASNet [1], while GA [9] uses ARMCI [8] as underlying communication substrate.

Originally, all the RMA operations in DART-MPI are substantially mapped directly to the corresponding MPI-3 RMA operations. In particular, DART-MPI invokes MPI RMA operations when source and target of a transfer reside on the same node and share local, physical memory. Alternatively, one could do direct load/store operations without additional copies in the runtime layer. In this paper, the contributions we make on DART-MPI are threefold:

- We utilize the MPI-3 shared-memory extensions to enable direct memory access (memory sharing) for DART-MPI blocking operations for intra-node transfers. However, we turn to the MPI RMA operations when the non-blocking or inter-node data movements happen.
- We redefine the existing translation table to facilitate the reference to the DART-MPI collective global pointer when beginning with the shared memory window in mind.
- Using the low-level and application-level benchmarks, we show the improved performance achieved by embedding the shared-memory-related functionality into DART-MPI.

The rest of the paper is organized as follows: In Sect. 2, we present the background for our work. In Sect. 3, we describe the improved implementation of DART-MPI and evaluate the performance of DART-MPI in Sect. 4. We summarize in Sect. 5.

2 Background

From the perspective of PGAS models, the recent MPI-3 standard [7] significantly improves the one-sided communication system. The relaxation of the RMA semantics, the concretization of the memory consistency model, the introduction of new window types, fine-grained mechanisms for synchronization and

data movement, and atomic operations, make MPI-3 RMA attractive as backend for PGAS implementations. Additionally, the results in Dinan et al. [3] indicate that the new MPI-3 RMA system has performance advantages over the MPI-2 interface. In this section, we briefly explain two new MPI-3 window types: *dynamically-allocated window* and *shared-memory window*, which will play a central role in understanding how to enable memory sharing within a node in DART-MPI. A more detailed description of the other new functionalities can be found in Hoefler et al. [5].

2.1 MPI Dynamically-Allocated Memory Window

A dynamically-allocated window is a new concept in MPI-3 that allows to arbitrarily grow and resize a given window by repeatedly attaching/detaching multiple, non-overlapping, user allocated memory regions to/from the associated window object.

The function *MPI_Win_create_dynamic* is called to generate a window object *d-win* without associating any initial memory block with it. User allocated memory is attached to *d-win*, and thus made available for RMA operations, by invoking the function *MPI_Win_attach*, and detached with *MPI_Win_detach*. Once memory regions are detached from *d-win*, they will not be the target of any MPI RMA operation on *d-win* unless they are re-attached. Notably, any local memory region may be attached and detached repeatedly, and multiple, but non-overlapping memory regions are allowed to be attached to the same window.

MPI_Get_address returns the address of the given memory and should be called to validate the RMA operations on *d-win*. This is due to the fact that the address of the target memory location is passed directly as window displacement parameter to the MPI RMA operations. Therefore, the target process is required to send the address of a certain memory location, that locals to it, to the origin process who inquires for it.

Noticeably, Potluri et al. [13] have published benchmark results which demonstrate that dynamically-allocated windows perform as good as the traditional static MPI-created windows in terms of put latency.

2.2 MPI Shared-Memory Window

The unified memory model, which is fully supported in MPI-3 in order to utilize the cache-coherence characteristics embodied in the modern hardware architectures, is a requirement for exposing the MPI shared-memory window.

To collectively allocate the shared memory region across all processes in a given communicator, MPI-3 defines a portable, shared-memory window allocation interface – *MPI_Win_allocate_shared* to generate a shared-memory allocated window object *shmem-win*. In addition, the communicator that the *shmem-win* associates with should be a *shared-memory capability* communicator, which means it is allowed to build a memory sharing region on top of this communicator. Therefore, the additional function *MPI_Comm_split_type*, as an extension of

the function *MPI_Comm_split*, identifies sub-communicators on which the shared memory region can be created with the type of *MPI_COMM_TYPE_SHARED*. The function *MPI_Win_shared_query* is provided to query the base pointer to the memory on the target process. Coupled with the *shmem-win*, the locally-allocated memory can even be accessed by the MPI processes in the group of *shmem-win* with immediate load/store operations. Such access pattern can make data movements bypass the MPI layer and directly go through memory sharing, which brings in significant performance improvement.

3 The DART-MPI Implementation Design

In this section, we explain the approach of enabling the memory sharing option for the blocking RMA operations in DART-MPI and address the modifications and improvements that are made with respect to the existing DART-MPI.

There are two types of DART global memory, collective and non-collective [17]. The collective global memory, pointed to by a collective global pointer, is created and distributed across the given team. The non-collective global memory, pointed to by a non-collective global pointer, is only allocated in the global address space of the calling unit. We assume that all the following collective global memory blocks are allocated across team T consisting of P units.

3.1 Communication Hierarchy of the DART-MPI Blocking RMA Operations

To make the DART-MPI intra-node communication more efficient, we alter the existing implementation to let the DART-MPI blocking operations deal with the data locality explicitly. Note, that the DART-MPI non-blocking RMA interfaces do not yet support the memory sharing as described earlier in this paper.

In the team creation code, the team T is split into sub-teams on which it is possible to enable communication via sharing memory. We accomplish this by calling *MPI_Comm_split_type* with key *MPI_COMM_TYPE_SHARED*. In addition, a *d-win* is generated without any memory attached when team T is created, indicating one-to-one relationship is built between *d-win* and T. Such relationship is stored in an array named *dart_win_lists*. Therefore, the position of the team T in *teamlist* [17] can also be a perfect index into the array *dart_win_lists*. The *d-win* can potentially be utilized to complete all the data movements where the units are located in different sub-teams.

In the collective global memory allocation code, instead of allocating a block of memory from a memory pool that is reserved for T, we need to create a *shmem-win* spanning the memory of the specified size on each sub-team mentioned above. On top of that, each unit of the team T should attach the locally-allocated memory to the *d-win* explicitly to make them available for the units in the varying sub-teams. As the Fig. 1 shows, there are two overlapping windows sharing the same memory region for different purposes. On the one hand,

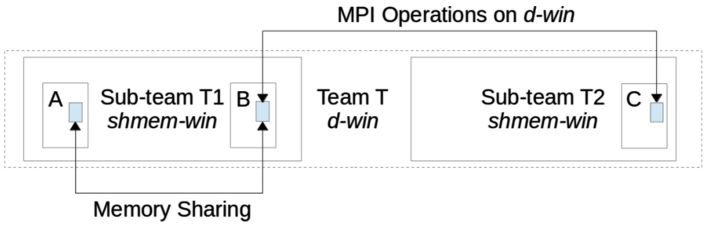


Fig. 1. Nesting of shared-memory window inside RMA window for blocking put/get operations

the units covered by the same *shmem-win* can communicate with each other via memory sharing (e.g., memcpy). On the other hand, the units located in different sub-teams should turn to the *d-win* for completing the remote accesses with MPI RMA operations. Note, that using shared-memory in the DART-MPI non-blocking RMA operations is anything but trivial as for instance the direct memcpy function itself is a blocking operation. Furthermore, the introduction of DMA copy engine could be a workaround to support asynchronous memory copying [16] for DART-MPI.

In the non-collective global memory allocation code, we provide two overlapping global windows, which indicates that all the DART-MPI non-collective allocations fall into two pre-defined global windows. One of the two windows is generated first spanning a large amount of shared memory region on the default communicator – *MPI_COMM_WORLD* [7] for intra-node communications, the other is then created with *MPI_Win_create* covering the above shared memory region to enable the message transferring across different nodes. As a result, these two windows share the same static shared memory region, and independently implement the data movements on them in an efficient manner.

3.2 DART-MPI Collective Global Pointer Dereference

In this section, we mainly explain the collective global pointer dereference of the updated DART-MPI since the non-collective global pointer basically continues to use the original dereference mechanism.

Besides the altered communication pattern, the meaning of the member *segid* in the global pointer is also re-specified for management convenience and data access efficiency. Therefore, the *segid* in the collective global pointer is no longer set to the related team ID but rather an increasing positive integer number, which can be used to determine any collective global block uniquely.

With the aid of the translation table [17], collective global pointer can get analyzed adequately. Thus it is critical for us to understand how the translation table reacts to the hierarchical communication pattern and the modification made in the global pointer, which also has an impact on the original collective global pointer deference method.

To be consistent with the modified definition of *segid* in global pointer, the key in the translation table is altered and the *segid* is utilized instead. The

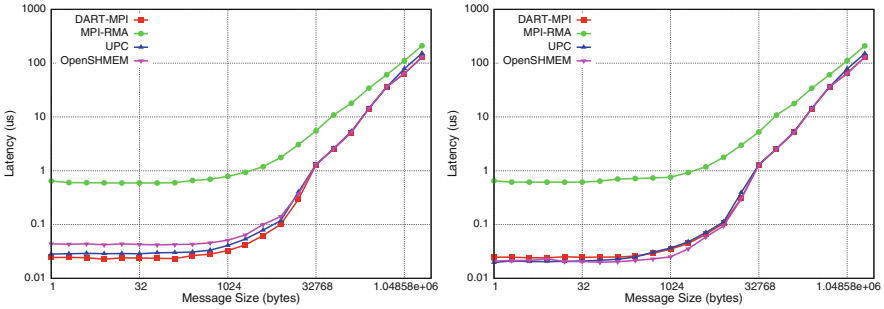
translation table is arranged in an ascending order based on the key *segid*. As a result, we do not need to bind a separate translation table to each team, instead a single translation table is active during the lifetime of a DART-MPI program. Once a block of collective global memory is created, a unique *segid* and the related *shmem-win* are generated and then added to the translation table together, signifying the one-one relationship between collective global pointer and the related *shmem-win*. In addition, according to the Sect. 2, we learn that after attaching the shared memory region onto the *d-win* locally, the routine *MPI_Get_address* should be invoked so as to collect the beginning address of the local shared memory region of each unit in team T. Thus, the translation table should also contain an array *disp-set* storing those separate addresses. As an example, when unit i in the team T is targeted, then the ith item in the related *disp-set* should be obtained and be utilized in the future to locate the target memory location in unit i. The offset returned in the generated collective global pointer is initialized to 0.

The location of target data is given by DART global pointer, which incorporates the information on the target unit, *segid* and a specific offset. For the collective global pointer, in the case of intra-node communications, we firstly query the appropriate *shmem-win* that covers the expected target location from the translation table according to the *segid*, then decode the location with offset. In the case of inter-node communications, we firstly query the *disp-set*, indicates the beginning address of the window segment of each unit in team T, from the translation table according to the *segid*, and then get the correct *d-win* from the array *dart_win_lists* and translate the absolute unit id to the relative unit id i in T, and finally access the remote data through MPI RMA operations, where the value of *offset*+*disp-set*[i] is passed as parameter *target_disp*.

4 Performance Evaluation

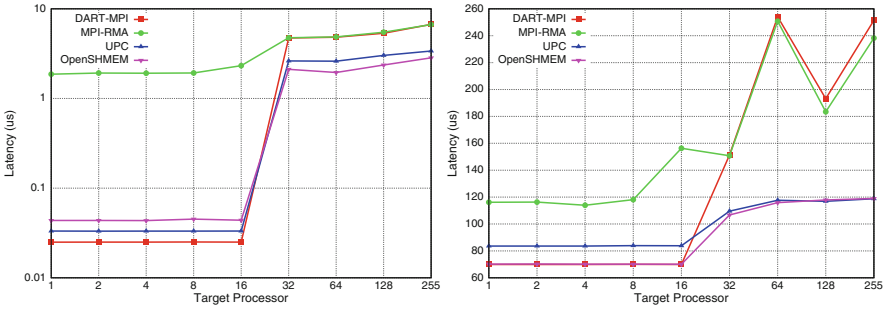
In the following, we evaluate the performance of DART-MPI using a set of benchmarks which includes low-level communication and application benchmarks. All the benchmarks are carried out on a Cray XC40 system named Hornet. Each compute node features two Intel Haswell E5-2680v3 2.5 GHZ processors and consists of 24 cores. The different compute nodes are interconnected through a Cray Aries network using Dragonfly topology. They use the Cray-MPI implementation of MPI-3.

Foremost, we are interested in the evaluation of the performance advantage of our DART-MPI, using MPI-3 shared-memory and RMA, over native MPI-3 RMA. As shown in a previous paper [17], the difference in performance of DART-MPI and MPI-3 RMA operations for non-local transfers is negligible. In that sense, MPI-3 can be seen as a proxy for the old DART-MPI. We will thus not show the latter explicitly in this paper. In addition, we compare DART-MPI with two important PGAS implementations: UPC and OpenSHMEM, which are both fully implemented and tuned on the Cray XC40 system. In all cases we use the Cray compiler, which also supports UPC (through the compiler flag *-h upc*)



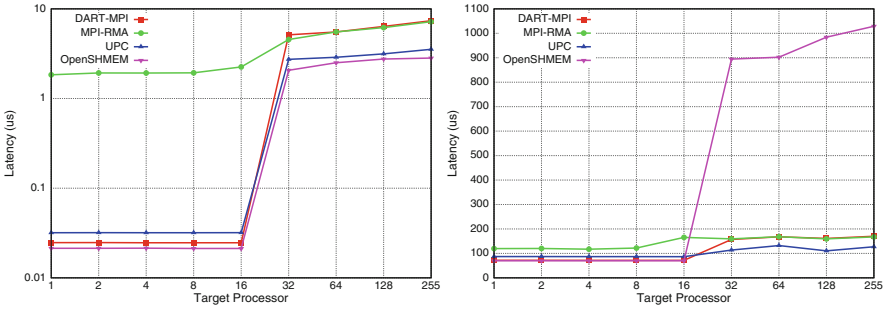
(a) Blocking put (intra-node) (b) Blocking get (intra-node)

Fig. 2. Blocking put/get latency on 2 ranks/units



(a) Blocking put (8 bytes) (b) Blocking put (1 Mb)

Fig. 3. Blocking put latency as a function of logically increasing distance between two involved ranks/units on 256 PEs



(a) Blocking get (8 bytes) (b) Blocking get (1 Mb)

Fig. 4. Blocking get latency as a function of logically increasing distance between two involved ranks/units on 256 PEs

and OpenSHMEM (as a library). All low-level communication benchmarks are averaged over 10000 executions. We do not show the error bars in the following figures, as these are always small and would only confuse the plots.

4.1 Low-Level Communication Benchmarks

In this section, we assess the raw communication performance based on the OSU Micro Benchmark [10]. Firstly, we test the average latencies of the blocking operations of DART-MPI as well as the counterparts of MPI (with passive target communication calls), UPC and OpenSHMEM [12] only in the case of intra-node (communication within one node). Secondly, we evaluate how the blocking put and get operations perform when increasing logical distance between two involved processes for DART-MPI, MPI, UPC and OpenSHMEM.

Figure 2 shows the average latency of intra-node blocking put and get operations for message size ranging from 2^0 to 2^{21}. In all cases the latency roughly keeps constant for small messages (here < 1024 byte). Beyond that the completion time is dominated by the actual message transfer time and basically grows linearly with the message size as expected. Noticeably, the curves for UPC, OpenSHMEM and DART-MPI are very close to each other. For small messages, native MPI performs more than 10 times slower than the other three models. This fully illustrates that the overhead of MPI one-sided operations is relatively high compared to that of direct load/store operations when data movements happen within one node.

A careful comparison shows, that DART-MPI always performs better for blocking put operations than UPC (by about 20 %) and OpenSHMEM (by about 40 %), although such advantage becomes negligible as the message size increases. For blocking get operations, the variance between them is much lower in absolute terms, but the trend of curves seems to suggest that DART-MPI (and to a lesser extend, also UPC) performs slightly slower than OpenSHMEM.

Next, we evaluate the performance of the blocking RMA operations as a function of logical distance between source and target. We send messages of fixed size from process 0 to target processes varying from 1 to 255. Note that the job consists of 256 ranks/units in total, which corresponds to 11 nodes on Hornet. Figures 3 and 4 show the performance of blocking put and get operations, respectively, for the short message size of 8 bytes and the long message size of 1Mb as a function of logically increasing distance between the origin and target.

As expected the latency remains constant for message transfers within one node. However, at a logical distance between 16 and 32, i.e., when leaving one node and targeting the second one, the latency goes up significantly in all cases except for native MPI, as the overhead of native MPI intra-node data transfers is relatively high to begin with (as reported above). The curves for DART-MPI nearly sit above those for native MPI for inter-node data transfers. This is expected since DART-MPI falls back on MPI when communicating across nodes. Both, however, perform in general slightly worse than UPC and OpenSHMEM in latency for inter-node communications. An exception occurs when executing the

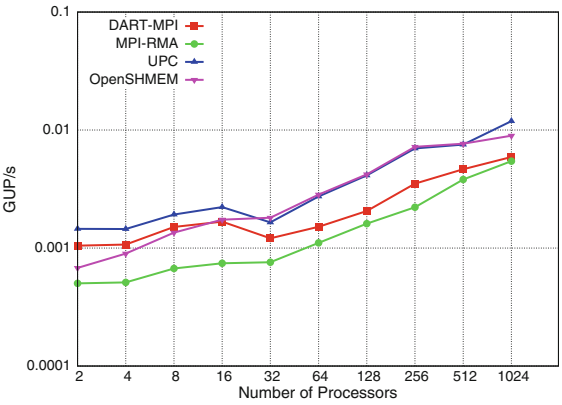
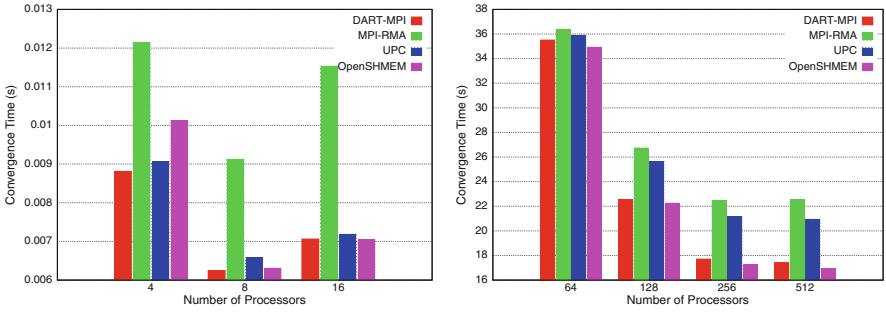


Fig. 5. Random Access performance comparison



(a) 64×64 (b) 1024×1024

Fig. 6. Five-point stencil performance comparison

OpenSHMEM blocking get operation when transferring large messages; it performs 5 to 10 times worse than the other three models in latency. We do not have an explanation for such behavior, but the full data set we have seems to suggest that OpenSHMEM blocking get operations show relatively poor performance for large messages.

4.2 Application Benchmarks

In this section we present the results of two application benchmarks, namely Random Access and a stencil code kernel. All benchmarks were run on up to 1024 cores, i.e., 45 nodes on Hornet.

Random Access: The Random Access (RA) benchmark [14] is one of the HPC Challenge benchmarks developed for the HPCS program. It consists of concurrent, atomic updates of random elements of a distributed array by all ranks [15].

The general performance metric is giga-updates per second (GUPs). The messages involved are very small, i.e., 8 bytes. Figure 5 shows the performance in terms of GUPs for the DART-MPI, native MPI, UPC and OpenSHMEM versions of the RA benchmark for the number of processes varying from 2 to 1024. Interestingly, DART-MPI, UPC and OpenSHMEM achieve similar performance. The performance of the native MPI version is relatively poor in all cases, and the performance of the DART-MPI version suffers at large number of ranks due to the underlying MPI.

The relative performance evaluation of the DART-MPI, UPC and OpenSHMEM versions is complex stemming from the fact, that the blocking get, put and atomic operations are involved. DART-MPI performs slightly better than OpenSHMEM when RA runs on a single node. However, we can see there is a clear gap between the performance of DART-MPI and that of UPC and OpenSHMEM when the application is carried out across nodes. This is due to the fact that the amount of the inter-node remote accesses increases as the growing of the running nodes. The inter-node communication time performance of DART-MPI is poor relative to that of UPC and OpenSHMEM for smaller messages (e.g., 8 bytes), as obvious from Figs. 3 and 4. In addition, the atomic operation contributes partly to such performance gap between the DART-MPI, UPC and OpenSHMEM versions, respectively. Noticeably, although the increase in the number of inter-node remote accesses exacerbates the performance of DART-MPI, DART-MPI can still perform better than native MPI, which has to do with the fraction of memory sharing for the intra-node data movements.

Five-Point 2D Stencil Computation: This kernel computes the 2D Poisson equation by applying a five-point stencil on a square grid, and solving in an iterative way with the Gauss-Seidel method. The grid of $N \times N$ elements is decomposed evenly by rows among *numprocs* distributed processes. Each element holds a 4-byte floating point number. The kernel uses extra halo zones to exchange boundary elements between neighbors, A total of $4 \times N \times (2 \times numprocs - 2)$ bytes of data per iteration is transmitted using blocking put operations. With those halo data, all the inner grid cells can get updated successfully. We run the stencil kernel until convergence of solution. The time recorded in the benchmark includes the execution time of the Gauss-Seidel solver (local computation part) and communication time for halo exchange.

We run the five-point stencil benchmark for DART-MPI, MPI, UPC and OpenSHMEM versions on the grids of 64×64 elements and 1024×1024 elements respectively. Figure 6(a) shows comparison results of a 64×64 grid distributed across 4, 8 and 16 processes on a single node. We can see that the DART-MPI version always performs slightly better than the UPC version, when all the data movement happen within a single node. In addition, DART-MPI, UPC and OpenSHMEM outperform native MPI by $\sim 35\,\%$ for 16 processes.

The performance of the DART-MPI version degrades when there are data movements across nodes. Figure 6(b) shows benchmark results of a 1024×1024 grid for 64, 128, 256 and 512 processes. The convergence time of the DART-MPI and OpenSHMEM versions decreases as the number of processes involved is

increased, which suggests that DART-MPI and OpenSHMEM are more scalable than native MPI and UPC from the perspective of this benchmark.

5 Conclusions

DART-MPI is the runtime system for the PGAS-like C++ template library DASH and built on top of MPI-3 one-sided communication primitives. In this paper we present an optimized design of DART-MPI which uses the new MPI-3 shared-memory extension for intra-node communications. In essence, we nest MPI-3 shared-memory windows inside RMA windows to do direct load/store operations for intra-node transfers, and MPI-3 one-sided communication operations on the RMA windows for inter-node transfers.

We expect that this optimization will improve the performance of DART-MPI for intra-node communication. To verify this claim, we run three classes of benchmarks, namely low-level put/get benchmarks, a Random Access benchmark and a stencil application kernel on the Cray XC40 system. We evaluate the performance of DART-MPI against that of native MPI. In addition, we compare DART-MPI to OpenSHMEM and UPC as two other PGAS-like programming models. The results of the evaluation demonstrate, first, that DART-MPI performs significantly faster than MPI RMA when messages are transmitted within a single node, i.e., that our optimization of DART-MPI leads to a better intra-node communication performance, second, that the comparison to OpenSHMEM and UPC show that the performance improvement that is brought by our optimization makes DART-MPI comparable with UPC and OpenSHMEM. Additionally, our performance evaluation also shows, that for some relevant operations – especially the inter-node RMA operations – DART-MPI still performs slower than the alternative PGAS approaches.

In this paper, we have only considered blocking RMA put and get operations. The current design of DART does not include an asynchronous progress engine, and therefore relies on other parts of the software stack to do progress as necessary for non-blocking operations. In particular, we rely on MPI for non-blocking RMA operations and thus see no benefit for non-blocking DART operations. An asynchronous progress engine which allows optimization of non-blocking intra-node transfers is a subject of further research.

Acknowledgments. The authors would like to thank Karl Fürlinger for fruitful discussion on the DASH runtime design. We gratefully acknowledge funding by the German Research Foundation (DFG) through the German Priority Programme 1648 Software for Exascale Computing (SPPEXA).

References

1. Bonachea, D., Jeong, J.: GASNet: A Portable High-Performance Communication Layer for Global Address-Space Languages. Technical report, CS258 Parallel Computer Architecture Project (2002)

2. Carlson, W., Draper, J., Culler, D., Yelick, K., Brooks, E., Warren., K.: Introduction to UPC and Language Specification. Technical report CCS-TR-99-157, IDA Center for Computing Sciences (1999)
3. Dinan, J., Balaji, P., Buntinas, D., Goodell, D., Gropp, W., Thakur, R.: An implementation and evaluation of the MPI 3.0 one-sided communication interface. In: Preprint ANL/MCS-P4014-0113. IEEE Computer Society (2013)
4. Fürlinger, K., et al.: DASH: data structures and algorithms with support for hierarchical locality. In: Lopes, L., et al. (eds.) Euro-Par 2014, Part II. LNCS, vol. 8806, pp. 542–552. Springer, Heidelberg (2014). http://dx.doi.org/10.1007/978-3-319-14313-2_46; http://dblp.uni-trier.de/rec/bib/conf/europar/FurlingerGGKTHIMMZ14
5. Hoefler, T., Dinan, J., Thakur, R., Barrett, B., Balaji, P., Gropp, W., Underwood, K.: Remote Memory Access Programming in MPI-3. Technical report, Argonne National Laboratory (2013)
6. MPI Forum: MPI: A Message-Passing Interface Standard. Version 2.2 (September 4th 2009), December 2009. http://www.mpi-forum.org
7. MPI Forum: MPI: A Message-Passing Interface Standard. Version 3.0 (September 21st 2012), September 2012. http://www.mpi-forum.org
8. Nieplocha, J., Carpenter, B.: ARMCI: A portable remote memory copy library for distributed array libraries and compiler run-time systems. Technical report (1999)
9. Nieplocha, J., Harrison, R.J., Littleeld, R.J.: Global arrays: a nonuniform memory access programming model for high-performance computers. J. Supercomputing **10**, 169–189 (1996)
10. OSU Micro-Benchmarks (2014).http://mvapich.cse.ohio-state.edu/benchmarks/
11. Partitioned Global Address Space (2014). http://www.pgas.org/
12. Poole, S., Hernandez, O., Kuehn, J., Shipman, G., Curtis, A., Feind, K.: OpenSHMEM - toward a unified RMA model. In: Padua, D. (ed.) Encyclopedia of Parallel Computing, pp. 1379–1391. Springer, US (2011)
13. Potluri, S., Sur, S., Bureddy, D., Panda, D.K.: Design and implementation of key proposed MPI-3 one-sided communication semantics on InfiniBand. In: Cotronis, Y., Danalis, A., Nikolopoulos, D.S., Dongarra, J. (eds.) EuroMPI 2011. LNCS, vol. 6960, pp. 321–324. Springer, Heidelberg (2011). http://dblp.uni-trier.de/db/conf/pvm/eurompi2011.html#PotluriSBP11; http://dx.doi.org/10.1007/978-3-642-24449-0_38; http://www.bibsonomy.org/bibtex/2b8d79bab12610e243ae48eceb2905b91/dblp
14. RandomAccess GUPS (Giga Updates Per Second) (2013). http://icl.cs.utk.edu/projectsfiles/hpcc/RandomAccess/
15. Shamis, P., Venkata, M.G., Poole, S., Welch, A., Curtis, T.: Designing a high performance OpenSHMEM implementation using universal common communication substrate as a communication middleware. In: Poole, S., Hernandez, O., Shamis, P. (eds.) OpenSHMEM 2014. LNCS, vol. 8356, pp. 1–13. Springer, Heidelberg (2014)
16. Vaidyanathan, K., Chai, L., Huang, W., Panda, D.K.: Efficient asynchronous memory copy operations on multi-core systems and I/OAT. In: IEEE International Conference on Cluster Computing (2007)
17. Zhou, H., Mhedheb, Y., Idrees, K., Glass, C.W., Gracia, J., Fürlinger, K., Tao, J.: DART-MPI: an MPI-based implementation of a PGAS runtime system. In: PGAS 2014, 06–10 October 2014. http://dx.doi.org/10.1145/2676870.2676875