International Journal of Networking and Computing – www.ijnc.org
ISSN 2185-2839 (print) ISSN 2185-2847 (online)
Volume 7, Number 2, pages 136–153, July 2017

Application Productivity and Performance Evaluation of Transparent Locality-aware One-sided Communication Primitives

Huan Zhou
High Performance Computing Center Stuttgart (HLRS), University of Stuttgart
70569 Stuttgart, Germany

and

José Gracia
High Performance Computing Center Stuttgart (HLRS), University of Stuttgart
70569 Stuttgart, Germany

Received: January 24, 2017
Revised: May 2, 2017
Accepted: May 31, 2017
Communicated by Ryusuke Egawa

Abstract

Nowadays, the individual nodes of a distributed parallel computer consist of multi- or many-core processors allowing to execute more than one process per node. The large difference in communication speed within a node through shared memory, versus across nodes through the network interconnect, requires to use locality-aware communication schemes for any efficient distributed application. However, writing an efficient locality-aware MPI code is complex and error-prone, because the developer has to use very different APIs for communication operations within and across nodes, respectively, and manage inter-process synchronization. In this paper, we analyze and enhance a recent one-sided communication model, namely DART-MPI, which is implemented on top of MPI-3. In this runtime system, the complexities of handling locality-awareness of MPI memory access operations, either remote or local, and the related synchronization calls are hidden inside the related DART-MPI interfaces resulting in concise code and improved application and developer productivity. We have carried out in-depth evaluation of our DART-MPI system. Foremost, a micro benchmark is conducted to help understanding the prime performance overhead of implementing APIs in DART-MPI system, which is small and becomes negligible with the growing message sizes. We then compare the performance of DART-MPI and flat MPI without locality awareness, in particular blocking and non-blocking memory operations, using a realistic scientific application on a large-scale supercomputer. The comparison demonstrates that in most cases the DART-MPI version of this application shows better performance than the flat MPI version. Further, we compare the DART-MPI version to a functionally equivalent MPI version, which thus includes code to deal with data-locality, and show that DART-MPI realizes almost the full potential of highly optimized MPI while maintaining high productivity for non-expert programmers.

Keywords: Locality-awareness, MPI, DART, DART-MPI, blocking, non-blocking, one-sided communication, programmer productivity

1 Introduction

The parallel programming models are developed to shorten the execution time of a distributed simulation by concurrently managing the multiple subtasks. Meantime, the communication and synchronization between different subtasks due to the data dependencies should be validated. The Message Passing Interface (MPI, [1]) remains as the dominant communication model as a result of its high performance, portability and standardization on leading computing systems. Multi- or many-core processors are commonly deployed in today's computation clusters. This fuels an explosive growth of processing capability. According to the new TOP500 Supercomputer report [2], the most powerful supercomputer as of Nov. 2016 – Sunway – exemplifies this fact with core counts exceeding 10,000,000. Comparatively, communication overhead becomes the main impediment to achieving high-speed and scalable distributed simulation performance.

However, multi-core clusters impose new challenge into the application software design in a way of highlighting the intra-node communication operations. To get the optimal intra-node communication performance, researchers need to adapt their parallel programs to be aware of data-locality and exploit all available communication paths. For instance, data exchange within a node should bypass the MPI communication primitives and directly use memory load/store operations.

The MPI standard has been scaling and evolving to keep up with the scalable and productive computation hardware. MPI-2 incorporates the Remote-Memory-Access (RMA, also called one-sided) interfaces to avoid the matching affairs inherent in the two-sided operations. Moreover, MPI-3 RMA is extended to support the shared-memory window. The results provided in earlier works [3,4] have proved that the MPI-integrated shared memory window is a promising alternative since it allows direct on-node load/store accesses by shared-memory mapping. Unlike Partitioned Global Address Space (PGAS) model, MPI does not intuitively support for the data locality-awareness. Thus, the programmer should have profound knowledge on the details in MPI when they purely write MPI parallel applications with data locality in mind. Besides that, the programmer should take great responsibility for guaranteeing the optimal performance as well as correctness of the applications, the procedure of which is however error-prone and time-consuming.

The earliest paper describes DART-MPI [5] as a portable and raw implementation of PGAS runtime system, which uses MPI-3 as low-level communication substrate. Originally, all RMA operations in DART-MPI are directly mapped to the corresponding MPI-3 RMA operations , even when source and target of a transfer reside on the same node and share local, physical memory resource. Obviously, this flat mapping methodology fails to support a locality-aware communication scheme. Further, the MPI-3 shared-memory extensions to enable direct memory access (memory sharing) for DART-MPI blocking operations for intra-node transfers are utilized in the optimized DART-MPI [6], which thus allows awareness of locality during the transfer of data. This leads DART-MPI to be an efficient as well as portable parallel programming for today's multi- or many-core clusters. So far this communication library has been adopted and proved as the efficient runtime system in a DASH project [7], which is a PGAS approach. In a recent paper [8], the conciseness and readability of DART-MPI code of data locality-awareness (written with the blocking one-sided primitives) is demonstrated due to that the DART-MPI system internally hides the MPI complexities away from the users. We demonstrate that little effort in porting one-sided MPI (without locality awareness) onto DART-MPI (intuitively with locality-awareness) is needed by using 3D heat conduction as a user case. Also, we observe the performance gain provided by the DART-MPI port.

The rest of the paper is organized in the following ways. In Section 2, we briefly discuss some background used in this paper. In Section 3, we explain the method of hiding details and complexities of MPI-3 RMA from the user in DART-MPI. We demonstrate the terseness and clarity of DART-MPI code in comparison to locality-aware MPI code and analyze the potential DART implementation cost in Section 4. Finally, in Section 5 we elementarily quantify the implementation overhead of DART-MPI due to its encapsulation of MPI semantics and process topology as well as demonstrate the performance of DART-MPI using a heat-diffusion problem, which is representative for a wide class of numerical simulation schemes. In particular, we present a comprehensive picture of the comparative results between the DART-MPI and *flat* MPI-3 (blocking or nonblocking)

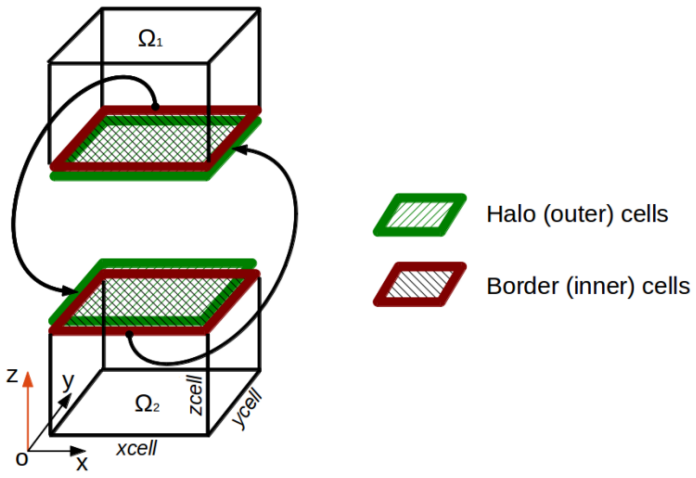


Figure 1: Halo cells exchange in 3D grid. Exchange of the matrix data (a total of $xcell \times ycell$ grid cells) happen on Oxy plane between neighboring processes in the direction of z-axis, where sub-domain Ω_2 is the *Down* neighbor of sub-domain Ω_1.

RMA implementations of this application on large-scale, supercomputers such as Cray XC40 system. Here, the flat MPI-3 RMA implementations do not take the data locality into consideration. We then further investigate the impact of the accumulated overhead of the DART-MPI implementation by comparing the performance of the flat MPI version, an optimized MPI version with locality awareness and the DART-MPI version of heat-diffusion problem, respectively.

2 Background

2.1 DART

DART defines common concepts, terminology and abstracts from the underlying communication substrate and hardware. DART establishes a partitioned global address space and provides functions to handle memory efficiently, such as memory allocation and data movement. In addition, DART also provides functions for initialization, synchronization and management of teams, which are similar to the MPI communicators. In a DART program an individual participate is called unit. DART provides the functions of *dart_init* and *dart_exit* for initialization and finalization. Importantly, providing and working with a global memory is the focus of DART. A region of global memory in DART functions like the window in MPI, which is supposed to be remotely accessed by other processes. The global address space is a virtual abstraction, with each unit contributing a part of its local memory. Remote data items are addressed by global pointers provided by DART. The DART one-sided communications (data movement) consist of blocking operations (refer to *dart_get_blocking* and *dart_put_blocking*) as well as non-blocking operations (refer to *dart_get* and *dart_put*). The DART blocking operations can guarantee both the local and remote completeness. For the non-blocking operations, DART provides functions, i.e., *dart_wait*/*dart_waitall* and *dart_test*/*dart_testall*, to guarantee that or check whether the message transfers are completed. The complete DART specification is available on-line [1].

2.2 Heat conduction

The heat conduction [9] is a mode of heat transfer owing to molecular activity and occurs in any material (e.g., solids, fluids and gases). We used the application source code parallelized with

[1] http://doc.dash-project.org/api/dash-0.2.0/html/files.html

MPI at [10]. This application simulates the phenomenon of 3D heat conduction in solids with temperature-dependent thermal diffusivity. The heat conduction is the transfer of heat between substances that are in direct contact with each other and is a kind of heat transfer due to molecular activity. Such conduction occurs in solids, fluids and gases. The Partial Differential Equation (PDE) for *unsteady* 3D heat conduction is obtained in the Cartesian coordinate system. Here, *unsteady* means the temperatures may change during the process of heat conduction. This application solves the PDE over a 3D grid by using the Finite Difference Method (FDM). Boundary conditions are constant temperatures at the edges of the 3D grid.

Moreover, parallelization is done based on the checkerboard domain decomposition. After the decomposition each process has six neighbors located in the direction of *East-West* (x-axis), *North-South* (y-axis) and *Up-Down* (z-axis) according to the Cartesian process topology. The exceptional case is the processes owning the edge cells will not have all of the six neighbors. Each process sends each face of its sub-domain (a 3D sub-grid), i.e., matrix, to its corresponding neighbor for the computation.

Each process exchanges the border data with its corresponding neighbors and the halo cells are reserved to receive the exchanged data. The original 3D heat conduction application fills the halo cells in each iteration using the point-to-point communication operation (i.e., *MPI_Sendrecv*). Each cell of the grid is a 8-byte double-precision floating-point number. The abort-criterion convergence is achieved by invoking the collective operation, such as all-to-all reduce. Assuming, that the 3D grid size is represented as ($size\_x \times size\_y \times size\_z$) and P is denoted as the number of participating processes (each process is pined to a core). The number of cores P can also be represented as ($domain\_x \times domain\_y \times domain\_z$) in the Cartesian coordinate system. The computation is then distributed equally among processes. Each process gets a sub-grid with cells of ($xcell \times ycell \times zcell$), where $xcell$ equals to $size\_x/domain\_x$, $ycell$ equals to $size\_y/domain\_y$ and $zcell$ is equivalent to $size\_z/domain\_z$. Figure 1, takes the direction of z-axis for example, to explain the way of boundary data exchange between two neighboring processes.

3 Methods of employing the MPI RMA synchronization calls in DART

The MPI RMA communication operations coupled with explicit passive synchronization operations (with shared lock) form a theoretic foundation for building the DART RMA model [5]. In fact, each MPI RMA communication call associated with a window is non-blocking. It must proceed within an access epoch for this window at a process to start and complete the issued RMA communications. Superficially, the RMA semantics in MPI and DART both require the independent management of synchronization and communication. However, besides the non-blocking operations, DART RMA also includes blocking operations. Also, the concept of access epoch is not relevant in the DART RMA semantics according to the DART specification. In addition, a handle is associated with each DART non-blocking RMA communication operation and in the further passed to the DART RMA synchronization calls. Therefore, we need to bridge the semantic gap between MPI RMA and DART RMA in terms of communication and synchronization operations.

The implementation of DART non-blocking RMA operations – *dart_put* and *dart_get* depends simplistically on the counterparts in MPI (i.e., *MPI_Put* and *MPI_Get*).

Additionally, there are two typical methods of solving the semantic mismatches between MPI RMA and DART RMA synchronization operations. Note, that the two methods reflect a common effort: hiding all primitives pertaining to MPI access epoch from DART programmers. This focus creates the illusion that all DART global memory regions are protected and exposed once allocated and also the expected processes are allowed to access those global memory regions directly. One method is to use the single shared lock/unlock interfaces (i.e., *MPI_Win_lock* /*MPI_Win_unlock*), where an origin process could lock only one target at a time. Therefore, the operations regarding the open or closure of MPI access epoch to a certain target should be integrated into the DART RMA communication call or synchronization call on demand. This leads to a situation where only one MPI RMA communication call proceeds within an access epoch. I.e., the amount of MPI RMA

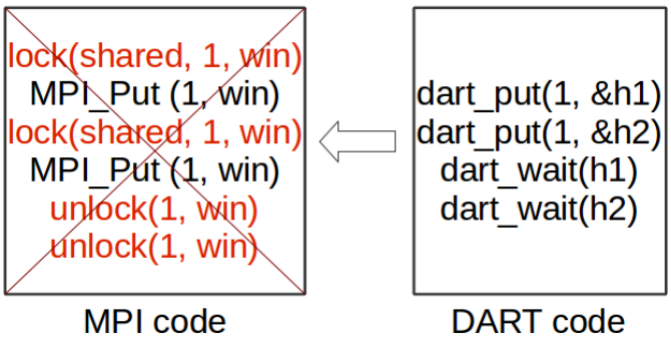


Figure 2: An unsupported overlapping scenario based on MPI single lock operations.

synchronization calls embedded is in direct proportion to that of MPI RMA communication calls. In fact, the MPI RMA synchronization operation adds a substantial overhead [11]. Hence, this method will definitely lead to great performance loss for applications with considerable RMA communication requests. Besides the performance loss, the overlap structures among different single lock operations are limited. Figure 2 displays a common invocation pattern of DART RMA communication operations, where two consecutive *dart_put* calls working on the same target and global memory block are issued first, then followed by two *dart_wait* calls to guarantee the completion of the previous two put operations. The corresponding MPI code is published by unfolding the DART code and shows an overlap scenario where two access epochs pertain to the same window and target. Such overlap scenario will lead to runtime error, which, in turn, says that DART RMA model fails to support the common invocation pattern listed in Fig. 2 when the single shared lock/unlock is adopted. Therefore, applying the single shared lock/unlock to DART RMA model is neither practical nor safety, which stimulates the demand for another adequate method.

A global lock model supported in MPI-3 RMA provides two routines, i.e., *MPI_Win_lock_all* /*MPI_Win_unlock_all*), to allow locking/unlocking multiple targets simultaneously. I.e., the pair of *MPI_Win_lock_all* and *MPI_Win_unlock_all* enables programmer to lock/unlock all processes in a certain window with a shared lock. Hence, a new method emerges to implementing the DART RMA model by leveraging the global lock/unlock routines. Given the DART collective and non-collective global memory managements differ in concept and design, we will explain how to safely expose these two types of global memory independently based on this method.

A global *win* (MPI-created window object) and a global *shmem-win* (shared memory window object) are created for non-collective global memory allocations once a DART program is initiated [6]. Therefore, each process/unit needs to add two *MPI_Win_lock_all* calls for the global *win* and *shmem-win* to start two protected shared access epochs to all other units. Prior to freeing up the global *win* and *shmem-win* (done inside *dart_exit*), each unit issues two matched *MPI_Win_unlock_all* calls to end the above two access epochs.

Unlike non-collective global memory allocations, collective global memory allocations always involve all units in the given team. Thus, there are two scenarios where the creation of *d-win* takes place:

1. When a DART program is launched, a *d-win* for *DART_TEAM_ALL* is created.
2. When a new team (e.g., team T) is created, a *d-win* for team T is created.

In each of the above scenarios, a shared lock all epoch for the *d-win* is started. Additionally, I associate a *shmem-win* with a region of allocated collective global memory. This necessitates the call to *MPI_Win_lock_all* for the *shmem-win*.

Likewise, we can complete the lock all epoch by a call to *MPI_Win_unlock_all* when the DART program is finalized or team T is destroyed. A call to *MPI_Win_unlock_all* is also entailed to complete a shared RMA access epoch along with the deallocation of the collective global memory region. Com-

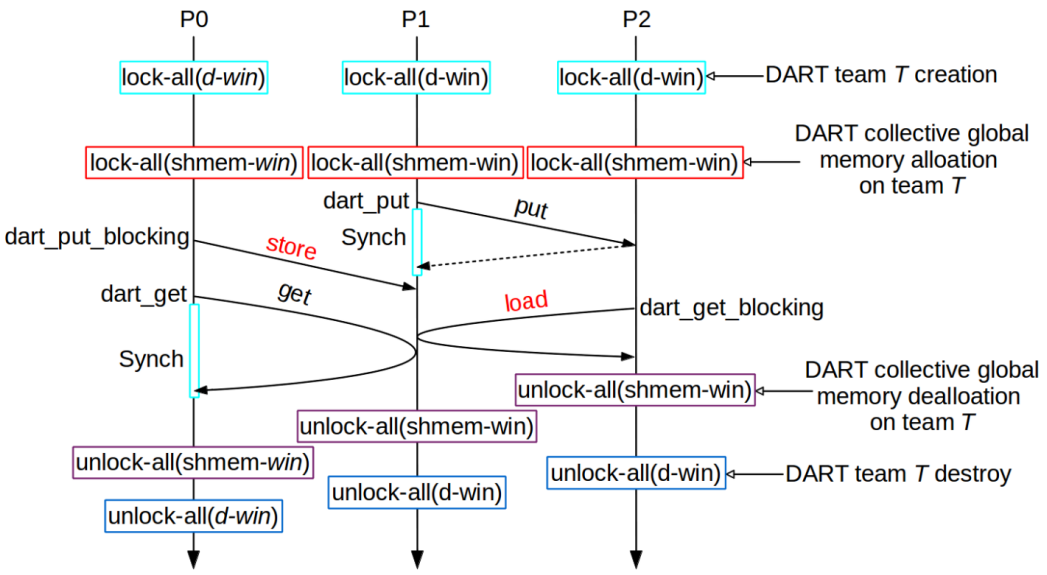


Figure 3: Example of DART-MPI RMA operations within two passive epochs. Here, the two passive epochs are started by a call to *MPI_Win_lock_all* and ended by a call to *MPI_Win_unlock_all*. The Synch means the operation of *dart_wait*.

pared to the method based on the single lock/unlock mechanism, this method avoids the proneness to error and repeated open or closure operations on the access epoch.

When we harness the global lock mechanism to realize the DART RMA model, remote completion of communications without ending the access epoch can be achieved with the *flush* routines (i.e., *MPI_Win_flush* and its all-, any- and local-variants) or usual MPI request completion routines (i.e., *MPI_Wait* and its all variants). Particularly, an *MPI_Win_flush* specifies a certain target and ensures that all previous operations to the target are locally and remotely finished.

The DART synchronization routines are expected to ensure the remote and local completion of DART non-blocking RMA communications. Therefore, the explicit MPI bulk synchronization using *flush* is integrated into *dart_wait*. Likewise, the *dart_waitall* performs a series of related calls to *MPI_Win_flush*. Figure 3 thoroughly shows the exemplary DART RMA events happening on the collective global memory region across team T with MPI global lock mechanism. Here we assume that the team T consists of three processes locating within one node.

When possible (i.e., intra-node), the shared-memory windows described in the paper [6] are always used by DART blocking RMA operations to perform load or store instructions. Additionally, the DART intra-node non-blocking RMA operations are designed as the MPI RMA operations on *shmem-win*, whilst all the DART inter-node RMA operations turn to the MPI RMA operations on *d-win* or *win*. Detailedly, we invoke an *MPI_Rget* followed by an *MPI_Wait* inside the DART inter-node blocking get operation. As for the DART inter-node blocking put operation, an *MPI_Put* is invoked followed by an *MPI_Win_flush*. Hence we can observe that DART spontaneously has data locality in mind when performing the RMA operations.

4 Comparison of DART and locality-aware MPI code and analysis of DART implementation cost

Figures 4 and 5 are provided to show the MPI and DART codes that are used to send a message from the origin to another random target process with the data locality in mind. Specifically, the red code lines, highlighted in Fig. 5, are required for handling the intra-node data transfers. It is clearly observed that in MPI code, the window creation/destroy operations as well as the access

```
dart_init ();
dart_team_memalloc_aligned (DART_TEAM_ALL, nbytes, &gptr);
dest = randomdst_generator ();
dart_gptr_setunit (&gptr, dest);
dart_put_blocking (gptr, srcptr, nbytes);
dart_team_memfree (DART_TEAM_ALL, gptr);
dart_exit ();
```

Figure 4: An example for DART code. This example shows the DART code for transferring data from an origin unit to another random target unit with data locality in mind.

```
MPI_Init (...);
MPI_Comm_size (MPI_COMM_WORLD, &size);
MPI_Comm_split_type (MPI_COMM_WORLD,..., &sharedmem_comm);
MPI_Win_create_dynamic (MPI_COMM_WORLD, &d-win);
MPI_Win_lock_all (d-win);
if (sharedmem_comm!=MPI_COMM_NULL){
   MPI_Win_allocate_shared
   (nbytes,..., &membase, &sharedmem_win);
   MPI_Win_lock_all (sharedmem_win);}
MPI_Win_attach (d-win, membase, nbytes);
MPI_Get_address (membase, &disp);
disp_s = (MPI_Aint*)malloc (size * (sizeof(MPI_Aint)));
MPI_Allgather (&disp, 1, MPI_AINT, disp_s,..., MPI_COMM_WOLRD);
dest = randomdst_generator ();
/*The array is_shmem indicates the position of the given target process with respect to the origin*/
if ((j=is_shmem[dest])>=0){//on-node communication
   /*The j is the relative target rank in the corresponding sharedmem_comm*/
   MPI_Win_shared_query (sharedmem_win, j,..., &baseptr);
   memcpy (baseptr, srcptr, nbytes);}
else{//across-node communication
   MPI_Put (srcptr, nbytes, dest, disp_s[dest],..., d-win);
   MPI_Win_flush (dest, d-win);}
MPI_Win_detach (d-win, membase);
if (sharedmem_comm!=MPI_COMM_NULL){
   MPI_Win_unlock_all (sharedmem_win);}
MPI_Win_unlock_all (d-win);
MPI_Win_free (&sharedmem_win);
MPI_Win_free (&d-win);
free (disp_s);
MPI_Finalize ();
```

Figure 5: An example for MPI code. This example shows the MPI code for transferring data from an origin process to another random target process with data locality in mind.

epoch start/end operations should be dealt with explicitly and carefully. In the end, programmers still need to take care of the resource reclaims, such as unlock and window free operations. In comparison to the MPI code, the DART code is obviously more concise and easier-to-read. This is due to that DART-MPI presents easy-to-use interfaces for the user through internally encapsulating all MPI-related complexities and handling for process topology and locality-awareness. According to this code comparison, we can clearly observe that with the DART interfaces, higher programmer productivity can be achieved compared to purely with the MPI interfaces.

However, in order to hide the MPI details inside the DART interfaces, extra data structures or functions are required to be created to meet the definition of our DART concept without violating the MPI semantics. They will in turn influence the DART-MPI performance to varying extents. Basically, inside *dart_init* we establish a hash table to store the information on the process topology, with which we can in the future obtain the relative distance between two communication parties.

Such initiation operation, as an entry point, is only called just once to setup the environment for the following DART routines. Besides, while invoking the DART global memory allocation operation, a MPI window spanning the given team will be created, which is not cheap. However, once a region of DART global memory is created, multiple RMA operations can be performed on it until it is deallocated. In this sense, the overhead of creating a region of global memory can be amortized by those communication operations happening on it. Theoretically, the DART RMA interfaces (data movement operations) are most likely to be massively invoked, where the operation of dereferencing global pointer will play a critical role in contributing to the DART implementation overhead over MPI. Detailedly, first we search the hash table (created in *dart_init*) to indicate the distance of the two communication sides, which determines intra-node or inter-node data movement. We then adopt a specific communication path for each of them with the help of translation table, which is thus looked up inevitably to get the correct parameters identified by MPI semantics. Apparently, the table lookup operations will frequently occur and are eventually aggregated to contribute to the DART implementation overhead. Therefore, we should estimate their cost and observe how this cost affects the DART RMA performance. Additionally, in the evaluation below (refer to Sect. 5.2) the cost of table lookup operations is interpreted as the DART implementation overhead.

5 Experimental evaluation

In this section we present all performance evaluation experiments and analyze those results thoroughly. In Sec. 5.2 we measure the overhead of DART-MPI RMA operations against directly using MPI shared-memory extensions. In the following Sec. 5.3 we compare the performance of the heat-diffusion problem implemented with DART-MPI, flat MPI, and locality-aware MPI, respectively.

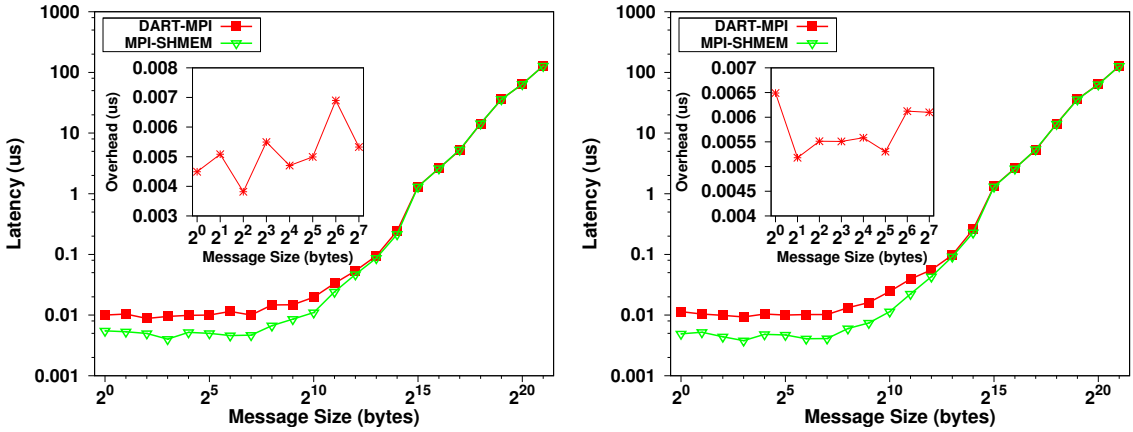
Foremost, DART system internally accounts for the cost of process topology identification and look-ups on the translation table (refers to Sect. 4). Such cost estimate (aka. DART implementation overhead evaluation) will be of great significance to enhance the usage of DART-MPI as an efficient as well as portable runtime system. This benchmark is performed based on OSU Micro Benchmark [12] and averaged over 10000 executions. We do not show the error bars for their plots, as the standard deviation from the mean is always relatively small. We then present a comprehensive picture of the comparative results between DART-MPI RMA and *flat* MPI RMA (blocking and non-blocking) through a numerical simulation of 3D heat conduction problem. In this experiment, we stop the calculation after 5000 iterations and collect and analyze the time results of the computation (update the inner grid cells) and halo cells exchange for different grid sizes and participating processes. We plot the average data results of 25 runs with small execution time variation reported. Further, we show in Section 5.3.2 the influence of the overhead of the implementation overhead on the DART-MPI blocking RMA version of this application relative to the locality-aware MPI blocking RMA version of this application.

5.1 Experimental testbed

The experimental environment was the Cray XC40 system, named Hazel Hen at HLRS, on which the benchmarks mentioned above are carried out. Hazel Hen system is equipped with 7,712 compute nodes made up of dual twelve-core Intel Haswell E5-2680v3 processor (one processor per socket), which has exclusive 256 KB L2 unified cache for each core. Therefore, each compute node has 24 cores running at 2.5GHZ with 128 GB of DDR4 (Double Data Rate) RAM (Random Access Memory). Additionally, a compute node is regarded as a NUMA system where each processor forms a NUMA domain. The different compute nodes are interconnected via a Cray Aries network using Dragonfly topology. The Hazel Hen system features a hierarchical network topology. Detailedly, the Cray XC Rank-1 network is used for communications happening across different compute blades within a chassis over backplane. Each chassis consists of 16 compute blades. Communications happening across different chassis within a 2-cabinet group over copper cables are supported with the Cray XC Rank-2 network. Further, the Cray XC Rank-3 network is employed for communications happening between the groups over the optical cables. Note that, the higher hierarchy the network reaches, the stronger performance penalty we probably gain. This is due to that every extra level added to

the network will increase the communication latency by at least one hop. I.e., the communications over greater distance will exacerbate the performance of message transfers. Moreover, the tasks are assigned to cores in SMP style [13], which means one node needs to be filled before going to next.

5.2 The DART-MPI implementation overhead



(a) Blocking put (intra-node)

(b) Blocking get (intra-node)

Figure 6: Latency comparisons of intra-node blocking put/get using two kinds of tests written with DART interfaces and MPI-3 shared-memory extensions. "DART-MPI" indicates DART blocking RMA operations. "MPI-SHMEM" indicates memory copy operations with the usage of MPI-3 shared-memory window.

The data movements indicated by the DART intra-node blocking RMA operations are performed through shared memory mapping, which leads to very low latency. Besides, data transfers happening within one node will not be disturbed by the external factor, such as network traffic. In this regard, the performance of DART intra-node blocking RMA communication operation is more sensible to the implied implementation overhead compared to inter-node operation. Therefore, in this section, we estimate the DART implementation overhead by comparing the DART intra-node blocking RMA interfaces against the MPI-3 shared-memory extensions.

Figure 6 preliminarily demonstrates the DART implementation overhead in a way that compares the DART intra-node blocking RMA operations and MPI shared memory window-based memory copy operations (refer to the outer plot) in an ideal environment (on-node) and quantifies the gaps between them (refer to the inner plots). The implementation overhead, defined as the difference of the execution times, is clearly shown in the inner plots. Specifically, observing the inner plot in Fig. 6a, we can find that the overhead for DART intra-node blocking put implementation is small at around 5 ns. Similarly, DART intra-node blocking get implementation also has a small overhead at around 5 ns in general. Thus, the absolute value of the DART overhead with respect to direct load/store operations is small and relatively independent of message size. For message sizes above roughly 2^{12} bytes, the overhead is negligible compared to the execution time of the data transfer as shown by the outer plots in Fig. 6. Also, the impact of the implementation overhead is likely to be substantially softened in a more realistic communication environment, where multiple intra-node or inter-node data transfers exist.

5.3 DART and MPI RMA version of 3D heat conduction application

For our test, we adapted the 3D heat conduction implementation to DART one-sided directives with two flavors of blocking and non-blocking. The halo cells exchange is achieved by using get

```
/* Communicate matrix data on Oxz plane */
if (South neighbor exists)
  for (i = 0; i < xcell; i++)
    get (zcell, South); // Get zcell consecutive grid cells from South neighbor
barrier;
if (North neighbor exists)
  for (i = 0; i < xcell; i++)
    get (zcell, North);
barrier;
/* Communicate matrix data on Oyz plane */
if (West neighbor exists)
  for (i = 0; i < ycell; i++)
    get (zcell, West); // Get zcell consecutive grid cells from West neighbor
barrier;
if (East neighbor exists)
  for (i = 0; i < ycell; i++)
    get (zcell, East);
barrier;
/* Communicate matrix data on Oxy plane */
if (Up neighbor exists)
  for (i = 0; i < xcell; i++)
    for (j = 0; j < ycell; j++)
      get (1, Up); // Get one grid cell from Up neighbor
barrier;
if (Down neighbor exists)
  for (i = 0; i < xcell; i++)
    for (j = 0; j < ycell; j++)
      get (1, Down);
barrier;
```

Figure 7: Pseudo-code for the halo cells exchange with get and barrier operations.

operation. We can deduce that six get operations will be invoked for each process. Figure 7 concisely shows the critical part of the halo cells exchange code within each calculation iteration. Here, each get operation refers to either blocking get or non-blocking get. *dart_get_blocking* is used in DART blocking RMA version. In DART non-blocking RMA version, we invoke *dart_get*, where the outstanding communication operations are finished in bulk with *dart_waitall*. Within each calculation iteration, after one round of halo cells exchange with certain neighbor an explicit process synchronization, such as barrier, is naively inserted in our code due to an implicit synchronization is ideally supported by point-to-point communication model. Therefore, the halo cells exchange time measured below includes the overhead brought by process synchronization. Regarding the process synchronization, we only measure the overhead introduced by the barrier operations instead of the time to wait between processes.

We also implement this 3D heat conduction algorithm (see Section 5.3.1) simply based on MPI one-sided interfaces with the above two flavors using passive target mode as the memory synchronization mechanism for a fair comparison. Particularly, *MPI_Rget* is invoked first and then closely followed by *MPI_Wait* in flat MPI blocking RMA version. On the other hand, the *MPI_Get* is employed in flat MPI non-blocking RMA version, where uses *MPI_Win_flush* to guarantee the local and remote completeness. Note that, the asynchronous progression is disabled for each of the non-blocking RMA versions. This flat MPI RMA implementation only invokes the RMA operations pertaining to the RMA window instead of shared memory window and thus does not take the data locality into consideration. Accordingly, we will observe the obvious performance benefit below (see Sect. 5.3.1) brought by the DART-MPI RMA operations. Further, in Section 5.3.2 we embark on comparing DART-MPI and locality-aware MPI RMA (blocking) version of the 3D heat conduction application, where the latter coordinates the RMA operations associated with the RMA window and shared-memory window. We then can obtain the performance impact of the DART-MPI implementation overhead.

5.3.1 Performance comparison between DART-MPI and *flat* MPI RMA version

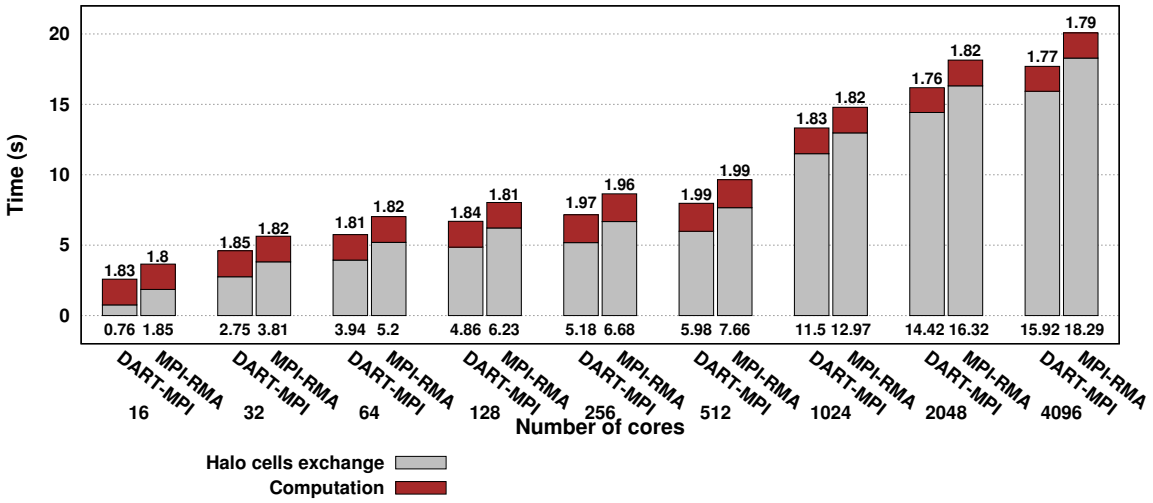


Figure 8: Weak scaling for blocking one-sided DART and MPI. The grey bar signifies the halo cells exchange time, the value of which is written on the bottom. The brown bar signifies the computation overhead, the value of which is written on the top.

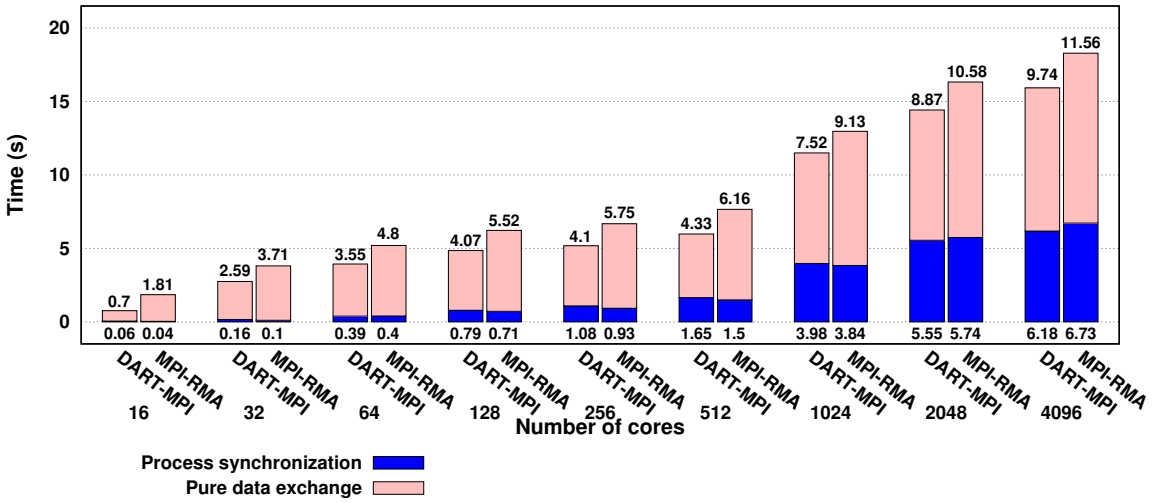


Figure 9: Breakdown of halo cells exchange time for the weak scaling problem. The blue bar signifies the overall process synchronization overhead, the value of which is written on the bottom. The pink bar signifies the pure data exchange overhead, the value of which is written on the top. The sum equals to the halo cells exchange overhead that we measured in this benchmark.

Figure 8 gives the quantitative results showing the performance of DART-MPI blocking RMA version and flat MPI blocking RMA version on Cray XC40 system over the number of cores (from

16 to 4096). A weak scaling evaluation, where the number of grid cells per core is fixed (32K, 1K denotes 1024) and is applied with grid sizes varying from (64×64×128) to (256×2048×256). Each cell is a 8-byte double precision floating point number. In this figure, "DART-MPI" is the DART-MPI RMA implementation of application. "MPI-RMA" is the flat MPI RMA implementation of application, which means that it does not take the data locality into consideration. The meaning of them can also be applied to the following plots. The problem size is chosen such, that overall execution is dominated by the time to exchange halo cells in order to emphasize the statistical significance of the latter. Basically, the computation time almost keeps constant over the number of cores, as was expected. As expected also, the time to update the inner grid cells in DART and flat MPI implementation is pretty much the same. When only the on-node performance is considered, that is 16 cores, the improvement in the halo cells exchange time can be obviously seen, *cf.* 0.76s for DART-MPI, and 1.85s for flat MPI. This is mostly attributed to the enabling of direct load/store access within one node in the DART implementation. On the whole, the DART implementation can provide a relative speedup of ∼1.38x over flat MPI on average in terms of halo cells exchange time performance. Such speedup is expected because most of the data exchanges still happen within one node in this evaluation. This fact highlights and visualizes the advantage of the memory sharing mechanism in intra-node case employed by DART implementation. Besides, we can observe that the performance speedup gets decreased as the number of cores is increased, which is not surprising given the number of across-node data exchanges is accordingly increased and DART blocking RMA uses the same communication infrastructure as MPI RMA internally in the inter-node case.

Moreover, shown in Fig. 8, the halo cells exchange time get sustained growth over the number of cores. Specifically, a sudden rise occurs when the communications go beyond one node and start crossing nodes (32 cores). The fact, that is, several inter-node data transfers start to be involved accordingly, creates big difference in performance. Besides that, the process synchronization may also be a part of the growth of the halo cells exchange time, as hinted above. Therefore, Figure 9 breaks down the halo cells exchange time in terms of pure data exchange and process synchronization, which can help us understand how the two components affect the amount of time the halo cells exchange takes to run. Observing this figure, it is immediately clear that both the process synchronization and pure data exchange overhead increase gradually over the number of cores. The breakdown information tells us that the pure data exchange component plays an important role in the halo cells exchange time rise especially when the number of cores reaches up to 1024. In this experiment, inter-group communications over the greatest distance take place when 1024 cores are launched, which would greatly degrade data exchange performance, in comparison to the inter-blade (intra-chassis) communications and inter-chassis (intra-group) communications. This is the reason for the sudden rise in the pure data exchange time when ranging the number of cores from 512 to 1024. Here, we would add that the Fig. 9 sufficiently illustrates that the data locality-aware implementation of DART RMA results in the improvement in the halo cells exchange time.

Next, a weak scaling test on Cray XC40 system is undertaken to study the scalability of MPI and DART non-blocking RMA communication operations. The number of cores is also increased from 16 to 4096. From Figure 10 we see the comparative results showing the performance of DART-MPI non-blocking RMA version and flat MPI non-blocking RMA version with problem sizes varying from (32×32×512) to (256×256×2048). Each core gets a sub-grid with 32K cells after decomposition. The value of *zcell* is kept as 512 in this test. Revisiting Fig. 7, we can observe that the data movements in the direction of east-west and north-south involve multiple get operations, each of which carries 512 numbers with double-precision floating pointer format (each message size is 4KB). For message size of 4KB, both *dart_get* and *MPI_Get* employ the eager protocol, which can potentially support the communication-computation overlapping. As a result, we measure the message transmission time that the CPUs are engaged in, which is also understood as the latency of the halo cells exchange shown in Fig. 10. The supported overlap weakens the advantage of harnessing locality-awareness for DART-MPI because the communication latency will be mostly hidden by the ongoing computation task. This, coupled with the DART implementation overhead lead to a phenomenon that the DART-MPI non-blocking RMA version performs slightly worse than the flat MPI non-blocking RMA version up to a point when the number of cores reaches 256. As the number of cores is increased, we can accordingly obtain smaller proportion of the data transmission that is overlapped

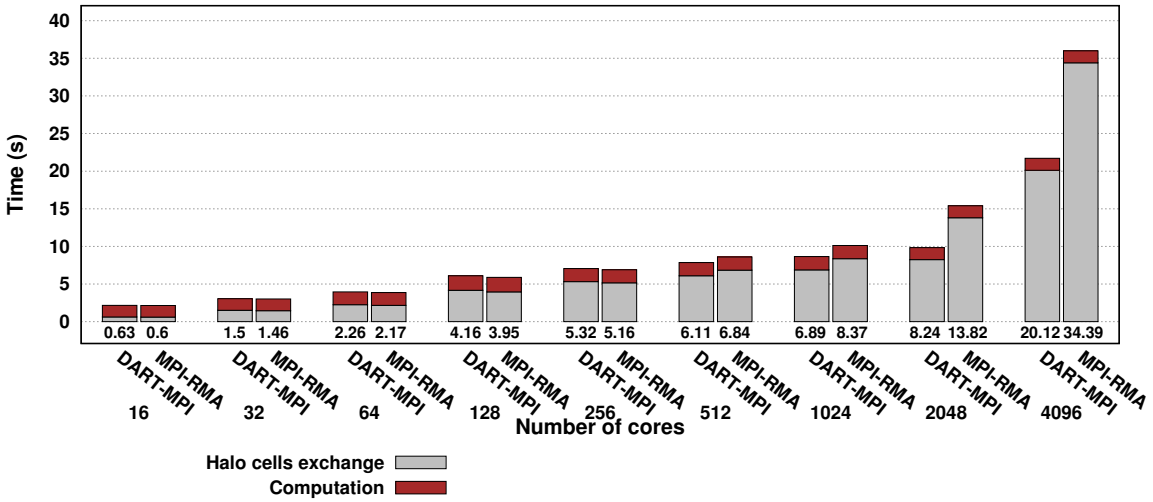


Figure 10: Weak scaling for non-blocking one-sided DART and MPI. The grey bar signifies the halo cells exchange time that the CPU engages in, which is not overlapped with the previous computation task. Refer to the Fig. 8 for the remaining annotations.

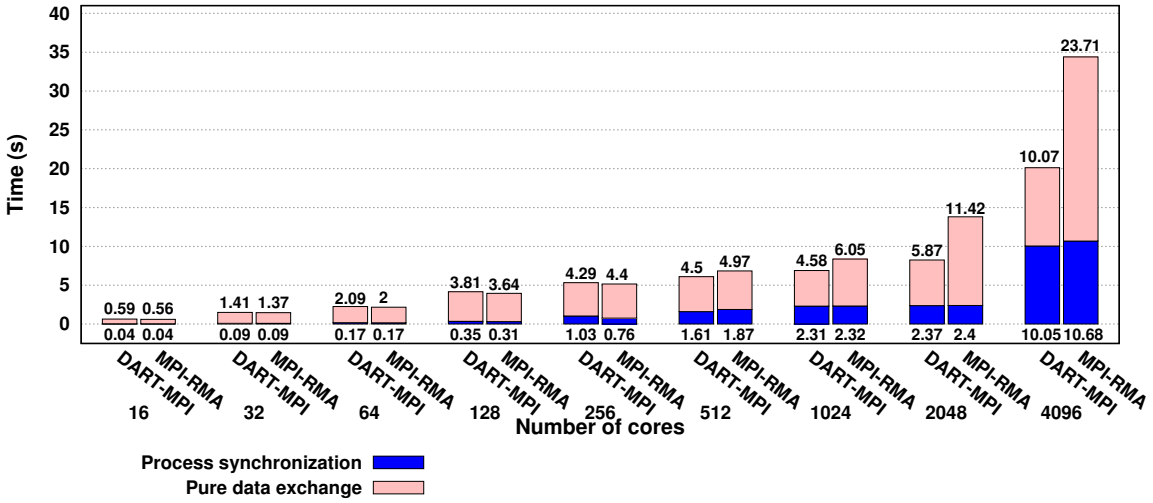


Figure 11: Breakdown of the halo cells exchange time for the weak scaling problem. Refer to Fig. 9 for the annotations.

with the previous workload and further the halo cells exchange plays a greater role in contributing to the total execution time. In this regard, the margin of improvement using the *shmem-win* becomes larger which corresponds to the lesser halo cells exchange time of DART-MPI non-blocking RMA version compared with the flat MPI non-blocking RMA version in Fig. 10. Specifically, DART-MPI implementation can consistently show less execution time than the flat MPI implementation with slightly increased speedup (from 1.12 to 1.8) for number of cores not less than 256. The supported overlap also explains the fact that the DART non-blocking RMA version performs sightly better

than the DART blocking version at some points. The conductions of this experiment and the last experiment concerning the blocking RMA implementations are already a few months apart, where the softwares installed on the machine get involved (e.g., the version of Cray-MPI is upgraded from 7.3.3 to 7.5.0). This can help to justify the execution time difference between them. Additionally, a micro-benchmark evaluation (refer to [14]) demonstrates that *dart_get* shows better time performance than *MPI_Get* for long messages when only the data transmission is considered (i.e., pure data transmission time).

Ideally, the scaled-up workload can be solved in a weak scaling evaluation at a fixed execution time. Practically, shown from the Fig. 10, the execution time however grows gradually except when the number of cores is increased from 2048 to 4096. According to the explanation for the results of the blocking RMA implementation, we learn that the increase in time is mainly attributed to two factors – pure data exchange operation and process synchronization, where the pure data exchange overhead is closely influenced by the associated network hierarchy. Figure 11 breaks down the halo cells exchange time in terms of these two factors. It is clearly observed that the pure data exchange overhead accounts for the sudden time increase at the point of 4096 cores. In this trail, two more groups are arranged to place all the participating processes (two groups are reserved for the 2048 cores) when the number of cores reaches 4096. This means, that there will be twice the inter-group communications over the largest distance injected into the network, which will accordingly degrade the performance of the halo cells exchange.

5.3.2 Performance impact of DART-MPI implementation overhead

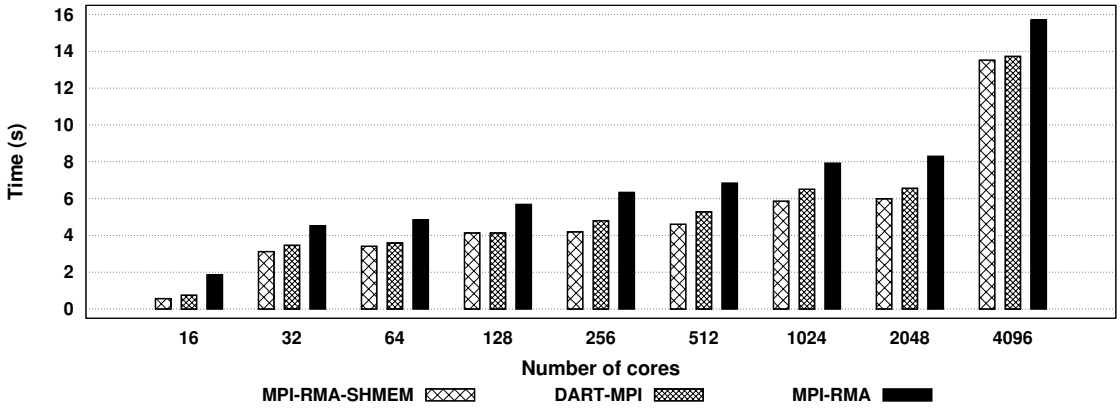


Figure 12: Halo cells exchange time performance comparison of three versions of 3D heat conduction application. "MPI-RMA-SHMEM" is the locality-aware MPI RMA version, "DART-MPI" is the DART-MPI RMA version, "MPI-RMA" is the flat MPI RMA version.

The DART-MPI implementation overhead measured in Sec. 5.2 is represented as the time performance gap concerning a single put/get operation. Such implementation overhead, and that of some other DART interfaces, will then be accumulated in a full application. This necessarily leads to a performance which is lower than a functionally equivalent, fully optimized and locality-aware MPI version. In order to minimize the effect of measurement variability, we disregard the computation time and only study the time performance of halo cells exchange, which consists of multiple get operations (see Fig. 7). Therefore, we investigate further into the influence of the accumulated DART-MPI implementation overhead on the halo cells exchange time performance of the above DART-MPI blocking RMA version of 3D heat conduction application (denoted as $T_{DART-MPI}$), versus that of the locality-aware MPI blocking RMA version (denoted as $T_{MPI-RMA-SHMEM}$). Plus, the halo cells exchange time performance of the flat MPI blocking RMA version is denoted as $T_{MPI-RMA}$.

Figure 12 shows the performance impact in a way of comparing $T_{DART-MPI}$ to $T_{MPI-RMA}$ and $T_{MPI-RMA-SHMEM}$, respectively. This experiment is executed recently and thus the results regarding the DART-MPI blocking RMA version and flat MPI blocking RMA version do not match with those shown in Fig. 8, which was done months ago. First, from this figure it is intuitively obvious that the adverse performance impacts exist (refer to the gap between $T_{DART-MPI}$ and $T_{MPI-RMA-SHMEM}$), but they are low enough to be accepted when the significant performance benefits over the flat MPI blocking RMA version (refer to the gap between $T_{DART-MPI}$ and $T_{MPI-RMA}$) are considered. Second, we quantitatively evaluate the performance overhead by studying deeper into the ratios of $T_{DART-MPI}$ and $T_{MPI-RMA-SHMEM}$. The ratios always sit below 1.2 once the number of cores is beyond 24 (across-node data transfers begin to emerge). In addition, the ratios steadily decrease when the number of cores is ranged from 512 up to 4096, at which point the ratio approaches 1. Such low impact of overhead can enhance the usage of DART-MPI in the large scale distributed-memory system. Theoretically, assuming the number of cores continually grows, the diminishing of the impact of overhead proceeds.

6 Related work

A heuristic study [15] conducted on Cray XE6 shows that one-sided MPI-2 has relatively inferior scaling behavior compared to UPC and Cray SHMEM RMA. This is due to that random allocated memory can be specified to be remotely accessible in MPI-2. Unlike MPI-2, the memory for RMA operation is somehow customized for Cray SHMEM (symmetric memory) and UPC (shared memory). Further, MPI-3 RMA enables direct load/store accesses by supporting a shared-memory window [3, 4], which is employed in DART to enhance the intra-node RMA communication performance.

Additionally, MPI researchers conduct long-term optimization works on the MPI RMA [16–19] and collective operations [20–22]. Those optimizations are also liable to be applied onto our DART and then benefit the performance of applications.

Like MPI, DART one-sided communication operations revolve around describing data movement (*dart_get/put*) and memory synchronization (*dart_wait/waitall*). However, the synchronization among processes is often achieved via heavy-weight synchronization mechanisms, such as setting a barrier or explicitly sending a control message in the form of two-sided. Such synchronization mechanisms fail to efficiently implement producer-consumer pattern due to non-negligible overhead along with them. Therefore, the devise of a light-weight synchronization scheme notifying the remote side as soon as the remote completion for one-sided DART could potentially benefit the user applications featured with the producer-consumer pattern [23].

7 Discussion

MPI is widely used in high-performance computing (HPC) today due to its high performance, scalability, and portability. Certainly, DART-MPI can easily get the interoperability with the existing MPI applications since it takes MPI as the underlying communication conduit. The interoperability can be achieved by adding some interactive interfaces, with which the programmers can seamlessly switch between MPI and DART operations.

DART-MPI aims at providing locality-aware as well as simple RMA interfaces for the programmers. In order to achieve this, all the MPI complexities are encapsulated inside DART-MPI system. Programming locality-aware HPC applications will then be facilitated with DART-MPI. Although DART-MPI RMA implementation overhead with respect to the locality-aware MPI code with shared-memory supported exists inevitably, a marginal overhead and relatively low impact of this overhead are shown in the above experiments. If the programmers insist in writing locality-aware MPI code to eliminate the extra overhead, they should be burdened with the MPI complexities and maintenance of the verbose code, which is non-trivial. In this regard, the application productivity is certain to be impaired when we are only concerned with the requirement of writing locality-aware MPI code. Here, we quantitatively evaluate the application productivity by revisiting the DART-MPI blocking

RMA and locality-aware MPI blocking RMA versions of 3D heat conduction application mentioned in Sec. 5.3.2. We learn that the DART-MPI blocking RMA version shows comparable performance to the locality-aware MPI blocking RMA version from Fig. 12. However, the code lines for the locality-aware MPI blocking RMA version will be increased by 96, with respect to that for the DART-MPI blocking RMA version. Besides that, the locality-aware MPI version needs to manually control and maintain two window objects and the number of increased code lines grows linearly as the number of the sub-communicators. With DART-MPI, the improvement in the application productivity can be consistently gained. Plus, shown in Fig. 8, the DART-MPI blocking RMA version speeds up the communication performance by 27% on average over the flat MPI blocking RMA version. Therefore, DART-MPI is an optimal option for implementing HPC applications after weighing the application productivity against performance.

Besides DART-MPI, hybrid programming model emerges as another way to cater for the hierarchical hardware architecture in multi-/many-core clusters. The hybrid programming model encourages the combination of different programming models to address the issue of locality awareness. The most familiar combination is MPI and shared-memory model (such as Pthreads and OpenMP), where shared-memory model serves the on-node data communications and MPI serves the across-node data communications (nodes are connected by network). This model can provide flexibility and performance improvement potential for programmers. Meanwhile though the programmers are confronted with challenges of efficiently applying the two mixed programming models and guaranteeing a thread-safe MPI program [24] as well.

8 Conclusion

We have analyzed the approaches of applying the MPI RMA synchronization mechanisms (global shared lock or single shared lock) to the related DART interfaces. Comparatively, using the global lock mechanism facilitates the epoch pattern as well as retains the DART RMA performance. Due to that DART-MPI has internally taken over the responsibility for the locality-awareness by hiding the complexity from the user, the DART code of data locality-awareness is more concise and easier-to-read than MPI code and thus higher programmer productivity can be achieved by using DART-MPI.

We have evaluated the performance behaviors exhibited by DART-MPI RMA using a communication micro-benchmark and a 3D heat conduction problem. The micro-benchmark results show that the DART-MPI implementation overhead is small enough that we can ignore it for large message sizes. We then adapted the 3D heat conduction application onto flat one-sided MPI and DART with two flavors of blocking and non-blocking. The DART-MPI blocking RMA version is proved to be efficient for different number of cores and grid sizes on Cray XC40 system and significantly improves the communication performance (halo cells exchange), compared to the flat MPI blocking RMA version in this application level evaluation. Additionally, DART-MPI non-blocking RMA version is comparable to flat MPI non-blocking RMA version to a lesser extent. Also, as the number of cores increases, DART-MPI non-blocking RMA shows higher potential to achieve scalable performance than flat MPI non-blocking RMA version does. Such performance improvement is induced by the fact that one-sided DART is implemented using the feature of MPI-3 integrated shared memory window in the intra-node case while retains the MPI inter-node RMA performance. Moreover, we evaluate the performance impact of the DART-MPI implementation overhead on the DART-MPI blocking RMA version of 3D heat conduction problem, versus the locality-aware MPI blocking RMA version. The observed low impact, especially when the across-node data transfers happen, further proves that DART-MPI has reached the goal of balancing application productivity and performance.

9 Acknowledgement

This work has been supported by the European Community through the project Mont Blanc 3 (H2020 programme under grant agreement number 671697). We gratefully acknowledge funding by the German Research Foundation (DFG) through the German Priority Programme 1648 Software for Exascale Computing (SPPEXA) and the project DASH.

References

[1] MPI Forum, "MPI: A Message-Passing Interface Standard. Version 3.0," Tech. Rep., Sep. 2012, available at: http://www.mpi-forum.org.

[2] "TOP500 - List Statistics - Nov. 2016," http://www.top500.org/statistics/list/, accessed: Jun. 2016.

[3] T. Hoefler, J. Dinan, D. Buntinas, P. Balaji, B. W. Barrett, R. Brightwell, W. Gropp, V. Kale, and R. Thakur, "Leveraging MPI's One-Sided Communication Interface for Shared-Memory Programming." in *EuroMPI*, ser. Lecture Notes in Computer Science, J. L. Träff, S. Benkner, and J. J. Dongarra, Eds., vol. 7490. Springer, 2012, pp. 132–141.

[4] J. R. Hammond, S. Ghosh, and B. M. Chapman, "Implementing OpenSHMEM Using MPI-3 One-Sided Communication." in *OpenSHMEM*, ser. Lecture Notes in Computer Science, S. W. Poole, O. R. Hernandez, and P. Shamis, Eds., vol. 8356. Springer, 2014, pp. 44–58.

[5] H. Zhou, Y. Mhedheb, K. Idrees, C. W. Glass, J. Gracia, and K. Fürlinger, "DART-MPI: An MPI-based Implementation of a PGAS Runtime System." in *PGAS*, A. D. Malony and J. R. Hammond, Eds. ACM, 2014, pp. 3:1–3:11.

[6] H. Zhou, K. Idrees, and J. Gracia, "Leveraging MPI-3 Shared-Memory Extensions for Efficient PGAS Runtime Systems." in *Euro-Par*, ser. Lecture Notes in Computer Science, J. L. Träff, S. Hunold, and F. Versaci, Eds., vol. 9233. Springer, 2015, pp. 373–384.

[7] K. Fürlinger, C. W. Glass, J. Gracia, A. Knüpfer, J. Tao, D. Hünich, K. Idrees, M. Maiterth, Y. Mhedheb, and H. Zhou, "DASH: Data Structures and Algorithms with Support for Hierarchical Locality," in *Euro-Par 2014: Parallel Processing Workshops - Euro-Par 2014 International Workshops, Porto, Portugal, August 25-26, 2014, Revised Selected Papers, Part II*, 2014, pp. 542–552.

[8] H. Zhou and J. Gracia, "Towards Performance Portability through Locality-Awareness for Applications Using One-Sided Communication Primitives." in *CANDAR*. IEEE, 2016, pp. 536–542. [Online]. Available: http://ieeexplore.ieee.org/xpl/mostRecentIssue.jsp?punumber=7818132

[9] M. Williams, "What is heat conduction?" http://phys.org/news/2014-12-what-is-heat-conduction.html, Dec. 2014.

[10] "MPI Parallelization for numerically solving the 3D Heat equation," https://dournac.org/info/parallel_heat3d, accessed: Apr. 2016.

[11] R. Thakur, W. D. Gropp, and B. R. Toonen, "Minimizing Synchronization Overhead in the Implementation of MPI One-Sided Communication." in *PVM/MPI*, ser. Lecture Notes in Computer Science, D. Kranzlmüller, P. Kacsuk, and J. Dongarra, Eds., vol. 3241. Springer, 2004, pp. 57–67.

[12] "OSU Micro-Benchmarks," http://mvapich.cse.ohio-state.edu/benchmarks, 2014.

[13] "Reordering MPI Ranks," http://www.nersc.gov/users/computational-systems/retired-systems/hopper/performance-and-optimization/reordering-mpi-ranks/, accessed: Apr. 2016.

[14] H. Zhou and J. Gracia, "Asynchronous Progress Design for a MPI-Based PGAS One-Sided Communication System." in *ICPADS*. IEEE, 2016, pp. 999–1006. [Online]. Available: http://ieeexplore.ieee.org/xpl/mostRecentIssue.jsp?punumber=7822825

[15] C. M. Maynard, "Comparing One-sided Communication with MPI, UPC and SHMEM," in *Proceedings of the Cray User Group (CUG)*, 2012.

[16] J. Liu, W. Jiang, P. Wyckoff, D. K. Panda, D. Ashton, D. Buntinas, W. D. Gropp, and B. R. Toonen, "Design and Implementation of MPICH2 over InfiniBand with RDMA Support." in *IPDPS*. IEEE Computer Society, 2004.

[17] P. Lai, S. Sur, and D. K. Panda, "Designing Truly One-sided MPI-2 RMA Intra-node Communication on Multi-core Systems." *Computer Science - R&D*, vol. 25, no. 1-2, pp. 3–14, 2010.

[18] R. Gerstenberger, M. Besta, and T. Hoefler, "Enabling Highly-scalable Remote Memory Access Programming with MPI-3 One Sided." in *SC*, W. Gropp and S. Matsuoka, Eds. ACM, 2013, pp. 53:1–53:12.

[19] S. Potluri, H. Wang, V. Dhanraj, S. Sur, and D. K. Panda, "Optimizing MPI One Sided Communication on Multi-core InfiniBand Clusters Using Shared Memory Backed Windows." in *EuroMPI*, ser. Lecture Notes in Computer Science, vol. 6960. Springer, 2011, pp. 99–109.

[20] R. Thakur, R. Rabenseifner, and W. Gropp, "Optimization of Collective Communication Operations in MPICH." *IJHPCA*, vol. 19, no. 1, pp. 49–66, 2005.

[21] J. Liu, A. R. Mamidala, and D. K. Panda, "Fast and Scalable MPI-Level Broadcast Using InfiniBand's Hardware Multicast Support." in *IPDPS*. IEEE Computer Society, 2004.

[22] A. R. Mamidala, R. Kumar, D. De, and D. K. Panda, "MPI Collectives on Modern Multicore Clusters: Performance Optimizations and Communication Characteristics." in *CCGRID*. IEEE Computer Society, 2008, pp. 130–137.

[23] R. Belli and T. Hoefler, "Notified Access: Extending Remote Memory Access Programming Models for Producer-Consumer Synchronization." in *IPDPS*. IEEE Computer Society, 2015, pp. 871–881.

[24] W. Gropp, "MPI, Hybrid Programming, and Shared Memory ," Tech. Rep., 2014.