Recent Experiences in Using MPI-3 RMA in the DASH PGAS Runtime

Joseph Schuchart
schuchart@hlrs.de
High Performance Computing Center Stuttgart (HLRS)
University of Stuttgart
Germany

Roger Kowalewski
Karl Fuerlinger
roger.kowalewski@nm.ifi.lmu.de
karl.fuerlinger@nm.ifi.lmu.de
Computer Science Department, MNM Team
Ludwig-Maximilians-Universität (LMU) Munich
Germany

ABSTRACT

The Partitioned Global Address Space (PGAS) programming model has become a viable alternative to traditional message passing using MPI. The DASH project provides a PGAS abstraction entirely based on C++11. The underlying DASH RunTime, DART, provides communication and management functionality transparently to the user. In order to facilitate incremental transitions of existing MPI-parallel codes, the development of DART has focused on creating a PGAS runtime based on the MPI-3 RMA standard. From an MPI-RMA user perspective, this paper outlines our recent experiences in the development of DART and presents insights into issues that we faced and how we attempted to solve them, including issues surrounding memory allocation and memory consistency as well as communication latencies. We implemented a set of benchmarks for global memory allocation latency in the framework of the OSU micro-benchmark suite and present results for allocation and communication latency measurements of different global memory allocation strategies under three different MPI implementations.

KEYWORDS

Partitioned Global Address Space, PGAS, MPI-RMA, communication latency, global memory allocation, DASH

HPC Asia 2018 WS, January 31, 2018, Chiyoda, Tokyo, Japan

ACM ISBN 978-1-4503-6347-1/18/01…$15.00
https://doi.org/10.1145/3176364.3176367

ACM Reference Format:
Joseph Schuchart, Roger Kowalewski, and Karl Fuerlinger. 2018. Recent Experiences in Using MPI-3 RMA in the DASH PGAS Runtime. In *HPC Asia 2018 WS: Workshops of HPC Asia 2018, January 31, 2018, Chiyoda, Tokyo, Japan.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3176364.3176367

1 INTRODUCTION

The PGAS (Partitioned Global Address Space) programming model is widely considered a promising approach for programming current and future large-scale systems [1, 12, 25]. It lends itself well for the development of unstructured and irregular applications and exposes a high potential for overlapping communication and computation to hide communication latencies by decoupling communication and process synchronization [2].

The PGAS approach relies on a one-sided communication model, where the target of a communication operation is not actively involved in the data transfer. The hardware realization of this one-sided communication model is available in the form of Remote Direct Memory Access (RDMA), which most modern interconnect networks support. Several PGAS runtime systems exploit the capabilities of RDMA-enabled networks, including GASnet [2], ARMCI [19], GASPI [14], and OpenShmem [5]. However, these libraries are often not part of the standard software stack of HPC systems and can be tricky to install and tune for individual users.

In contrast, MPI is ubiquitous as the de-facto standard in HPC and thus an MPI library installation is available on almost all HPC platforms. With version 3.0 of the MPI standard, a well-designed interface for one-sided communication became available in the form of MPI-3 RMA [9, 16] which addresses conceptual limitations of the earlier MPI-2 standard [3]. Moreover, we expect that MPI implementations will be at the forefront in supporting next-generation large-scale systems, including future Exa-scale architectures.

DASH makes extensive use of C++11 features to abstract some of the complexities of this PGAS model while providing an incremental path for migration of traditional C++ MPI

applications. As a new project, without the shackles of legacy code, we decided to investigate the possibility of using MPI-3 RMA features as a basis for the runtime system of DASH, called the DASH RunTime (DART). The main benefits we hoped for by following this approach were to enable an easier integration with existing (MPI) applications, easing the installation burden on users, and benefiting from as well as participating in the active development happening in the MPI community.

Previous work has presented the foundations and preliminary performance evaluations of this approach [27, 28]. The main contribution of this work is to provide an update on recent experiences in implementing the DART runtime based on MPI and to point out some of the challenges we encountered. We also contribute measurements for communication and global memory allocation latencies that will influence future design decisions in the development of DART.

The remainder of this paper is structured as follows: Section 2 provides a short overview on DASH and DART followed by an outline of the implementation of DART and the used MPI features in Section 3. Section 4 provides a performance evaluation of DART and MPI features using benchmarks. A discussion of related work can be found in Section 5 followed by our conclusions in Section 6.

2 INTRODUCTION TO DASH/DART

DASH aims at providing distributed data-structures and parallel algorithms operating on them. The DASH library follows the design principles of the C++ STL (standard template library) to offer users a familiar interface and ensure compatibility with existing algorithms, thus helping in the parallelization of existing C++ codes.

At the heart of the DASH template library is a set of distributed data-structures, including one- and multidimensional arrays as well as lists and unordered maps. The distribution of the data among the participating processes (called *units* in DASH) can be controlled by user-defined *patterns* [10].

Elements in the distributed data structures are accessed through iterators and overloaded C++ operators, hiding the underlying put and get semantics in traditional value assignment semantics. To facilitate efficient local data access and thus avoid overhead, all DASH containers provide access to a unit's local elements through local iterators or raw pointers.

All communication operations in DASH are performed through the DART library, which aims at providing a thin abstraction of the underlying communication backend [28]. Using DART, global shared memory can be allocated as segments involving either all units of a team (collective allocation) or a single unit (local allocation). In contrast to MPI, DART has no notion of access epoch. Instead, read and write accesses can be performed between allocation and release of a global memory segment at any time by any unit that participated in the collective allocation or any unit in the case of local allocation.

Global shared memory is addressed byte-wise through DART global pointers, which are 128-bit wide descriptors containing the team and segment IDs of the allocation as well as the target unit ID and a 64-bit wide byte offset at the target. Global pointers are globally unique and can thus be copied between units in the same team to simplify the construction of data structures like global linked lists. Local allocations can be shared with any unit as they are not bound to a specific team.

DART offers several communication operations, including put, get, and atomic operations (element-wise `accumulate`, `fetch_and_op`, `compare_and_swap`), which initiate the transfer and require a `flush` operation for local or remote completion. In addition, DART also offers blocking variants of put and get, which guarantee local or remote completion of transfers and are mainly meant for single-element accesses in DASH containers. While blocking access is the default in DASH, non-blocking element-wise operations can be performed by using the `async` accessor proxy object of a container.

An example of the interactions between DASH, DART, and MPI is provided in Figure 1. Here, a fixed-size array is created and filled by a single unit using asynchronous write operations, which only require local but not remote completion to ensure that the iteration variable can be reused. It should be noted, that this is not the most efficient way: a call to `dash::generate_with_index()` or `dash::for_each()` on all units would have each unit to fill their local portion of memory and would not require any communication. Remote completion will be guaranteed after a call to `array.flush()` or (as displayed in the example) a call to `array.barrier()`, which combines a call to `array.flush()` with a barrier on the team used to construct the array (the global team by default).

In addition to the communication operations outlined above, DART also provides blocking collective reduction and synchronization operations, hardware locality information, team management functionality, and a distributed locking mechanism to allow for user-controlled mutual exclusion of conflicting accesses to global shared memory.

DASH distributed data structures can seamlessly be integrated into existing MPI applications, giving users the opportunity for an incremental transition. Although DART only supports basic data types, any `trivially_copyable` C++ type can be used in DASH containers, which will trigger byte-wise transfers in DART for any types other than basic datatypes.

3 MPI FEATURES AND CHALLENGES

DASH and DART rely on a range of features to be provided by the underlying communication back-end. This section

```
dash::Array<int> array(N);
// initialize array
// better: dash::generate()
if (dash::myid() == 0) {
  for (int i = 0; i < N; ++i) {
    array.async[i] = i;
  }
}
array.barrier();

if (dash::myid() == 1)
  print(array[0]);
```

(a) DASH code

```
dart_team_memalloc_aligned();

dart_put_blocking_local();

dart_flush_all();
dart_barrier();

dart_get_blocking();
```

(b) DART routines

```
MPI_Win_create_shared();
MPI_Win_attach();
MPI_Allgather();

MPI_Rput();
MPI_Wait();

MPI_Win_flush_all();
MPI_Barrier();

MPI_Rget();
MPI_Wait();
```

(c) MPI routines

Figure 1: DASH operations and the corresponding DART and MPI library routine calls.

provides an overview of the functionality offered by DART and the corresponding MPI operations. Moreover, this section outlines some of the challenges we have been facing in designing and implementing DART based on MPI-3 RMA.

3.1 Process groups

DASH supports the concept of hierarchical process groups, called *teams*, which are created by splitting existing teams (starting from a single global team), either based on the number of sub-groups required or based on specific locality information such as nodes or NUMA domains. The team management functionality is part of DART, which leverages the flexibility of MPI groups and communicators. A group of units in DART is mapped to MPI groups and used to construct a team in DART, which directly maps to an MPI communicator. The convenience and flexibility of the MPI process group management functionality has made it easy for us to implement the hierarchical team approach in DASH and DART.

3.2 Memory allocation

DASH and DART rely on the underlying communication backend to allocate memory in the global address space. MPI offers several ways for exposing local memory to other MPI processes to form the global address space, e.g., by allocating and exposing memory in a single function call (using `MPI_Win_allocate`) or by exposing pre-allocated memory (using `MPI_Win_create`).

In addition, MPI also offers so-called dynamic windows, which can be created using `MPI_Win_create_dynamic` and allow attaching pre-allocated local memory on the fly using `MPI_Win_attach`. In contrast to the non-dynamic windows, the displacements in dynamic windows are relative to the beginning of the local address space of the processes.

Moreover, MPI supports so-called shared memory windows, which can be created across processes residing in the same shared memory domain using `MPI_Win_allocate_shared` and allow direct access to remote memory, e.g., using `memcpy`. DART uses shared memory windows to optimize intra-node `put` and `get` operations without going through the MPI library, potentially reducing transfer overheads [27]. To do so, DART allocates a single dynamic window per team and allocates shared windows for each segment, whose memory is then attached to the dynamic window. Thus, memory allocation only consists of creating a shared window, which involves the subsets of processes running on the same nodes, and globally communicating the displacements across the team using a single team-wide `MPI_Allgather`. The details of this shared memory optimization are outlined in [27].

The per-team dynamic window is locked for all processes in the allocating team, thus relying on the *passive target mode* of MPI. We expected this allocation scheme to yield improved scalability over regular window allocation across the full team. Section 4 will present measurements comparing the different allocation strategies.

Unfortunately, so far not all MPI implementations provide full support for shared memory windows, requiring DART to maintain separate code paths for different MPI implementations and potentially specific library versions. In that case, DART will still use dynamic windows but cannot shortcut node-local communication through `memcpy`. However, there is no standardized way for automatically determining the used MPI implementation at compile time, thus leaving the burden of disabling unsupported features on the user.

In its current implementation, global memory allocation may become a performance-critical operation as some DASH parallel algorithms rely on temporary allocations to perform reductions on intermediate results of derived data types, e.g., finding the position of the minimum element in an array of comparable derived datatypes. DART thus has to provide a lightweight allocation process, which has motivated our use of dynamic MPI windows.

3.2.1 Local memory alignment. Special care should be given to the local alignment of global memory allocated through MPI. The MPI standard does not mandate any alignment requirements for memory allocated through MPI library routines. While the standard recommends allocating memory through MPI library routines, we have observed only natural alignment to be guaranteed, i.e., alignment suitable for the largest native data type (8 byte on 64-bit machines). Users requiring stricter alignment guarantees, e.g., to fully exploit the potential of vectorization, are left on their own to guarantee proper alignment. As stated above, DART allocates global memory using `MPI_Win_allocate_shared` and thus cannot easily use system routines such as `posix_memalign` for allocation of aligned memory. We plan to add support for user-defined alignments in a future version of DART.

3.2.2 System shared memory allocation. MPI implementations rely on system shared memory to optimize intra-node communication even on regular windows, i.e., on each node a shared memory segment is allocated during a call to `MPI_Win_alloc` to which all node-local processes attach. The GNU/Linux operating system allows to share memory between processes running on the same node by mapping a shared memory object into each process's address space using `mmap(2)` with the flag `MAP_SHARED`. The shared memory object can be allocated through POSIX shared memory (`shm_open(3)`) or an arbitrary file created by `open(2)`. Alternatively, SysV shared memory can be used to allocate a shared memory object (`shmget(2)`) and attach it to the process's address space (`shmat(2)`).

The POSIX standard does not mandate how shared memory objects allocated through `shm_open` should be handled internally. On GNU/Linux, a call to `shm_open` commonly creates a file in a dedicated filesystem (commonly `tmpfs` mounted under `/dev/shm`). The size of the shared memory object can be configured using `ftruncate(2)`, which will grow (or shrink) the file. The file is immediately unlinked and subsequently deleted as soon as all references to it have disappeared.

It is left to the system administrator to configure the size of the filesystem mounted at `/dev/shm`, which effectively limits the total size of shared memory allocated through POSIX shm. Since `tmpfs` supports sparse files, `ftruncate` allows growing files beyond the size limits of the underlying filesystem and memory pages will only be physically allocated as soon as the pages are accessed, i.e., as soon as the application accesses the shared memory segment. Oversubscribing the filesystem leads to the application being terminated by a `SIGBUS` without the cause being immediately visible to the user.

Unfortunately, the POSIX shm interface does not provide means of portably checking the limits of shared memory allocations and users commonly cannot influence the size of the underlying `tmpfs` filesystem on GNU/Linux. In contrast to that, the SysV interface under GNU/Linux has clearly defined limits that can be queried through entries in `/proc/sys/kernel/`, which can be used to catch (at least some) cases of shared memory oversubscription. However, in case of unexpected failures, SysV shared memory objects may persist beyond the life-time of the application, effectively using up memory that can only be detected using the `ipcs` command line tool.

At the time of this writing, Open MPI allocates shared memory by mapping a file created under `/tmp` (or a path provided by the user) after checking for sufficiently available space. MPICH, on the other hand, relies on POSIX shm by default with a compile-time option for SysV shm. In both cases, we have experienced problems on systems with limited `/tmp` and `/dev/shm` filesystems, effectively limiting the amount of memory that can be allocated in DASH distributed data structures and – in the case of POSIX shm – leading to hard-to-debug crashes reported by users. It is thus imperative that HPC system administrators are aware of the caveats of shared memory and provide sufficent resources for both `/tmp` and `/dev/shm` filesystems, ideally allowing users to allocate (nearly) all memory available on the node.

3.3 Data transfer and memory consistency

The MPI implementation of DART relies heavily on communication operations provided by MPI, with the notable exception of intra-node communication described above. MPI offers two basic communication operations, `MPI_Put` to write data to the target and `MPI_Get` to read from it. Both operations may be non-blocking and their completion is only guaranteed after a call to one of the `MPI_Win_flush` functions, which distinguish between local completion (the local memory can be reused) and remote completion (the data has been written to the remote memory).

Based on these MPI primitives, DART offers a set of put and get operations. The basic operations `dart_get` and `dart_put` provide the same semantics as their MPI counter-parts. The operations are guaranteed to be complete after a call to the corresponding DART flush operation, i.e., `dart_flush` for remote and `dart_flush_local` for local completion.

To ensure compatibility of DASH data structures with serial STL algorithms, the default access mode for elements

in DASH containers has to be blocking read-access and immediate remote completion on write access to ensure that written values are visible to subsequent reads. To avoid the overhead of two separate DART calls for `put`/`get` and `flush`, DART offers blocking variants such as `dart_get_blocking` and `dart_put_blocking`. The former combines `MPI_Rget` and `MPI_Wait` to wait for the completion of the transfer. Unfortunately, ensuring remote completion of an `MPI_Put` operation in `dart_put_blocking` requires an MPI flush call since waiting for a request returned from `MPI_Rput` only ensure local completion. This makes it effectively impossible to mix blocking `put` and other non-blocking data transfers on the same window, as the `flush` operation completes all previously initiated non-blocking operations.

Similar to MPI, DART communication calls accept the type and number of data elements in the buffer. In contrast to MPI, the parameter describing the number of elements N is of type `size_t` instead of `int`, allowing for transfers of data beyond 2 GB in a single call (on 64-bit machines). Internally, such large transfes are split into two operations: the transfer of $\lfloor N/2^{31} \rfloor$ blocks of size 2^{31} elements copied using pre-allocated MPI types followed by the transfer of the remaining elements.

Since DASH relies heavily on templates and compile-time optimizations using `constexpr`, the basic DART-types are constants to which C++ integral types are mapped at compile-time. At the same time, DART also supports the creation of strided and indexed types for efficient non-contiguous data access, which are dynamically created and mapped directly to MPI types. Hence, types in DASH are expressed as an integral value that can store both constant values as well as opaque pointers to dynamically created types.

3.4 Thread-safety

One of the design goals of DASH has been thread-safety, which allows DASH applications to leverage the performance and productivity of thread-based parallelization techniques such as OpenMP. Thread-support in DASH and DART can be configured at compile-time. If enabled, DART initializes the underlying MPI runtime using `MPI_Init_thread` and determines the thread-support level provided. The four thread-support levels defined by the MPI standard are mapped to either `DART_THREAD_SERIALIZED` or `DART_THREAD_CONCURRENT`, leaving out some of the semantic complexities of MPI.

The runtime guarantees that access to DART functionality is thread-safe, provided that there are no race-conditions on shared data, e.g., thread-parallel accesses to non-overlapping global memory regions are thread-safe. However, the outcome of concurrent or unsynchronized put operations to the same memory location is undefined.

There is a notable exception to these thread-safety rules in DASH and DART: any collective operation on the same team cannot be considered thread-safe. In particular, this includes team management and global memory allocation as well as synchronization and reduction operations on the same team. As a consequence, some DASH distributed algorithms may not be called on the same container by multiple threads as they rely on reduction operations or temporary global memory allocation. Where sensible, we will consider introducing variants of these algorithms that are thread-safe even if called on the same container or team.

Overall, this limitation is coherent with (and dictated by) the limitations outlined in chapter 12 of the MPI 3.1 standard.

3.5 Asynchronous Progress of One-Sided Communication

As described in Section 3.2, DART relies on the *passive target mode*, which enables participating units to asynchronously read or write globally shared data without involving the respective target unit, i.e., the memory owner. This model naturally matches PGAS semantics and provides high potential for computation-communication overlap in scientific and data-intensive applications. Ideally, MPI libraries asynchronously progress these one-sided operations even if the target unit is blocked in computation and thus does not call any MPI routine. This is best done by exploiting the RDMA-capabilities of today's high-performance networks.

However, MPI implementations commonly support less advanced networks that do not support RDMA as well. On some of these platforms we have observed communication operations at the source unit to stall if the target unit does not call MPI routines. According to the MPI standard "implementations must guarantee that a process makes progress on all enabled communications it participates in, while blocked on an MPI call" [9, Chapter 11.7.3]. While it is immediately clear that a process blocked on a barrier should make progress on imcoming RMA operations, the wording is less clear for local operations, e.g., busy-waiting on a signal using `MPI_Get` combined with a local flush. In the past, we have observed at least one implementation to not trigger remote progress in this use-case, effectively leading to livelocks. While, after reporting, these implementations now support progress in these cases, a clarification on which MPI operations trigger the progress engine would be desirable from a user's perspective. Moreover, an interface for querying the progress semantics of the current platform and implementation and for explicitly triggering the progress engine would be helpful for developers of PGAS abstractions based on the MPI-3 RMA standard to automatically adjust the behavior of the runtime to the characteristics of the given platform.

Table 1: Test system overview.

System	CPU	Network	Compiler	MPI
Hazel Hen	2 x E5-2680v3 12C	Cray Aries	GCC 6.3.0	CCE 8.5.3
SuperMUC	2 x E5-2697v3	IB FDR14	ICC 16.0.4	IBM POE 1.4
Linux-Cluster	1 x E5-2697v3	IB FDR14	ICC 16.0.4	OpenMPI 2.0.2

4 EVALUATION

We have used the OSU MPI benchmark suite to implement benchmarks for measuring the latency of global memory allocation in MPI, an aspect of MPI RMA that has not been previously covered by the suite. The code is modeled after the existing MPI-RMA latency measurement benchmarks and is available on GitHub at https://github.com/dash-project/dash-bench.[1] An example kernel for measuring the latency of MPI window allocation is presented in Figure 2, which is similar to the allocation process described in Section 3.2. We also added support for measuring communication latencies in DART in order to determine the induced overhead.

We conducted our measurements on three different systems: the Cray XC40 'Hazel Hen' installed at High Performance Computing Center Stuttgart (HLRS) as well as the IBM iDataPlex system 'SuperMUC' (phase 2) and a commodity GNU/Linux cluster, both installed at the Leibniz Supercomputing Center (LRZ). The system specifications are summarized in Table 1. All codes were compiled using either the GNU compiler collection (GCC) or Intel compiler (ICC). Unless noted otherwise, we have enabled support for the Distributed Memory Application (DMAPP) API [22] on Hazel Hen, which measurably improves MPI-RMA performance.

4.1 Memory Allocation Overhead

As described in Section 3.2, some DASH algorithms rely on temporary global memory for reduction operations. We were therefore interested in measuring the latency of global memory allocation in different MPI implementations to determine the best strategy, i.e., whether to use dynamic or regular windows. We therefore measured the latency of allocating MPI windows of various sizes using different numbers of processes and included three different allocation strategies: `Win_allocate`, `Win_create`, and `Win_dynamic`. The presented times reflect the average of 10 iterations performed in one benchmark run, thus following the OSU benchmark pattern.

The first two strategies simply use `MPI_Win_allocate` and `MPI_Win_create` with a buffer allocated through `malloc`, respectively. We have observed mostly similar results for these two strategies and thus refrain from presenting results for the latter. The third strategy uses a pre-allocated dynamic MPI window and allocates shared windows, whose memory

[1]The original MPI benchmark suite from the Ohio State University is available at http://mvapich.cse.ohio-state.edu/benchmarks/.

```
for (size = 0; size <= MAX_SIZE; size = size * 2) {
  for (i = 0; i < opts.skip + opts.loop; i++) {
    if (i == opts.skip) t_start = MPI_Wtime ();
    disp_set = malloc(comm_size * sizeof(MPI_Aint));
    MPI_Win_allocate_shared(size, 1, win_info,
      sharedmem_comm, &baseptr, &win);
    if (size > 0)
      MPI_Win_attach(dynamic_win, baseptr, size);
    MPI_Get_address(baseptr, &disp_local);
    MPI_Allgather(&disp_local, 1, MPI_AINT,
      disp_set, 1, MPI_AINT, MPI_COMM_WORLD);
    if (size > 0)
      MPI_Win_detach(dynamic_win, baseptr);
    MPI_Win_free(&win);
    free(disp_set);
  }
  t_end = MPI_Wtime ();
  print_latency(rank, size);
}
```

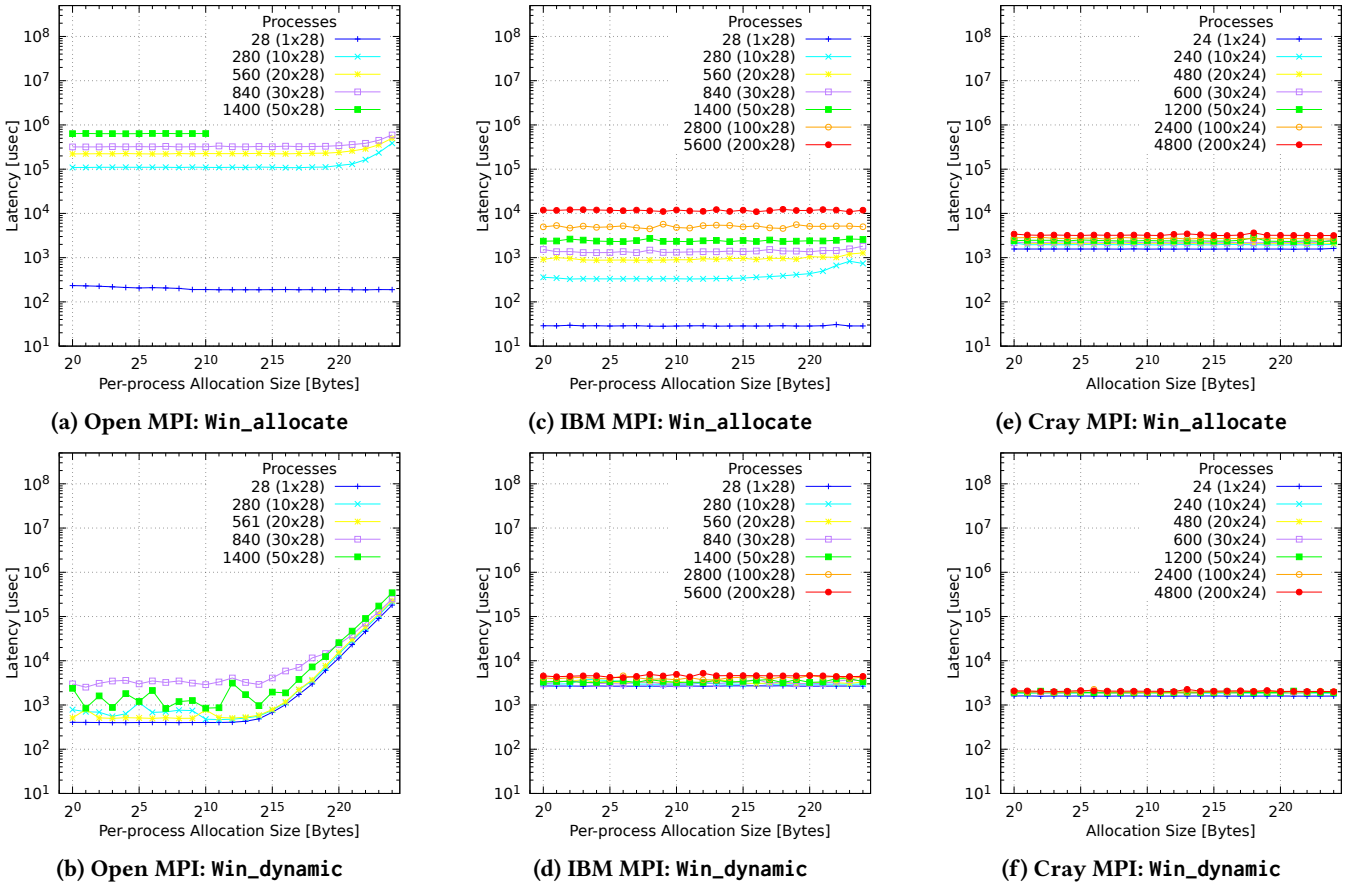
Figure 2: Kernel for measuring the latency of allocating an MPI window using shared memory windows.

is attached to the dynamic window and the global offset communicated using `MPI_Allgather`. This strategy is currently the default in DART, as described in Section 3.2.

The results reveal significant differences in latency, both between the three strategies as well as between the MPI implementations under test. The most striking result is the memory allocation latency in Open MPI 2.0.2, presented in Figures 3a and 3b. Results for `Win_allocate` are displayed in Figure 3a, which shows a significant difference between single-node and multi-node allocations, the latter exhibiting latencies of at least 100 ms and rising up to 600 ms for 32 MByte windows on 1400 processes. With `Win_dynamic`, the latencies are notably lower than with `Win_allocate`, achieving between 0.4 ms and 2 ms for smaller allocations. However, a steep increase starting at 32 KiB is visible, leading to 200 ms for 16 MiB. We attribute the lower allocation latencies to the fact that memory pages attached to dynamic windows are not registered with the Infiniband device. Neither are allocated windows if the application is running on a single node.

The latencies for IBM MPI measured on SuperMUC are depicted in Figures 3c and 3d. Here, the latency of `Win_dynamic` is less dependent on the process count than with `Win_allocate`. The latter shows lower latencies for lower process counts (up to 1400 processes) but higher latencies for larger process counts, reaching up to ≈5 ms compared to more than 10 ms for 5600 processes.

With Cray MPI, depicted in Figures 3e and 3f, the latencies for `Win_allocate` appear similar to the latencies of IBM MPI using `Win_dynamic`, ranging from 1.6 ms to 3.3 ms. Using



(a) Open MPI: `Win_allocate` (c) IBM MPI: `Win_allocate` (e) Cray MPI: `Win_allocate`

(b) Open MPI: `Win_dynamic` (d) IBM MPI: `Win_dynamic` (f) Cray MPI: `Win_dynamic`

Figure 3: Latency of different window allocation strategies.

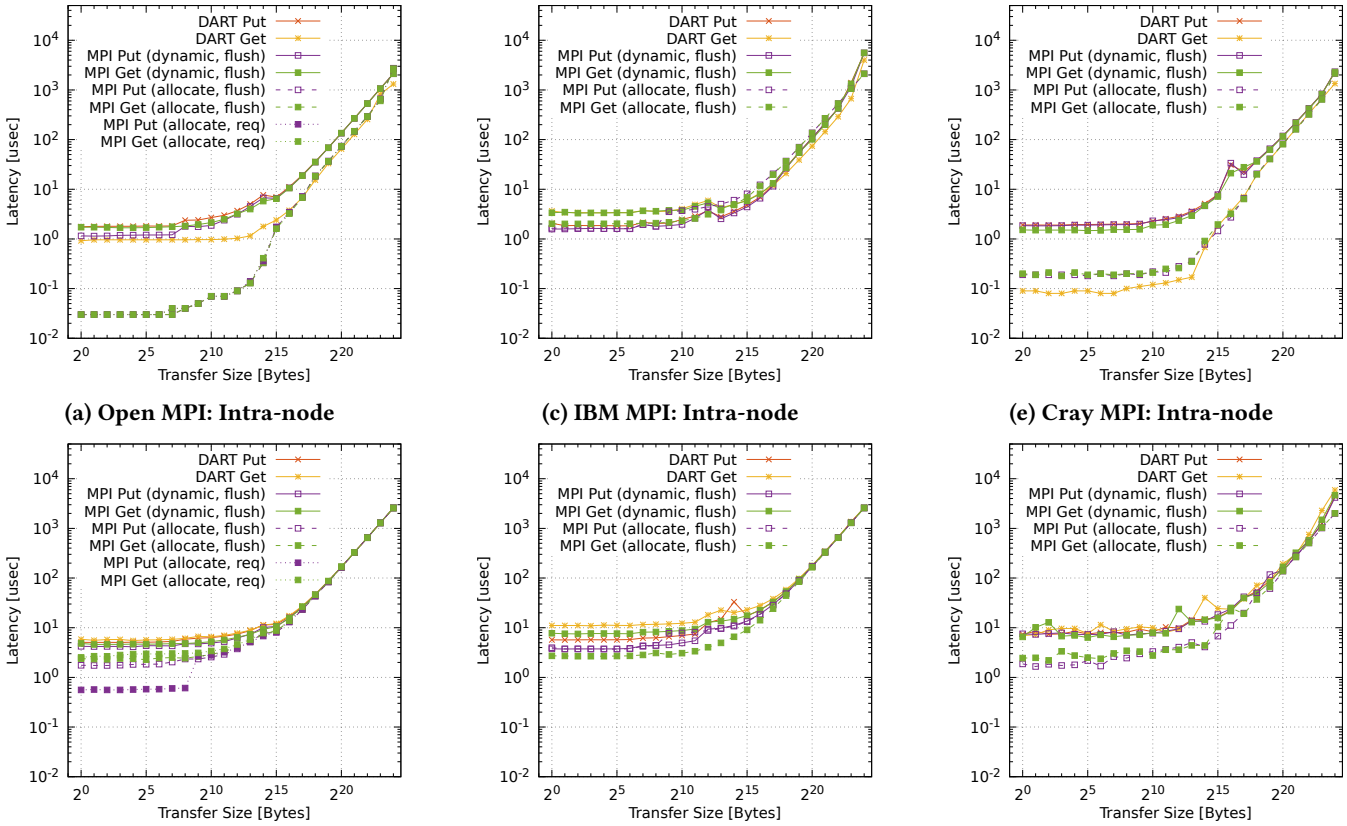
the `Win_dynamic` strategy on Hazel Hen, we observe almost no influence of the number of processors on the allocation latency, marking it stable at around 2 ms. Moreover, we do not observe any influence of the per-process allocation size in our benchmark.

4.2 Communication Latency

To complement the picture drawn in the previous section, we provide latency measurements for DART as well as MPI communication with both allocated and dynamic windows, i.e., using the strategies `Win_allocate` and `Win_dynamic`. The measured times reflect the average of 100 or 10 iterations, for sizes below and above 8 KiB, respectively. As before, we conducted measurements for Open MPI (Figures 4a and 4b), IBM MPI (Figures 4c and 4d), and Cray MPI (Figures 4e and 4f) for both intra- and inter-node communication. All DART measurements include the time for remote completion.

For intra-node communication, the benefit of DART's shared memory window optimization (as described in Section 3.2) are only visible for Open MPI and Cray MPI because we were unable to use shared memory windows with IBM MPI. However, it appears that using allocated windows in Open MPI for intra-node communication induces only marginal overhead when used with `MPI_Rput` and `MPI_Rget` ("*allocate, req*" variant). DART on the other hand yields around 1 μs for `get` and higher for `put` even with shared memory optimization enabled. We will have to further investigate this discrepancy as there seems to be some constant overhead induced by DART when using Open MPI. On Hazel Hen, the shared memory optimization in DART outperforms the MPI implementation for `get` operations by a factor of two. However, we have to investigate the discrepancies between `get` and `put` on this machine since the latter does not seem to benefit from the shared memory optimization.

For inter-node communication, a clear difference emerges between allocated and dynamic windows. It appears that dynamic windows incur higher latencies with all tested implementations, amounting to factor 3-4 x for small transfer sizes. For larger transfer sizes, this difference is negligible due to the generally higher latencies. We attribute that to the previously mentioned lack of registration of the memory



(a) Open MPI: Intra-node (c) IBM MPI: Intra-node (e) Cray MPI: Intra-node

(b) Open MPI: Inter-node (d) IBM MPI: Inter-node (f) Cray MPI: Inter-node

Figure 4: Latency of `put` and `get` operations in DART and MPI.

pages with the network device, which prevents RDMA and requires a software protocol to be used.

An interesting effect can be observed with Open MPI: the latency of `MPI_RPut` in combination with `MPI_Wait` ("*MPI Put (allocate, req)*") is significantly lower for small transfer sizes up to 256 B than the combination of `put` and `flush` or `flush_local`. We consider this a strong argument for DART to offer multiple blocking put operations as described in Section 3.3, e.g., one that only guarantees local completion. However, we have not observed similar differences with other implementations.

In general, DART seems to incur a small additional overhead compared to the raw MPI operations. However, we are still in the process of optimizing DART, trying to etch out remaining sources of overhead.

4.3 Discussion

Our measurements presented above have shown heterogeneous performance across different MPI implementations using the two allocation strategies `Win_allocate` and `Win_dynamic`. While all implementations generally provide lower latencies for dynamic window allocation in combination with shared memory windows, the communication latency is significantly higher for dynamic windows across all implementations. In contrast, regular window allocations (`Win_allocate`) yield higher allocation latencies for larger numbers of processes with one implementation going beyond 100 ms per allocation. These results confirm our initial expectation expressed in Section 3.2 that segment allocation in DART induces lower latencies if done through dynamic and shared memory windows compared to regular window allocations.

However, the results of our communication latency measurements require us to reconsider the default allocation strategy in DART. On the one hand, the lower allocation latencies are better suited for temporary allocations in DASH. However, frequent allocations induce noticeable overhead due to the general latency in the millisecond range. On the other hand, increased inter-node communication latencies of dynamic windows may have a bigger impact than temporary allocation latencies on most applications. Moreover, the impact of the shared memory optimization will be diminished

in multi-threaded DASH applications. Hence, choosing the right allocation strategy is an application-specific decision.

Even without frequent global memory allocations these high allocation latencies can become problematic. In other benchmarks (not depicted here), we have observed latencies in the multi-second range for large allocations. Extrapolation of our measurements to current and future large-scale systems indicate severe scalability issues in some MPI implementation that should be addressed on the path to Exascale.

A proposal made in the MPI Forum[2] for a future version of the MPI standard to allow shared memory access even on regular windows would enable us to combine both low-latency inter-node communication with application-level optimization for intra-node data exchange.

As a consequence of our measurements presented above, we have extended DART to also support allocated windows, which can be chosen at compile-time. This will prevent the usage of the shared memory optimization but should yield lower latencies between units on different nodes. We will also consider alternatives to allocating temporary global memory in DASH algorithms, e.g., using pre-allocated scratch space. We hope that our measurements raise awareness of this performance heterogeneity among both MPI-RMA implementors and application developers.

5 RELATED WORK

Several different approaches towards the PGAS programming paradigm exist today, which can be classified into three major categories: newly designed PGAS languages, extensions to existing languages, and library-based approaches. The set of new languages supporting the PGAS programming paradigm include Chapel [4] and X10 [6]. However, forcing users to port whole applications comes with a high entry barrier and no incremental transition path.

Several approaches have been built as an extension to existing programming languages, including Unified Parallel C (UPC) [23] and Fortran 2008 co-arrays (CAF) [21]. Efforts have been made to provide an implementation of CAF that is built on MPI-3 RMA operations [8, 24]. The XcalableMP (XMP [18]) approach adds PGAS functionality to existing C or Fortran applications through compiler pragmas, which are mapped to MPI-3 RMA primitives.

Among the library-based approaches are OpenShmem [5], GlobalArrays [20], as well as the MPI-3 RMA extensions, which all provide a C interface for remote memory access. Both OpenShmem and GlobalArrays have been ported to run on top of MPI-RMA [7, 15].

Several approaches make use of C++11 language features to provide access to global data without explicit API function calls. Among them are UPC++ [26] and Co-array C++ [17] together with the DASH library. We refer to the DASH overview paper for a detailed discussion of the DASH approach and it's comparison with other PGAS approaches [11].

As the runtime system for DASH, DART can be seen as a member of this category as it provides an abstraction of RMA operations provided by existing PGAS libraries, e.g., MPI-3 RMA operations. The initial design and optimizations using the MPI-3 shared memory extensions have been described in [28] and [27].

Several works have also conducted an evaluation of the performance of MPI-RMA operations, e.g., for MPI-2 [13] and MPI-3 [16].

6 CONCLUSION AND FUTURE WORK

In this paper, we have presented the general requirements of the DASH RunTime (DART) for its underlying PGAS communication backend and described some of the design decisions. Moreover, the paper discusses details on some of our recent experiences in the implementation of DART based on MPI-3 RMA, including our approach to memory consistency, global memory allocation strategies, and progress related issues. We have extended the OSU micro-benchmark suite to include measurements of the performance characteristics of different window allocation strategies in MPI, including memory allocation latency and communication latency. The results show significant differences in both metrics, with the general trend towards lower allocation and higher communication latencies when using dynamic windows.

Based on the measurements presented in this paper, we will reconsider our use of dynamic and shared memory windows in DART and try to find ways to adapt DART to the performance characteristics of the underlying MPI implementation. As a first step, we have made the use of dynamic windows in DASH optional for our upcoming release, allowing users to adapt DART to the application's needs. In addition, future work will include further optimization of DART and additional benchmarks for DART communication operations in the framework of the OSU benchmark suite.

An important aspect of DART is the ability to ensure asynchronous progress of communication operations. We will thus closely monitor the developments of the MPI standard and adapt our implementation accordingly. A systematic analysis of the capabilities of current MPI implementations and networks to asynchronously progress RMA operations might yield important information for the development of a well-designed progress model in DART. We will also consider using an alternative communication backend in addition to MPI to test the design decisions made so far against different RMA implementations, e.g., UCX and GASPI.

Finally, despite the issues outlined in this paper, we are confident that using MPI-3 RMA as the basis for the DASH

[2] https://github.com/mpi-forum/mpi-forum-historic/issues/397

PGAS abstraction has been a sensible choice as it opens the door for an incremental transition of traditional MPI applications to DASH and allows for easy porting to new systems that provide a compatible MPI implementation.

ACKNOWLEDGMENTS

We would like to thank the members of the DASH team and gratefully acknowledge funding by the German Research Foundation (DFG) through the German Priority Programme 1648 Software for Exascale Computing (SPPEXA) in the SmartDASH project.

REFERENCES

[1] Saman Amarasinghe, Dan Campbell, William Carlson, Andrew Chien, William Dally, Elmootazbellah Elnohazy, Mary Hall, Robert Harrison, William Harrod, Kerry Hill, et al. 2009. Exascale software study: Software challenges in extreme scale systems. *DARPA IPTO, Air Force Research Labs, Tech. Rep* (2009).

[2] Roberto Belli and Torsten Hoefler. 2015. Notified access: Extending remote memory access programming models for producer-consumer synchronization. In *Parallel and Distributed Processing Symposium (IPDPS), 2015 IEEE International*. IEEE.

[3] Dan Bonachea and Jason Duell. 2004. Problems with Using MPI 1.1 and 2.0 As Compilation Targets for Parallel Language Implementations. *Int. J. High Perform. Comput. Netw.* 1, 1-3 (Aug. 2004). https://doi.org/10.1504/IJHPCN.2004.007569

[4] Bradford L. Chamberlain, David Callahan, and Hans P. Zima. 2007. Parallel Programmability and the Chapel Language. *International Journal of High Performance Computing Applications* 21 (August 2007).

[5] Barbara Chapman, Tony Curtis, Swaroop Pophale, Stephen Poole, Jeff Kuehn, Chuck Koelbel, and Lauren Smith. 2010. Introducing OpenSHMEM: SHMEM for the PGAS community. In *Proceedings of the Fourth Conference on Partitioned Global Address Space Programming Model*.

[6] Philippe Charles, Christian Grothoff, Vijay Saraswat, Christopher Donawa, Allan Kielstra, Kemal Ebcioglu, Christoph Von Praun, and Vivek Sarkar. 2005. X10: an object-oriented approach to non-uniform cluster computing. *ACM Sigplan Notices* 40, 10 (2005).

[7] J. Dinan, P. Balaji, J. R. Hammond, S. Krishnamoorthy, and V. Tipparaju. 2012. Supporting the Global Arrays PGAS Model Using MPI One-Sided Communication. In *IEEE 26th International Parallel and Distributed Processing Symposium*. https://doi.org/10.1109/IPDPS.2012.72

[8] Alessandro Fanfarillo, Tobias Burnus, Valeria Cardellini, Salvatore Filippone, Dan Nagle, and Damian Rouson. 2014. OpenCoarrays: Open-source Transport Layers Supporting Coarray Fortran Compilers. In *Proceedings of the 8th International Conference on Partitioned Global Address Space Programming Models*. ACM. https://doi.org/10.1145/2676870.2676876

[9] MPI Forum. 2015. *MPI: A Message-Passing Interface Standard*. Standard. http://mpi-forum.org/docs/mpi-3.1/mpi31-report.pdf

[10] Tobias Fuchs and Karl Fürlinger. 2016. Expressing and Exploiting Multi-Dimensional Locality in DASH. In *Software for Exascale Computing-SPPEXA 2013-2015*. Springer.

[11] Karl Fuerlinger, Tobias Fuchs, and Roger Kowalewski. 2016. DASH: A C++ PGAS Library for Distributed Data Structures and Parallel Algorithms. In *2016 IEEE 18th International Conference on High Performance Computing and Communications*. https://doi.org/10.1109/HPCC-SmartCity-DSS.2016.0140

[12] Antonio Gómez-Iglesias, Dmitry Pekurovsky, Khaled Hamidouche, Jie Zhang, and Jérôme Vienne. 2015. Porting Scientific Libraries to PGAS in XSEDE Resources: Practice and Experience. In *Proceedings of the 2015 XSEDE Conference: Scientific Advancements Enabled by Enhanced Cyberinfrastructure (XSEDE '15)*. ACM.

[13] William D. Gropp and Rajeev Thakur. 2007. *Revealing the Performance of MPI RMA Implementations*. Springer Berlin Heidelberg. https://doi.org/10.1007/978-3-540-75416-9_38

[14] Daniel Grünewald and Christian Simmendinger. 2013. The GASPI API specification and its implementation GPI 2.0. In *7th International Conference on PGAS Programming Models*.

[15] Jeff R. Hammond, Sayan Ghosh, and Barbara M. Chapman. 2014. *Implementing OpenSHMEM Using MPI-3 One-Sided Communication*. Springer International Publishing. https://doi.org/10.1007/978-3-319-05215-1_4

[16] Nathan Hjelm. 2014. Optimizing One-sided Operations in Open MPI. In *Proceedings of the 21st European MPI Users' Group Meeting (EuroMPI/ASIA '14)*. ACM. https://doi.org/10.1145/2642769.2642792

[17] Troy A Johnson. 2013. Coarray C++. In *7th International Conference on PGAS Programming Models*.

[18] J. Lee and M. Sato. 2010. Implementation and Performance Evaluation of XcalableMP: A Parallel Programming Language for Distributed Memory Systems. In *2010 39th International Conference on Parallel Processing Workshops*. https://doi.org/10.1109/ICPPW.2010.62

[19] Jarek Nieplocha and Bryan Carpenter. 1999. ARMCI: A portable remote memory copy library for distributed array libraries and compiler run-time systems. In *Parallel and Distributed Processing*. Springer.

[20] Jaroslaw Nieplocha, Robert J Harrison, and Richard J Littlefield. 1994. Global Arrays: a portable shared-memory programming model for distributed memory computers. In *Proceedings of the 1994 ACM/IEEE conference on Supercomputing*.

[21] Robert W. Numrich and John Reid. 1998. Co-array Fortran for parallel programming. *SIGPLAN Fortran Forum* (Aug. 1998). https://doi.org/10.1145/289918.289920

[22] Monika ten Bruggencate and Duncan Roweth. 2010. DMAPP – An API for One-sided Program Models on Baker Systems. In *52. Cray User Group (CUG)*.

[23] UPC Consortium. 2005. *UPC Language Specifications, v1.2*. Tech Report LBNL-59208. Lawrence Berkeley National Lab. http://www.gwu.edu/~upc/publications/LBNL-59208.pdf

[24] Chaoran Yang, Wesley Bland, John Mellor-Crummey, and Pavan Balaji. 2014. Portable, MPI-interoperable Coarray Fortran. In *Proceedings of the 19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '14)*. ACM. https://doi.org/10.1145/2555243.2555270

[25] Katherine Yelick, Dan Bonachea, Wei-Yu Chen, Phillip Colella, Kaushik Datta, Jason Duell, Susan L. Graham, Paul Hargrove, Paul Hilfinger, Parry Husbands, Costin Iancu, Amir Kamil, Rajesh Nishtala, Jimmy Su, Michael Welcome, and Tong Wen. 2007. Productivity and Performance Using Partitioned Global Address Space Languages. In *Proceedings of the 2007 International Workshop on Parallel Symbolic Computation (PASCO '07)*. ACM.

[26] Yili Zheng, Amir Kamil, Michael B Driscoll, Hongzhang Shan, and Katherine Yelick. 2014. UPC++: a PGAS Extension for C++. In *Parallel and Distributed Processing Symposium, 2014 IEEE 28th International*.

[27] Huan Zhou, Kamran Idrees, and José Gracia. 2015. Leveraging MPI-3 Shared-Memory Extensions for Efficient PGAS Runtime Systems. In *Euro-Par 2015: Parallel Processing - 21st International Conference on Parallel and Distributed Computing, Vienna, Austria, August 24-28, 2015, Proceedings*. https://doi.org/10.1007/978-3-662-48096-0_29

[28] Huan Zhou, Yousri Mhedheb, Kamran Idrees, Colin Glass, José Gracia, Karl Fürlinger, and Jie Tao. 2014. DART-MPI: An MPI-based Implementation of a PGAS Runtime System. In *The 8th International Conference on Partitioned Global Address Space Programming Models (PGAS)*. https://doi.org/10.1145/2676870.2676875