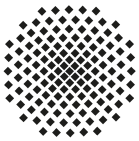


H L R S

Institut für
Höchstleistungsrechnen

FORSCHUNGS- UND ENTWICKLUNGSBERICHTE

COMMUNICATION METHODS FOR HIERARCHICAL GLOBAL ADDRESS SPACE MODELS IN HPC

Huan Zhou

ISSN 0941 - 4665 Dezember 2016 HLRS-15

Höchstleistungsrechenzentrum
Universität Stuttgart
Prof. Dr.-Ing. Dr. h.c. Dr. h.c. Prof. E.h. M. M. Resch
Nobelstrasse 19 - 70569 Stuttgart
Institut für Höchstleistungsrechnen

COMMUNICATION METHODS FOR HIERARCHICAL GLOBAL ADDRESS SPACE MODELS IN HPC

von der Fakultät Energie-, Verfahrens- und Biotechnik der Universität Stuttgart zur Erlangung der Würde eines Doktor-Ingenieurs (Dr.-Ing.) genehmigte Abhandlung

vorgelegt von

Huan Zhou
aus Anhui / China

Hauptberichter:	Prof. Dr.- Ing. Dr. h.c. Dr. h.c. Prof. E.h. Michael M. Resch
Mitberichter:	Prof. Dr. Edgar Gabriel
Tag der Einreichung:	25.05.2016
Tag der mündlichen Prüfung:	16.12.2016
CR-Klassifikation:	I.3.2, I.6.6

D93

ISSN 0941 - 4665 Dezember 2016 HLRS-15

Abstract

Simulation is becoming increasingly important in the field of engineering science and viewed as an extension of the traditional theoretical and experimental investigation. However, the common obstacles we are facing in the large-scale simulations are the large volumes of data sets to be processed and the computational complexity of the methods. To simulate the large-scale problems within reasonable time, High Performance Computing (HPC) techniques are needed. One method to speeding up computational execution time is parallel processing. I.e., a single simulation program is distributed over multiple processes. Various parallel programming models are developed for writing efficient parallel applications to perform the distributed simulation on parallel computer hardware. The Message Passing Interface (MPI) remains as the dominant communication model as a result of its high performance, portability and standardization on cutting-edge computing systems. However, writing an efficient MPI application is challenging. The emergence of multi- and many-core processors fuels an explosive growth of processing capability in current-generation high-end systems. In this sense, communication speed becomes a limiting factor in the distributed simulation performance. To alleviate the communication bottleneck, an adequate communication scheme stemming from the existing high performance parallel programming models is needed.

In this dissertation, I explore the approach of designing an efficient communication runtime system on top of MPI-3 Remote Memory Access (RMA) model by hiding the complexity of MPI from the user. To take the hierarchical memory layout of HPC systems into account, I adopt a hybrid strategy which combines the MPI RMA model (for inter-node communications) and the shared-memory programming model (for intra-node communications). Moreover, to achieve higher communication-computation overlap, I design a progress engine by using a process-based approach. The progress engine can start the asynchronous progression of communication independent of MPI synchronization calls on the origin side. I have evaluated the performance improvement offered by the proposed communication scheme using application benchmarks as well as communication micro-benchmarks. The micro-benchmark evaluations demonstrate that the proposed communication runtime system can consistently yield better performance than MPI RMA operations in the intra-node case. Also, it can retain the performance of MPI RMA operations in the inter-node case without adding noticeable overheads. Most importantly, it is demonstrated that the proposed communication scheme can lead to substantial performance increase for real engineering applications. Besides that, the proposed communication runtime system shows a clear performance advantage compared with other low-level RMA libraries in the intra-node case according to the micro-benchmarks and

it can also match those low-level RMA libraries in terms of the application benchmarks involving inter-node data transfers.

Zusammenfassung

Simulation wird immer wichtiger im Bereich der Softwareentwicklung und wird als Erweiterung der traditionellen theoretischen und experimentellen Untersuchung angesehen. Allerdings, wie wir heute wissen, begegnen wir Hürden, wie große Mengen an zu verarbeitenden Datensätzen und die hohe Rechenkomplexität von Methoden. Um diese Simulationen in angemessener Zeit berechnen zu können, werden High Performance Computing (HPC) Techniken benötigt. Eine Methode, um die Rechenzeit zu beschleunigen, ist die parallele Verarbeitung. D. h., ein Simulationsprogramm wird über mehrere Prozesse hinweg verteilt. Verschiedene parallele Programmiermodelle sind für die effiziente parallele Implementierung von Anwendungen entwickelt worden, um die verteilte Anwendung auf paralleler Rechnerhardware auszuführen. Wegen der hohen Leistung, Portabilität und der Standardisierung für modernste Computersysteme wird Message Passing Interface (MPI) als das Kommunikationsmodel angesehen. Auf der anderen Seite ist die Implementierung einer effizienten MPI Applikation herausfordernd. Die Entstehung von Multi- und Many-Core Prozessoren beschleunigt den explosiven Wachstum der Rechenleistung in der aktuellen High-End System Generation. In diesem Fall wird die Kommunikationsgeschwindigkeit ein begrenzender Faktor der Performance in verteilten Simulation. Um diesen Engpass zu mildern, ist ein angemessenes Kommunikationssystem aus der bestehenden Hochleistung Parallele Programmierung vonnöten.

In dieser Dissertation untersuche ich die Vorgehensweise eines effizienten PGAS Laufzeit-Kommunikationssystem auf Basis des MPI-3 RMA Modells. Hierbei wird die MPI Komplexität vor dem Benutzer verborgen. Um das hierarchische Speicherlayout von HPC System mit in Betracht zu ziehen, verwende ich eine hybride Strategie, welche das traditionelle MPI RMA Modell (für Inter-Node Kommunikation) mit der, in MPI integrierte, Shared Memory Parallelisierung (für Intra-Node Kommunication) kombiniert. Um eine höhere Überlappung zwischen Kommunikation und Berechnung zu erreichen, verwende ich den Progress-prozessbasierten Ansatz, dadurch findet der asynchrone Progress der Kommunikation unabhängig von den MPI-Synchronisation statt. Mit Applikation Benchmarks sowie Kommunikation-Mikro-Benchmarks wurden die Leistungssteigerungen des im Rahmen dieser Dissertation entwickelte Kommunikationsschemas ausgewertet. Die Auswertung von Mikro Benchmark zeigt, im Fall von Intra-Node Kommunikation, eine bessere Leistung als MPI RMA. Auch kann gezeigt werden, dass die Leistung für Inter-Node Kommunikation, mit vernachlässigbaren Overhead, im Vergleich zu MPI RMA gehalten werden kann. Der wichtigste Aspekt ist jedoch die erhebliche Leistungssteigerung des vorgeschlagenen Kommunikationsschemas für echte Engineering Anwendungen. Darüber hinaus zeigt im Intra-Node Mirkro-Benchmark Vergleich zu anderen RMA-Bibliotheken das vorgeschlagene Kommunikationsschema einen klaren

Vorteil und kann im Bezug auf Anwendungen auch bei Inter-Node Kommunikation mit den RMA-Bibliotheken mithalten.

Acknowledgements

This work was made possible through the support and love of family members, colleagues and many friends and through the assistance and contribution of collaborators who helped me in my project titled "DASH - Hierarchical Arrays for Efficient and Productive Data-Intensive Exascale Computing".

I am grateful to Prof. Michael Resch of the High Performance Computing Center Stuttgart (HLRS) for all his support and instructive comments that are necessary for accomplishing my dissertation. I would like to thank my advisor Dr. Jose Gracia for his patient guidance during the course of my PhD study. I appreciate the considerable number of time and energy that he has devoted to my dissertation. Not only has he provided me invaluable advices and encouragements that led to the completion of my dissertation, but he also gave me respect and plenty of independence to establish my own identity.

I gratefully acknowledge the funding provided by the German Research Foundation (DFG) through the German Priority Programme 1648 Software for Exascale Computing (SPPEXA).

I am thankful to all the members in the DASH project who have collaborated with me during the past years. I got a lot of insightful advices and detailed feedbacks throughout our enjoyable collaboration. In particular, I am indebted to Dr. Karl Fürlinger for his fruitful discussion and practical advices regarding the completion of my work in this project.

I would like to thank all the colleagues at HLRS, especially our work group members including Dr. Colin W. Glass, Christoph Niethammer, Kamran Idrees, Vladimir Marjanovic. I am gratitude to Mathias Nachtmann for his help with the correction of German abstract. Their thoughts and encouragements have inspired my research enthusiasm. I am also grateful to Stefan Dieterich for his help as a Cray member with the data analysis and Dr. Jing Zhang for her encouragement and patient instructions.

Last but definitely not least, I would like to thank my family and my friend – Hong Yao for their sustained support and care for my PhD study. They encouraged me to come to Germany, and their understanding and love gave me confidence in undertaking the challenges I have encountered during the course of my PhD study.

Contents

List of Figures

List of Tables

Acronyms

AMO	**A**tomic **M**emory **O**peration
APGAS	**A**synchronous **PGAS**
API	**A**pplication **P**rogram **I**nterface
ARMCI	**A**ggregate **R**emote **M**emory **C**opy **I**nterface
CAF	**C**o-**A**rray **F**ortran
CCE	**C**ray **C**ompilation **E**nvironment
CC-NUMA	**C**ache **C**oherent **NUMA**
CC-UMA	**C**ache **C**oherent **UMA**
CMP	**C**hip **M**ulti**P**rocessor
COW	**C**luster **O**f **W**orkstations
CPU	**C**entral **P**rocessing **U**nit
DD	**D**omain **D**ecomposition
DDR	**D**ouble **D**ata **R**ate
FDM	**F**inite **D**ifferential **M**ethod
GA	**G**lobal **A**rray
GASNet	**G**lobal **A**ddress **S**pace **Net**working
GUPS	**G**iga **U**pdates **P**er **S**econd
HPC	**H**igh **P**erformance **C**omputing
HPCC	**HPC C**hallenge
MCS	**M**ellor-**C**rummey and **S**cott
MM	**M**ain **M**emory
MIMD	**M**ultiple **I**nstructions, **M**ultiple **D**ata streams
MISD	**M**ultiple **I**nstructions, **S**ingle **D**ata stream
MPI	**M**essage **P**assing **I**nterface
NUMA	**N**on-**U**nified **M**emory **A**ccess

OpenMP	**Open** **M**ulti-**P**rocessing
PDE	**P**artial **D**ifferential **E**quation
PGAS	**P**artitioned **G**lobal **A**ddress **S**pace
PSCW	**P**ost-**S**tart-**C**omplete-**W**ait
QPI	**Q**uick **P**ath **I**nterconnect
RA	**R**andom **A**ccess
RAM	**R**andom **A**ccess **M**emory
RDMA	**R**emote **D**irect **M**emory **A**ccess
RMA	**R**emote **M**emory **A**ccess
SMB	**S**andia **M**icro-**B**enchmark
SMP	**S**ymmetric **M**ulti-**P**rocessors
SIMD	**S**ingle **I**nstruction, **M**ultiple **D**ata streams
SISD	**S**ingle **I**nstruction, **S**ingle **D**ata stream
STAPL	**S**tandard **T**emplate **A**daptive **P**arallel **L**ibrary
SPMD	**S**ingle **P**rogram, **M**ultiple **D**ata
UMA	**U**nified **M**emory **A**ccess

To my family, friends and advisor

Chapter 1

Introduction

1.1 Motivation

Computer simulation has provided the researchers insights into the complex physical behaviors. Importantly, simulation could be an instrumental extension of the theoretical or experimental science when the events are not detectable or even measurable. However, simulating real-world problems, e.g., aerodynamics, fluid dynamics, molecular dynamics, weather forecasting, thermal and electromagnetic analysis and so on, leads to large consumption in both time and memory. This is induced by the sophisticated simulation algorithm involved and the requirement of the processing of large amounts of data. Obviously, solving those problems serially on a single machine (with limited computer memory) would be impractical since the researchers could risk waiting long time until they finish. Besides, some problems may demand more working memory which is beyond the memory capacity of a single machine. High Performance Computing (HPC) technology likely attracts engineers or scientists to distribute a large simulation across multiple processes and run it in reasonable time.

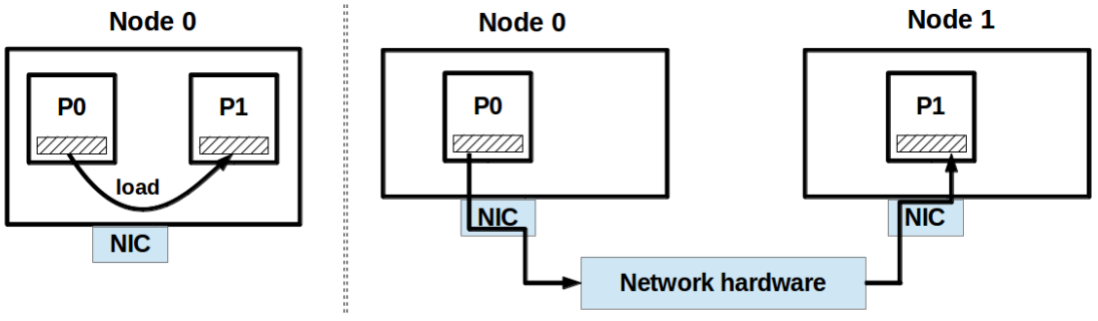


FIGURE 1.1: Communication approach with data-locality awareness. Left: intra-node data transfers are supported with the direct load/store accesses. Right: inter-node data transfers go through network hardware.

HPC technique is a key facilitator of large-scale simulations in science, engineering and business. To enable parallelism, we take a software technique – parallel programming – to shorten the running time of a program by managing multiple independent tasks in the program concurrently. Meantime, this software technique is used to validate the communication and synchronization between different subtasks (induced by data dependencies), for guaranteeing accurate execution of a distributed simulation. The accumulative computing and memory capability in the current parallel hardware systems (e.g., clusters) partly enhances the simulations. Comparatively, communication speed becomes the main impediment to achieving high-speed and scalable distributed simulation performance.

Today, the most popular and leading parallel machines are commodity clusters with at least two-tier hierarchical memory architecture. Accordingly, taking data locality into consideration is entailed in order to write efficient parallel applications. Figure 1.1 shows the data locality-aware communication scheme where different approaches are utilized according to the distance between two communication sides. Partitioned Global Address Space (PGAS, [5]) programming model is intuitively fit into the clusters of hierarchical memory hardware and enhances performance by explicitly exposing data locality to applications. However, most PGAS languages/libraries are implemented based on hardware-specific or vendor-specific communication layers, e.g., Global Address Space Networking (GASNet, [6]) and Aggregate Remote Memory Copy Interface (ARMCI, [7]), supported by the vendors for the particular target platforms. The native implementations of these communication layers are may not optimized for a wide range of interconnects or even not feasible on some systems: Tianhe and K computer. This exacerbates the situation where the scientific applications rely on these PGAS languages/libraries, and let alone there is a urgent need for the high-performance implementations. Recently, there has been much interest in implementing these PGAS languages/libraries based on a highly-portable, yet efficient low-level communication software interface, which is supposed to provide all the functionalities required to implement the languages/libraries. HPC hardware vendors and Message Passing Interface (MPI, [8]) forum put considerable effort to support and optimize MPI implementations for the latest HPC network architectures, leading to the portability to different interconnects. Plus, MPI is rich in efficient functionalities and features. This, coupled with the portability of MPI, contribute to an attractive candidate for an optimal communication layer of the PGAS languages/libraries. Certainly, taking MPI as the underlying communication conduit provides great compatibility/interoperability with the existing MPI applications. It would be better when the performance of MPI is retained.

1.1.1 Existing designs for PGAS runtime systems using MPI

Researchers in the past have put great efforts to explore different approaches of benefiting PGAS models or PGAS-oriented scientific applications using MPI.

In order to incrementally implement scientific applications for PGAS models, an extended runtime supporting both MPI and UPC communications on InfiniBand clusters is proposed by Jose et al. [9]. Although the extended runtime experimentally achieves better results than the pure UPC runtime, it is required to be acted as a new GASNet communication conduit. Therefore, the new design is still tainted by the flaw of relatively-low portability inherent in the UPC runtime – GASNet.

On the other hand, improving the portability of PGAS models can bring economic advantages to system vendors. Thus, researchers struggled to study the approaches of supporting PGAS models with MPI RMA or two-sided interfaces as the potential underlying communication substrate. Dinan et al. [10] fully implemented the low-level ARMCI runtime system on top of MPI-2 RMA model. Semantically, MPI-2 RMA is suitable as backend for higher-level PGAS models. However, the benchmark evaluation results demonstrated that the sub-optimal MPI-2 RMA implementation performed worse than the native ARMCI implementation and thus failed to design a scalable PGAS communication subsystem. Daily et al. [11] explored three alternative methods to check the possibility of applying MPI-2 on the implementation of the one-sided communications in PGAS models. Although the two-sided communication model naturally requires an implicit synchronization, the MPI two-sided model based design with the asynchronous progress mechanism in mind can even achieve higher overlap of communication and computation than that of using RMA model. This is due to the highly-tuned implementation of two-sided primitives in MPI-2.

Furthermore, Bonachea et al [12] gave the reasons why the MPI-2 RMA interfaces introduced in MPI-2 as well as the traditional two-sided primitives are the inefficient compilation targets for PGAS models. Using two-sided message paradigm for PGAS languages adds considerable overhead, which is especially detrimental to small message performance. Clearly, MPI two-sided model also cannot be an efficient underlying communication backend for PGAS models. Besides, a set of informative suggestions for optimizing the MPI-2 RMA semantics are provided and MPI-3 RMA has already taken those suggestions into consideration. PGAS programming model are dedicated to provide high-performance for remote access. This necessitates the performance comparison of RMA operations for MPI and other PGAS languages or libraries. An early paper [13] compares the performance and scalability of SHMEM and MPI-2 RMA routines on a SGI

Origin 2000 and a Cray T3E-600, in terms of different communication patterns. The results show that the SHMEM implementation is superior to the MPI-2 implementation, especially for the small messages. In addition, another heuristic study [14] is conducted to compare the performance of basic MPI-2, UPC and Cray SHMEM RMA communication routines in latency and bandwidth on Cray XE6. The results obtained by using a distributed hash table application show that one-sided MPI-2 has relatively inferior scaling behavior. Importantly, this study concludes that the performance differences among MPI-2, UPC and Cray SHMEM are attributed to the different exposed memory for remote accesses. Clearly, The memory for RMA operations is somehow customized for Cray SHMEM (symmetric memory) and UPC (shared memory). Comparatively, random (unrestricted) allocated memory can be specified to be remotely accessible in MPI-2. However, MPI-3 supports a specific window, where the data can be remotely accessed with direct load/store instructions. Specifically, a research [15] has implemented the OpenSHMEM based on the MPI-3 and importantly leveraging the MPI-3 shared-memory window. The evaluation results show that the intra-node RMA communication performance can get greatly improved when using the shared-memory window. However, the implementation is not yet adapted to cater for a generalized case where processes are spread across multiple nodes.

1.1.2 Proposed design

In this section, I first give a brief overview of my proposed design which addresses the poor-portability, suboptimal-performance and weak-adaptiveness issues that come with the existing designs. Next, I detailedly state the design challenges to efficiently use MPI RMA model and the proper solutions.

1.1.2.1 Overview

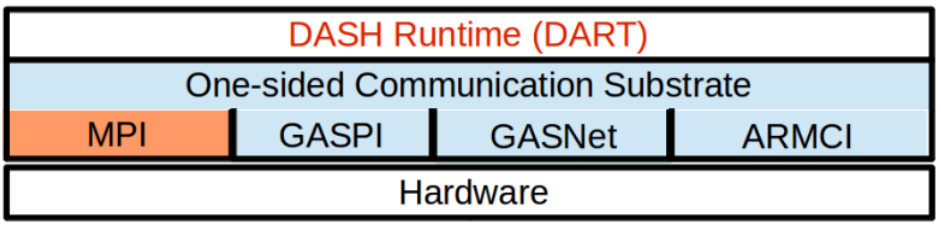


FIGURE 1.2: The layered structure of DART

I developed a MPI-based low-level hierarchical communication library providing the qualified Application Program Interfaces (APIs) which comply with the characteristics mentioned in Sect. 1.4.4. The MPI-based implementation design combines the advantages of portability provided by MPI and that of the ease-to-use interfaces provided by the

communication library. Compared with the traditional MPI-2 RMA model, the relaxed semantics and tuned performance of MPI-3 RMA model makes it become an attractive candidate for the underlying communication conduit. So far this communication library has been adopted and proved as the portable runtime system in a DASH [16, 17] project, which is a PGAS approach and aims to support hierarchical data locality in a form of a C++ template library. The DASH runtime system is called DART (DASH RunTime) and the layered structure of DART is shown in Figure 1.2. Theoretically, DART can actually be implemented based on the existing RMA communication libraries, e.g., MPI, GASPI, GASNet and so on. Hereafter the MPI-based library implementation is denoted as DART-MPI supporting shared memory-style on distributed memory systems. The DART APIs can be classified into the following five components:

- Initialization and shutdown
- *Team* and *group* management (act like MPI communicator and group)
- Synchronization
- Global memory management
- Communications (RMA and collectives)

These five components are not implemented independently but somehow have a close relationship.

Note that, MPI requires the programmer to identify the parallelism and design the parallel applications explicitly. Besides that, MPI does not intuitively expose the data locality to user applications. Thus, the programmer should have profound knowledge on the details in MPI when directly using it as an efficient runtime system for PGAS models. Clearly, the proposed MPI-based communication library can keep the user away from MPI complexities by encapsulating them inside. Therefore, it can also provide a portable, scalable and generic communication scheme that can be directly adopted by programmers/users or easily forms the basis for the runtime system of other PGAS projects.

1.1.2.2 Problem statement

I aim to design the communication components (RMA and collective) in DART on top of the counterparts in MPI. However, the DART system cannot be implemented straightforwardly as a result of the semantic gaps between MPI and DART. I intend to address the gaps and fill them without violating the semantics of MPI and adding notable overheads over the native MPI. Therefore, I set a goal that the DART presents the ease-to-use yet

efficient interfaces in a correct manner to the user while encapsulating the complexity of MPI code. I present the challenges to the design of DART based on MPI communication layer as below in detail:

- Is the MPI-3 RMA model a suitable underlying communication conduit for DART implementation and how to apply it roughly onto DART RMA if it is – In MPI-2 standard, the support for RMA communications is added. MPI-2 RMA only supports separate memory model which provides portability to a great extent, even on the non-cache-coherent architectures. After MPI-2, MPI-3 standard was released and fully backward compatible. It added a substantial extension to the existing RMA model. This extension largely enhanced the usability and effectiveness of the MPI RMA model by depending on hardware coherence. Choosing the relatively suitable MPI version, memory model and RMA synchronization mode is the first and also critical step for achieving high performance DART implementation.

- How can the DART team and group be built on top of the MPI communicator and group? – Like MPI, DART also supports the capability of process grouping, where programmers can create groups consisting of a series of units/processes. Unlike MPI group, the DART group is always arranged by unit IDs in an increasing order. Such difference should be solved. To form a valid scope for RMA and collective communications, DART associates a team with a single group. However, team is presented as an integer. It is important to find an efficient mapping relationship between team and communicator whiling taking the space and time complexity into consideration.

- How can the DART global memory be structured and designed based on the MPI window to facilitate data locality for DART RMA communications? – The global memory region in DART is allocated to be accessed by remote processes and thus conceptually matches the MPI window within certain access epoch. Both MPI-2 and MPI-3 standard define several MPI creation fashions. Hence, choosing the adequate MPI window types and utilizing them in a way that expose data locality could be a critical factor for performance enhancement. The expected remote memory location can be determined precisely with a global pointer. The global pointer should be fully identified to carry the key information on the position of target data location.

- How can the DART communication operations be designed with the data locality in mind? – DART, as the runtime system of a PGAS approach, should provide good performance of data transferring. Foremost, the correctness of DART communications should be guaranteed, even in complex situation where overlapping access

epochs happen. Secondly, there are various RMA communication and synchronization routines available in MPI standard. Especially, MPI-3 adds more attractive functionalities and features. For each DART RMA interface I should identify one memory access method which considers the data locality and retains the performance of the native MPI RMA interface as well. Finally, the way of dereferencing global pointer matters to achieve optimal performance.

- How can the DART non-blocking RMA communications be designed in a way that the network latencies are able to be hidden properly? – Originally, the progression implied by the DART RMA interfaces depends on the invoked MPI RMA routines. This means DART non-blocking RMA communications could end up failing to perform computation and communication in parallel. However, allowing overlap of communication and computation in parallel applications, especially for long messages, is critical to achieving good overall performance. Hence, DART should internally enable the asynchronous progression to hide latency by letting CPU contribute to the computation. However, enabling asynchronous progression on the basis of ensuring correctness and achieving low implementation overhead are challenging.

1.1.2.3 Research solutions

The research scope for this dissertation is depicted in Figure 1.3. I present the general approaches to the challenges mentioned above.

1. Choosing MPI-3 as the underlying RMA communication conduit of DART to enhance the performance and scalability – I have utilized MPI-3 RMA as the low-level communication conduit for DART implementation. The strict semantics in MPI-2 RMA brings some limitations which prevent the MPI-2 RMA from being applied as the underlying communication systems of higher-level programming models (refer to Chapter 3). The MPI-3 RMA extends MPI-2's separate memory model with a unified memory model, which provides relaxed semantics so that enhances the usability and effectiveness of the RMA on machines with automatic cache-coherent architectures. In addition, the new MPI-integrated shared-memory programming model also makes MPI-3 RMA suitable to run parallel applications on cluster of multi-core systems and thus be a qualified low-level communication subsystem for DART implementation. DART utilizes the unified memory model and passive synchronization mode which is closer in semantics to the truly one-sided communication operations than the active mode.

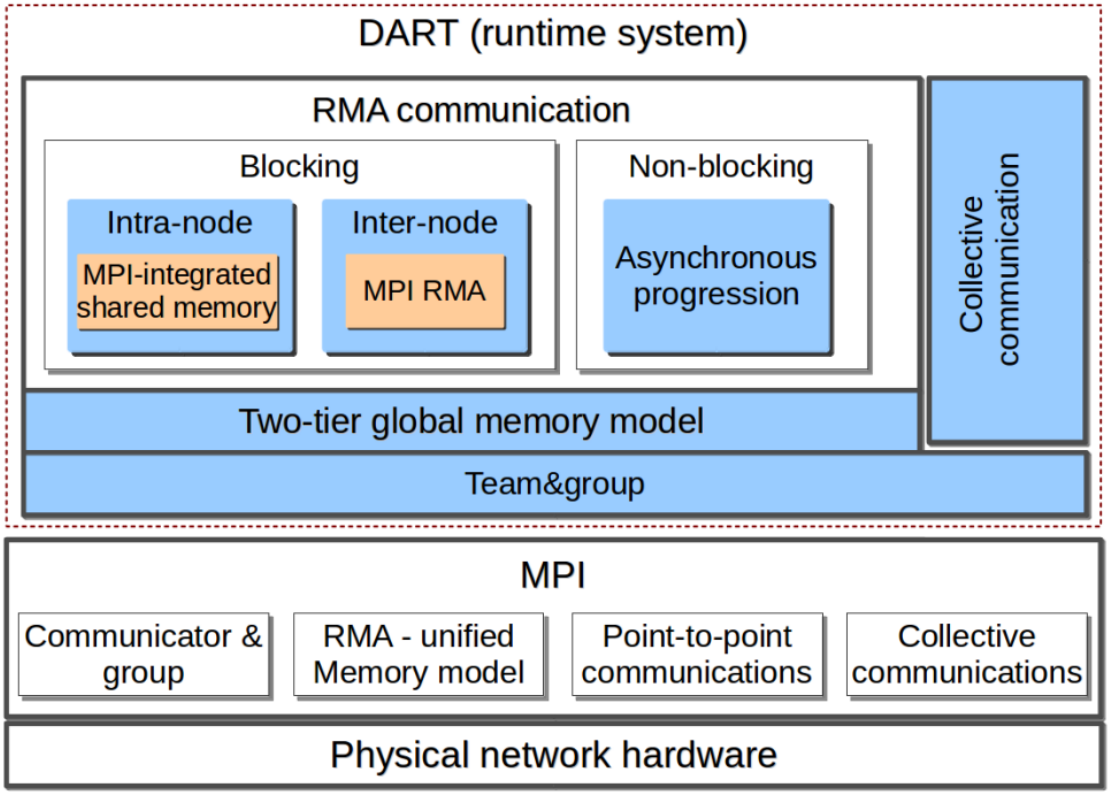


FIGURE 1.3: Research scope for this dissertation.

2. Designing one-to-one mapping relationship between DART team and MPI communicator – I have merge-sorted each new created DART group and attached team with another integer (this integer is not the team ID), with which the corresponding communicator can be determined correctly. Unlike the team ID, this integer gets controlled internally in DART.

3. Designing a hierarchical DART global memory model through the nested MPI windows – I have given a determined meaning to each field in the global pointer. Besides that, I have built up a two-tier global memory architecture by nesting shared-memory *window* (serve intra-node data transfer) inside dynamically-created *window* or MPI-created *window* (serve inter-node data transfer) according the type of global memory.

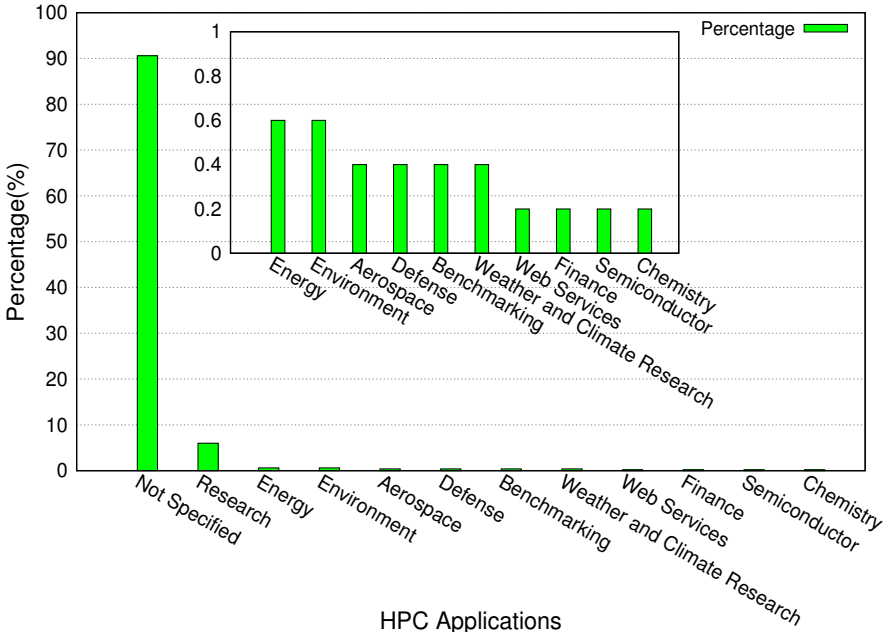
4. Designing an efficient DART communication system with the awareness of the data locality – I have described the locality-aware design for DART RMA communication operations based on the nested window structure. Importantly, I have guaranteed the correctness of DART RMA without violating the semantics of MPI RMA. In addition, I have designed the dereference path of the global pointer according to intra-node or inter-node communications.

5. Designing the asynchronous DART non-blocking RMA communications with the

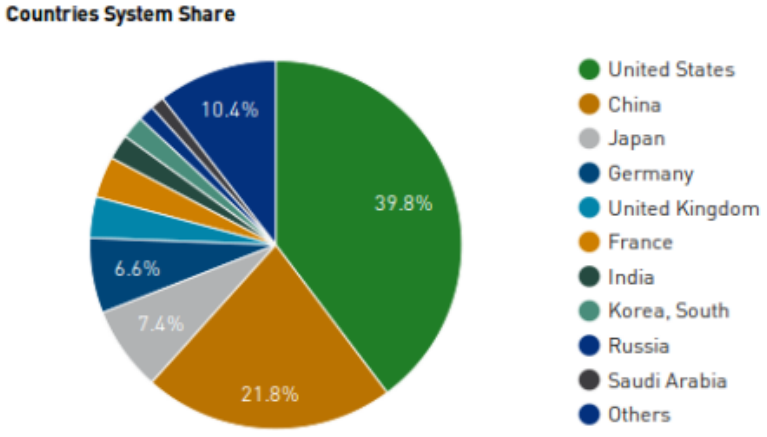
progress process-based approach – I have enabled the asynchronous communications of DART non-blocking RMA for large message sizes by exploiting the progress process-based approach. Such asynchronous progression can provide higher communication and computation overlap and in this thesis is designed to take the data locality into consideration.

1.2 Parallel computing

In contrast to the traditional serial computation, parallel computing is the simultaneous use of the multiple processes to solve a large problem [18]. Specifically, the large problem



(A) Application area system share



(B) Countries system share

FIGURE 1.4: Statistics out of TOP500 database on Nov. 2015 (TOP500 [1]).

should be split up into multiple and smaller ones that are calculated concurrently, which is technically called parallelism. Furthermore, Domain Decomposition (DD, [19]) is a natural way to divide a domain from the original large problem into sub-domains, which can then be conquered in parallel. Currently, the developments in computer hardware are embodied in the ubiquity of the multi-core and multi-processor system and fast network. Parallelism has been used for years to simulate large, complex problems in many areas of science, engineering and business. To make better use of the underlying hardware, it should be continued to be employed in the foreseeable future. The parallel computers tend to be constructed in a form of distributed Cluster of Workstations (COW), Symmetric Multi-Processors (SMP) or commodity PCs. High Performance Computing (HPC, a use of parallelism) could satisfy the demand for executing complex application programs efficiently based on the features of the advanced parallel computers – powerful processing capability and high memory availability. To enable parallelism, we take a software technique – parallel programming – to shorten the running time of a program by managing multiple independent tasks in the program concurrently. Parallel programming can be addressed using parallel programming models. Obviously, HPC technique is expensive, both from the programmer and hardware perspective. Nevertheless, as the Figure 1.4 shows, HPC technique is employed all over the world and in various fields of applications.

1.3 Parallel computer architecture

Nowadays different hardware architectures are available to perform a single task for differing purposes. Typically, Michael J. Flynn [18] proposed a classification for computer architectures, known as Flynn's taxonomy. With this classification scheme, the computer architectures are differentiated according to the possible concurrency of instruction and data streams (Single or Multiple). In general, there are four categories according to Flynn: Single Instruction, Single Data stream (SISD); Single Instruction, Multiple Data streams (SIMD); Multiple Instructions, Single Data stream (MISD) and Multiple Instructions and Multiple Data streams (MIMD). Detailedly, SISD indicates a sequential system where only one instruction stream is operated using one data stream as input. SIMD, MISD and MIMD systems address the types of parallel computer. MISD system is rarely used in the actual parallel computers. The SIMD execution scheme is also covered by many MIMD architectures. Single Program, Multiple Data (SPMD), as a class of MIMD, is a popular technique employed to achieve parallelism for distributed memory parallel supercomputers. In SPMD execution model, a task is divided into several subtasks, which run concurrently on multiple processors with different input data in order to speed up the performance. The majority of modern parallel computers fall within the

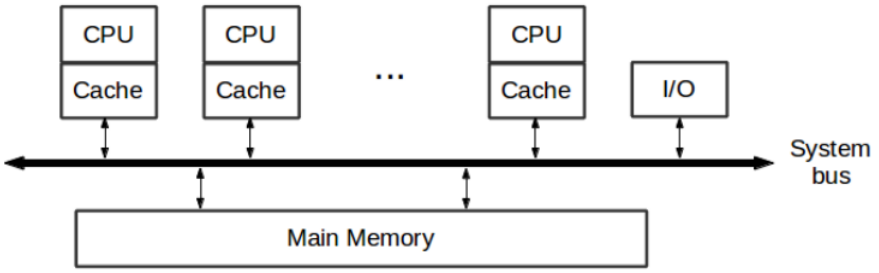


FIGURE 1.5: Diagram of a typical SMP system

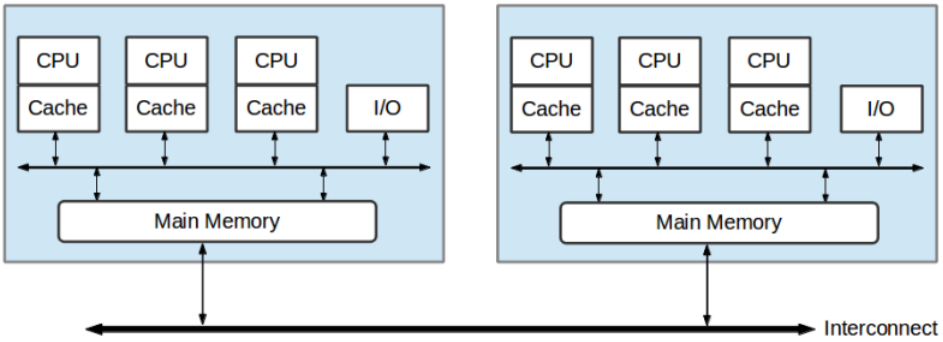


FIGURE 1.6: Diagram of a typical NUMA system.

MIMD category because of its comprehensibility. Consequently, the MIMD architecture is further classified by memory types. The parallel computer memory architectures below are introduced in order of scaling.

1.3.1 Shared memory

In shared memory machines, all Central Processing Units (CPUs) share the same address space. These CPUs (also can say processors) are allowed to communicate with each other through reading/writing data from/to the shared memory without explicit data transfers. This leads to a simple system for the parallel programming scheme. Two common categories of shared memory systems exist: Unified Memory Access (UMA, unified memory access time) and Non-Unified Memory Access (NUMA, non-unified memory access time).

An UMA machine is featured by the centralized shared memory known as Main Memory (MM) with several processors working on it. Each element of MM can be equally accessed by the processors in terms of latency and bandwidth. Therefore, UMA machine is also commonly represented as Symmetric Multiprocessing (SMP) system. To alleviate the system bus traffic problem, each processor is equipped with a private relatively speedy memory called cache memory. Meantime, the demand for cache coherency (i.e., all processors are aware of updates in MM triggered by any processor) is met at the hardware level. In this sense, UMA system is also called Cache Coherent UMA system (CC-UMA).

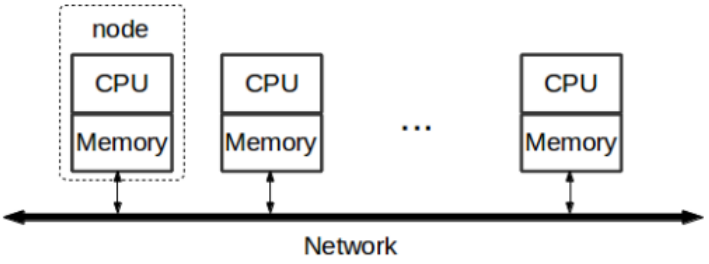


FIGURE 1.7: Diagram of a distributed parallel system.

A typical SMP system is shown in Figure 1.5. However, SMP systems likely hit the performance bottleneck when the number of associated processors is increased due to the bandwidth limitation of system bus. Hence, SMP systems are lack of scalability.

To overcome the bottleneck issue in SMP systems, the NUMA architectures emerge. Basically, NUMA machines are constructed by physically combining two or more SMPs. In NUMA machines, the speeds of accessing certain memory for a processor rely on the physical distance between them. I.e., the access time for a processor to the memory in another SMP (more distant memory) is slower than that to the memory in the same SMP (closer memory). NUMA memory architecture with cache coherency in mind at the hardware level is known as a Cache Coherent NUMA (CC-NUMA) system. Importantly, this feature is widely supported in current parallel hardware. As the Figure 1.6 shows, multiple CPUs are connected with each other with an interconnect port, which is called Quick Path Interconnect (QPI) by Intel, and Hyper Transport by AMD. Further, the modern processor chips are extensively featured by multi-core architecture. Each core can be treated as a single logical processor and all cores communicate with each other through shared memory or shared cache. The hardware to ensure cache coherency is integrated into the CPU. This technique is often named "Chip Multiprocessor (CMP, [20])". Most multi-core processor systems today use an SMP architecture where all cores have equal access time to the same memory space. Actually, several multi-core processors could be connected to describe a NUMA system where each multi-core processor is viewed as a NUMA domain. Noticeably, the NUMA machines are enhanced by the CMP technology.

1.3.2 Distributed memory

In distributed memory systems, each processor has its own private memory address, which is not allowed to map to another processor's. Unlike shared memory architectures, global address memory space across all processors does not exist in distributed memory systems. Therefore, explicit inter-processor communication and synchronization operations are required when a processor needs to access data in another processor (remote data). The programmers are responsible for the above operations. Therefore,

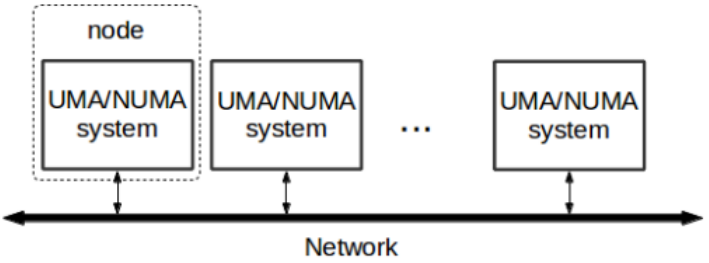


FIGURE 1.8: Diagram of a hybrid distributed-shared parallel system

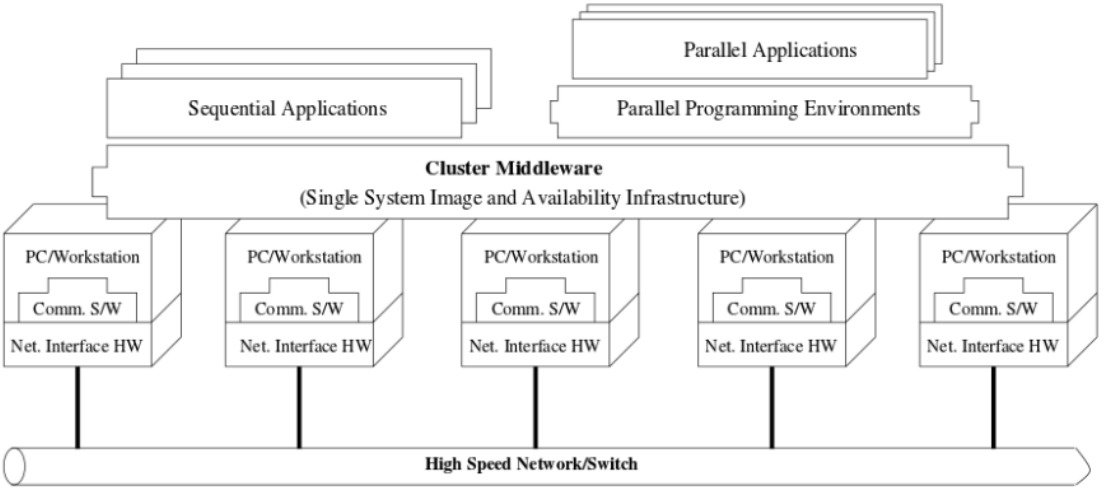


FIGURE 1.9: Cluster computer architecture (R.Buyya[2]).

a communication network is entailed in a distributed memory system to collect those discrete processors, as shown in Figure 1.7. Obviously, distributed memory machines could be highly scalable.

1.3.3 Hybrid distributed-shared memory

The distributed-shared memory system frequently use the hierarchical memory architectures, where the shared memory component is a shared memory machine (e.g., UMA, NUMA system) and the distributed memory component is the networking of multiple shared memory machines. Processors within the shared memory component share the same global memory address space. However, explicit network communications are required to move data from one machine to another. Figure 1.8 describes a typical hybrid distributed-shared parallel system. Such hybrid memory architecture combines the efficiency of shared memory and the scalability of distributed memory. For this reason this hybrid architecture has achieved a triumph in the HPC community and is frequently employed by the modern high-end computers.

A cluster [2], as the state-of-art parallel computer, is a classic example of the hybrid system. A cluster is a collection of loosely-coupled and stand-alone computers (generally

called nodes). A compute node can be a single or multiprocessor machine (PCs, workstation or SMPs) with memory and I/O facilities running on the same operation system. Those computers closely work together as a single, integrated system. In addition, some customized high-performance network hardware, such as the Myrinet, Infiniband and Cray Aries network, are specifically designed for clusters to enable high-speed communication between nodes. Compared to the expensive specialized supercomputers, clusters are more appealing since they are built using affordable, cost-effective and commodity hardware. Therefore, Clusters, as the general-purpose systems, are gradually becoming the standard platforms for high-performance and large-scale computing. This is fitted into a wide spectrum of applications of science, engineering, commerce and industry that demand low-cost and scalable HPC. Note that, the vast majority of the TOP500 [1] high-end computers are clusters. Figure 1.9 shows the typical architecture of a cluster, where the programming environments form the portable, scalable foundation for developing efficient applications.

1.4 Parallel programming models

There are several approaches for developing efficient parallel applications to solve some complex problems on the above parallel memory hardware. Those approaches lead to different parallel programming models. Therefore, selecting the appropriate programming model according to the underlying memory architecture is critical for achieving high performance HPC. The available parallel programming models below are introduced.

1.4.1 Shared memory model

The shared memory model can be applied on the shared memory parallel computers where the global shared memory is accessible to all attached processors. Shared memory programming is also known as threaded programming where threads are regarded as lightweight processes that run in the context of an operating system process. Threads require less overhead than spawning a new process because the system does not initialize a new system virtual memory space and environment for the process. Threads are supposed to carry the information describing the process that contains them. Multiple threads within a process share the same global memory address space and are executed simultaneously to implement parallelism. Data exchanges between threads are achieved through shared memory with load or store instructions. Meantime, the programmers need to take responsibility for the thread-safety when race conditions happen. Obviously, the parallel programs can be written in a simple way with the shared memory

programming model. The threaded programming method is limited to a single multi-processor or multi-core system. Nevertheless, using threads in programs can indeed shorten their execution time.

Parallel programs can be implemented for shared memory systems using the following two models: POSIC threads (Pthreads, [21]) and Open Multi-Processing (OpenMP, [22, 23]). The Pthreads is presented as a library defining a set of C language programming types and procedure calls. A thread is spawned by defining a function. On multi-process or multi-core systems, parallelism is achieved by scheduling the threads to run on different processors. OpenMP exists in form of an API, which is agreed by a handful of well-known parallel machine hardware and software vendors. Therefore, OpenMP allows the programmer developing portable and scalable parallel programs on the shared memory architectures.

1.4.2 Message passing model

The Message Passing Interface (MPI, [8]) model has been widely and commonly used to parallelize applications on clusters of distributed machines. MPI is frequently combined with SPMD execution model as its scalability and simplicity. MPI provides a rich of low-level communication interfaces including one-sided, two-sided and collective operations. MPI describes a sequential C, C++, or Fortran program which runs concurrently on multiple processors in the cluster. The attractiveness of MPI is that an MPI application can directly run on any type of cluster without any change. This is due to that MPI is standardized based on the consensus of the MPI forum organized by vendors, library developers, researchers and users. Thus, MPI is readily supported in any cluster, which leads to high portability. It is observed that MPI is more like an assembly language programming for parallel applications. However, MPI, as an explicit programming model, requires the programmers to identify the parallelism and design the parallel applications explicitly, including the distribution of sub-tasks and the communication and synchronization between them. As a result, the programmers would better have profound knowledge on the details in MPI so that write efficient and correct applications parallelized using MPI.

MPI has become the de-facto communication standard for parallel programming on distributed parallel machines. MPI is increasingly getting attention because of its capability of delivering acceptable and portable performance for diverse underlying network hardware. Besides that, MPI is updated and evolved gradually in a way of adding more functionalities and features to satisfy both the programmer's expectation on performance and the development in the underlying memory hardware. Detailedly, in 1995, the second version of MPI standard [24] added support for RMA operation on top of

the two-sided (point-to-point) and collective communication models. MPI programs that adhere to MPI-2 RMA may manifest acceptable performance on clusters of single processor workstations or PCs rather than SMPs or multi-core processor workstations. This is due to that explicit data transfers between processors sharing a global memory address exacerbate the performance. To address some of the limitations in MPI-2 RMA, MPI-3 standard was released in 2012 to propose a significant extension to MPI RMA with improved semantics and features. With the extension, MPI is qualified for supporting high-performance as well as portable RMA communications to large-scale applications. In this sense, MPI RMA model is suitable to be applied on any parallel machine regardless of the underlying physical architecture – shared memory, distributed memory or hybrid distributed-shared memory.

There are several well-tested and efficient implementations of MPI, which will differ in which platforms of the MPI programs they run on. MPICH [25] is a high-performance, freely-available and portable implementation of the MPI and MPICH3 supports the latest version of the MPI standard – MPI-3. A vast majority of other important MPI implementations, including IBM MPI, Intel MPI, Cray MPI, OSU MVAPICH/MVAPICH2 and so forth, are derived from MPICH.

1.4.3 Hybrid programming model

With the advent of multi-core architecture, Clusters of multi-core, are of value in the HPC community, leads to two-layer memory hierarchy– intra-node and inter-node. Clearly, multi-core clusters offer two parallelization layers – shared and distributed. The ever-increasing number of computing cores in a CMP, even many-core in the further, enhances the importance of writing parallel programs with optimal performance and scalability on multi-core clusters. Technically, a shared memory programming model providing fast communication through shared memory can be exploited within one node and the communications across nodes are achieved through a message passing programming model. Based on this approach, applications are parallelized by taking advantage of two programming models, which is known as hybrid programming model. Hybrid programming model could be a reasonable way to cater for the hierarchical hardware environment of multi-/many-core clusters.

A typical example of hybrid programming model is the combination of MPI with threads model (such as Pthreads and OpenMP). Concretely, threads model serves the on-node data communications and parallel execution over the nodes/network is achieved through MPI. There are applications [26–28] built using the hybrid model of mixing MPI with other shared memory models (Pthreads or OpenMP) for achieving the peak performance.

This model can provide flexibility and performance improvement potential for the parallel programmers. However, the programmers are burdened with the complexities of employing and coordinating the two mixed programming models in single application. The complexities include the challenge to write an efficient MPI code, the guarantee of a correct multi-threaded application (thread-safety) and the avoidance of letting the threads codes corrupt the MPI semantics [29]. Therefore, with the hybrid model, the applications could be error-prone. This will implicitly add more workloads, e.g., frequent debug and error tracing, to the programmers.

1.4.4 Partitioned Global Address Space (PGAS) programming model

To virtually enable the shared memory-style programming on the distributed-shared machines (e.g., multi-core clusters), an approach, Partitioned Global Address Space (PGAS) programming model was studied to provide the illusion of a global shared memory address space for the parallel programs on any parallel machine regardless of the underlying hardware. PGAS model treats a distributed system as if the memory were shared on a global address space that is logically partitioned. Each portion of the global address space has an affinity for a certain process and thereby the locality of reference is exploited by PGAS. Compared to the flat address space offered in the traditional shared memory approach, the data locality in PGAS model can be identified in each partition of the global address space. Therefore, the PGAS programming approach can greatly simplify the tasks of developing parallel applications (program very much like on shared memory systems) while exposing data locality to enhance performance. PGAS intends to combine the advantages of a SPMD programming style for distributed memory systems (as employed by MPI) with the shared memory programming style for shared memory systems.

A number of PGAS programming languages have been implemented in the past, including Unified Parallel C (UPC) [30], Co-Array Fortran (CAF) [31, 32], Chapel [33], X10 [34], Standard Template Adaptive Parallel Library (STAPL) [35] and Titanium [36]. X10 is designed as an object-oriented parallel programming language, through which the Asynchronous PGAS (APGAS, [37]) is realized. X10 needs to leverage a runtime system that named X10RT for doing the underlying communications. X10RT [38] is presented as a C library and can be implemented in different forms, in which the X10RT-MPI is realized on top of MPI-2. The STAPL is a productive framework for C++, it provides support for developing parallel program on both shared and distributed memory system. Typically, these approaches rely on one-sided communication conduits, e.g., Global Address Space Networking (GASNet), to perform RMA operations. GASNet [6] is a low-level networking library which implements remote memory access primitives and

is thus particularly fit into for PGAS models. Besides the PGAS languages, there are also approaches that implement PGAS in a form of a library. An example is SHMEM, a library API that allows its participating processes to view a partitioned global address space. Currently, the OpenSHMEM community project [39] is building a new and open specification to consolidate the various existing SHMEM versions into a widely accepted standard. Global Array (GA) [40] has originally been developed over 20 years and provides one-sided global data access for regularly structured one- or multi-dimensional arrays. The reference implementation of OpenSHMEM is also based on GASNet. GA exploits ARMCI [7] as its primary underlying communication layer. Basically, the following characteristics [12] are entailed in the interfaces of the PGAS languages or libraries:

- simple and ease-to-use
- allow performing RMA communications.
- yield acceptable performance for memory accesses in latency and bandwidth. This is targeted for the blocking or non-blocking RMA operations.
- be able to hide network latencies by overlapping communication with computation. This should be achieved by calling the non-blocking RMA operations.
- allow performing collective communications (e.g., broadcast, gather, reduce and so on) and process synchronization (such as barrier) operations.

Regarding the terms of blocking and non-blocking, we can refer to the next chapter.

1.5 Overview of dissertation structure

The remainder of this dissertation is organized as follows. Chapter 2 provides overview on MPI, PGAS, DASH and DART. Chapter 3 provides a justification for applying MPI-3 RMA instead of MPI-2 RMA onto the DART RMA communication system. The design method of the DART team and sorted group on top of the MPI communicator and group is described in Chapter 4. The approach of constructing the DART two-tier global memory model based on MPI nested windows is elaborated in Chapter 5. Chapter 6 elucidates the approach of designing the DART RMA communications with data locality in mind using the above two-tier global memory model. Chapter 7 provides the method of enabling the asynchronous progression for the DART non-blocking RMA communications. The results with regard to the application benchmarks are illustrated in Chapter 8. Finally, Chapter 9 draws conclusions of this work and outlines future research directions.

Chapter 2

Background

2.1 Overview of MPI basics

MPI [8] is the de facto standard for message passing that is originally proposed in 1993 by MPI forum. MPI provides a rich set of portable messaging interfaces that could be implemented efficiently on any machine. For this reason, MPI is adopted as the underlying communication substrate by DART system. The interfaces defined in MPI standard imply a number of functionalities and features which come with communication (point-to-point, one-sided and collective) semantics, communicator and group processing and language bindings for FORTRAN, C and C++ [8]. In this section, I will introduce the basic concepts related to the latest MPI standard – MPI-3. Understanding these basic concepts can help the user to better take advantage of the various MPI functionalities and features when writing efficient applications.

2.1.1 Communicators, groups and ranks

Understanding communicators and their roles in parallel applications is critical since communicators form the foundation for all MPI point-to-point and collective communications. The concept of *communicator* is brought up to provide an adequate scope for communication operations that happen in the parallel programs. Communicators come in two varieties with different purposes: intra-communicators and inter-communicators. Intra-communicators primarily consist of a *group* of MPI processes and a unique communication *context*. Intra-communicators are mainly of concern to message passing in MPI and only processes belonging to the same communicator are allowed to communicate. More specifically, an MPI group is a local representation of a set of MPI processes. Communicators allow a safe, interference-free way of differentiating disjoint

communication space. That is, that only communications sharing the same context are visible to each other. Importantly, the unique context provides a guarantee: the data transfers (message sends or receives) happening in different communicators will not be confused. Unlike intra-communicators, inter-communicators deal with operations between two disjoint groups of processes and however less commonly used. The number of processes in a group/communicator, indicates the size of group/communicator, is available via *MPI_ Comm_ size* or *MPI_ Group_ size*. An integer *rank* uniquely identifies each process in a specific group/communicator. The *rank* can be obtained by invoking *MPI_ Comm_ rank* or *MPI_ Group_ rank*. *Rank*s covered within a group/communicator are alway consecutive and starting from 0. Accordingly, each process will have its relative *rank* in a specific group/communicator. The *rank* of a process in the *MPI_ COMM_ WORLD* is regarded as the absolute *rank*.

In the lifetime of a parallel application, the MPI library is initialized via *MPI_ Init* and finalized via *MPI_ Finalize*. Note that, no MPI function can be invoked before initialization or after finalization. At the time of program launch, MPI provides a universal communicator – *MPI_ COMM_ WORLD* that consists of all the processes available for communications. We can actually create new sub-communicators from *MPI_ COMM_ WORLD* via various collective operations (such as, *MPI_ Comm_ create*, *MPI_ Comm_ split* and *MPI_ Comm_ split_ type*). The process group associated with a communicator can be extracted locally by the MPI function (*MPI_ Comm_ group*). A large number of operations on groups include creating new groups by unions, intersections and differences, extracting new groups from other parent groups and translating ranks between groups (due to the existence of the relative and absolute rank mentioned before).

2.1.2 Point-to-point communications

The point-to-point communication model, as the basic communication model in MPI, describes the explicit communication operations between two MPI processes (communication sides). The primary communication primitives in this model are *send* and *receive* operations. The communication side that invokes sending routine is called *sender* and the *receiver* should call a receive routine. Besides the data section, the transferred message carry the information (message envelop) that can be used to differentiate messages. The triples [*senddatabuf*, *senddatasize*, *senddatatype*] and [*recvdatabuf*, *recvdatasize*, *recvdatatype*] determine the message data that is sent and received, respectively. Message envelops provided at the *sender* and *receiver* side are presented as triples [destination,

tag, communicator] and [source, tag, communicator], respectively. In this case, the *destination* specifies the receiving process's rank and the rank of the corresponding sending process is specified by the value of *source*.

A message can be received on a receive operation when its envelop matches the triple [source, tag, communicator] provided by the receive operation. On top of the communicator context, the tag must be identified to provide a finer-grained control over message matching. The types of the basic elements at both communication sides should be checked after the message matching. That is, the *sendatatype* value provided at the sender side should be the same as the *recvdatatype* specified by the receiver. Note, however, that it is not required that the size of the buffer allocated by the sender (i.e., *senddatasize*) and that of the buffer allocated by the receiver (i.e., *recvdatasize*) is identical. In detail, the expected received data size could be greater than the practical sent data size. Importantly, in point-to-point communication operations, messages are non-overtaking, which means messages are matched in the order they are issued by the sender when multiple communications are issued with identical message envelop. Furthermore, a wildcard – *MPI_ANY_SOURCE* – can be used by the receiver to address the wide receptions from all the processes in the matching communicator. Likewise, with a wildcard *MPI_ANY_TAG*, the receiver is allowed to receive a message with an arbitrary tag. The receive operation returns a status object from which the critical information of the received message can be obtained. This information includes tag, source and *senddatasize*. MPI provides a routine – *MPI_Probe* to facilitate the probe for arrival of matched messages without actually handling them immediately. The *MPI_Probe*, however, also addresses a status output which provides necessary message information for the following actual receive operation.

There are two modes defined for MPI point-to-point operations: blocking and non-blocking. In blocking point-to-point operations, the calling processes do not return until the communication buffer can be accessed safely (i.e., the sender buffer can be modified or the receive buffer is ready to be accessed). The non-blocking point-to-point operations necessitate the decomposition of the routines for starting the communication (e.g., *MPI_Isend* and *MPI_Irecv*) and the routines for polling or completing the outstanding communication. The reason for doing so is to maximize the computation-communication overlap by freeing the CPU to perform useful computation while the communication is executed. However, whether or not achieve acceptable overlap when using non-blocking point-to-point operations strongly depends on the specific MPI implementation according to the progress rule (refer to Chapter 8). To be specific, the non-blocking point-to-point (send and receive) operations return immediately with request objects by which the status of the outstanding communications can be checked and those communications will eventually be completed. For this reason, MPI defines a number of test (e.g., *MPI_Test*

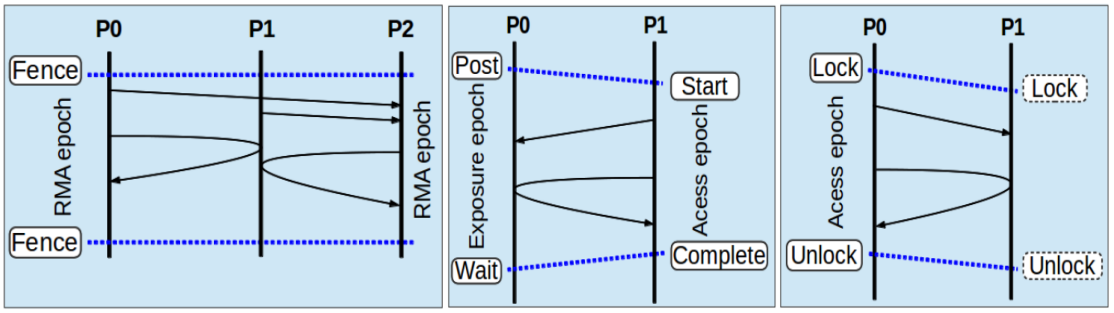
and its variants) and completion (e.g., *MPI_ Wait* and its variants) operations [8]. Likewise, the probe operation can also be non-blocking. *MPI_ Iprobe* returns immediately with a signal value without waiting for the matching message and then the receiver judges if there is a matching message or not in terms of the returned signal value.

The basic MPI predefined datatypes are specified for messages featuring a homogeneous and contiguous structure and whose data values correspond to the primitive datatypes of the host language. Take C for example, possible datatypes predefined in MPI could be *MPI_ INT*, *MPI_ DOUBLE*, *MPI_ CHAR* and so on. For messages that are made of values with different datatypes or items are not stored contiguously in memory, the derived datatype provided by MPI is however a good mechanism constructing the noncontiguous data elements or compound items (contain different datatypes) all in a single data block. With the derived datatypes, one can address the general and complex (mixed or noncontiguous) communication types and transfer directly objects of various shape and size without additional memory-to-memory data copy operations at both communication sides. In practical, derived datatypes are constructed on top of basic datatypes through the use of datatype *constructors*. MPI provides several functions of datatype constructors – *MPI_ Type_ contiguous*, *MPI_ Type_ indexed*, *MPI_ Type_ vector* and *MPI_ Type_ struct*, where the *MPI_ Type_ struct* is the most general datatype constructor.

The communication implementation usually supports two message protocols: eager and rendezvous. By following the eager protocol, the small messages are transfered. I.e, the send operation can send data without waiting for a matching receive as the auxiliary buffer is available on either the sender or the receiver side. On contrary, the rendezvous protocol is used for long message transfers. In this case, data communication is only started when the receiver posts the matching receive operation for the previously issued send operation.

2.1.3 One-sided communications

The one-sided communications, also called Remote Memory Access (RMA), are added in MPI-2 on top of the well-defined point-to-point communications and get enhanced in MPI-3. The communication between a pair of processes in one-sided communication model is exclusively initiated by the *origin* process, which provides all parameters necessary for communication with the *target* process. Basically, the communication (data movement) and synchronization are explicitly decoupled. The one-sided communications simplify programming and can avoid the message matching which is however entailed in point-to-point communication model. The one-sided communications can be significantly



(A) Fence (B) PSCW (C) Lock/Unlock

FIGURE 2.1: MPI RMA synchronization modes.

faster than send/receive on systems with hardware support for Remote Direct Memory Access (RDMA).

MPI *window* is a critical concept to enable MPI RMA communication operations in a way that declares a local memory region as accessible to remote processes. I.e., only the region of memory encompassed by window is eligible to be exposed to remote processes in its associated communicator. All MPI window creation operations proceed in a collective way. Each process in the given communicator generates a window in the allocated or given memory region and returns a window object. The typical MPI window creation operation is called *MPI_ Win_ create*. Here, each process specifies a given local memory region to be exposed. Besides *MPI_ Win_ create*, MPI-3 provides three other MPI window creation variants, namely *MPI_ Win_ allocate*, *MPI_ Win_ allocate_ shared* and *MPI_ Win_ create_ dynamic* respectively, to generate more specific or flexible MPI windows. *MPI_ Win_ free* can be invoked to deallocate the window object.

MPI offers the ability of reading, writing and atomically modifying data in the exposed memory regions. There are three basic MPI RMA communication calls – *MPI_ Put* (move data from origin to target), *MPI_ Get* (move data from target to origin) and *MPI_ Accumulate*, which works similarly to *MPI_ Put* but atomically updates the remote data. MPI-3 also provides counterparts that return request handles, i.e., *MPI_ Rput*, *MPI_ Rget* and *MPI_ Raccumulate*. Those request-versions of MPI RMA communication calls can be synchronized using *MPI_ Wait* to guarantee the local completion of RMA communications. MPI-3 offers several new Atomic Memory Operations (AMOs), including compare-and-swap, fetch-and-op, and get-accumulate. These operations greatly facilitate the creation of a mutex library that uses the Mellor-Crummey and Scott (MCS) algorithm in distributed memory [41]. All the MPI RMA communication calls are non-blocking. Hence, additional synchronization calls are needed to ensure that communications are completed at the local or remote side. The synchronization calls supported in

TABLE 2.1: Basic MPI collective communication routines.

Patterns	**Routines**	**Functionality**
One-to-all	*MPI_ Bcast*	Data movement
	MPI_ Scatter	Data movement
All-to-one	*MPI_ Gather*	Data movement
	MPI_ Reduce	Collective computation
All-to-all	*MPI_ Allgather*	Data movement
	MPI_ Allreduce	Collective computation
	MPI_ Alltoall	Data movement

MPI are classified into two modes – active and passive target, according to the mechanism for establishing epochs. In the active target synchronization mode, both the origin and target processes are involved in the synchronization and must call MPI routines. Again, the active target mode comes in two flavors: Fence and Post-Start-Complete-Wait (PSVW). Fence is a collective synchronization approach which starts and ends RMA epochs (including access and exposure epochs) on all processes in the given window with the call to *MPI_ Win_ fence*. Note that, the origin can issue a series of RMA communication calls within an access epoch and the target allows other processes to access/update its window within an exposure epoch. In PSCW, the origin and target process should shake hand before issuing communication calls. At the target side, the exposure epoch is opened with *MPI_ Win_ post* and closed with *MPI_ Win_ wait*. On the origin side, the access epoch is opened with *MPI_ Win_ start* and closed with *MPI_ Win_ complete*. Compared with PSCW, Fence synchronization provides a relatively simple way of establishing RMA epochs. On contrary, the passive mode does not require the target to explicitly participate in synchronization operations. The MPI passive mode occurs within an access epoch which should be initiated by locking the RMA window and terminated by unlocking it again. Figure 2.1 graphically illustrates the three basic approaches to establish RMA epochs. Noticeably, calls to MPI RMA communication routines must always happen within the access epochs. Furthermore, MPI-3 provide global lock calls – *MPI_ Win_ lock_ all* and *MPI_ Win_ unlock_ all* as an extension to the traditional lock method. Meanwhile, the *flush* functions e.g., *MPI_ Win_ flush* and *MPI_ Win_ flush_ local* are provided to support light-weight synchronizations.

2.1.4 Collective communications

Collective communication is a critical and also frequently-used component in MPI. To execute a collective communication, all processes within the scope of a communicator should be involved to call the communication routine with matching parameters. Unexpected behavior can occur if one task in the communication does not participate or pass

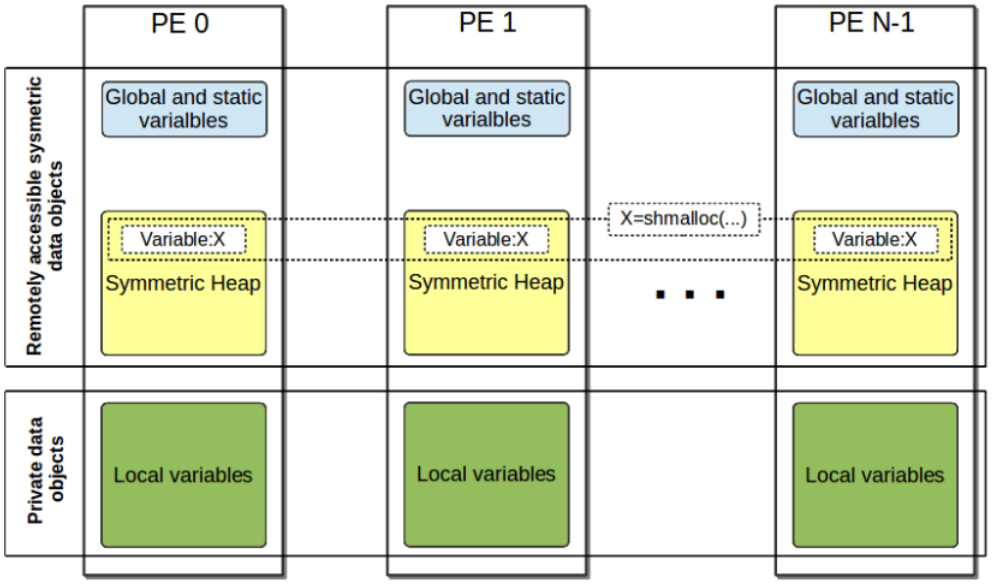


FIGURE 2.2: OpenSHMEM memory model [3].

a mismatched parameter. The MPI standard specifies three collective communication patterns, such as *One-to-All* (dissemination: *MPI_ Bcast* and *MPI_ Scatter*), *All-to-One* (*MPI_ Gather* and *MPI_ Reduce*) and *All-to-All* (*MPI_ Allgather*, *MPI_ Allscatter* and *MPI_ Alltoall*), in terms of the number of the involved sender and receiver. On the other hand, collective communications are classified into three types – synchronization, data movement and collective computation, according to their functionalities. *MPI_ Barrier* provides the global process synchronization where the processes wait until all members of the group have reached the synchronization point. In parallel applications, the barrier routine should be invoked properly to ensure the logical correctness of applications. Collective communication operations can be frequently used in scientific applications to synchronize or exchange data. Figure 2.1 lists several common collective communication routines with additional descriptions.

2.2 Overview of PGAS

PGAS models provide a shared memory-like abstraction that makes it easier to program parallel applications. They also ensure performance by exposing key locality information to the application developer. This has made them an attractive alternative to traditional message passing models like MPI. OpenSHMEM and UPC have been two popular PGAS modes.

```
#include <shmem.h>
#include <stdio.h>
/* Global variables */
char target[5];
int main (int argc, char *argv[])
{
  char source[5] = {0, 1, 2, 3, 4};
  int myid, numprocs;
  start_pes (0);
  myid = shmem_my_pe ();
  numprocs = shmem_n_pes ();
  if (myid == 0){
    shmem_putmem(target, source, 5, 1);
  }
  shmem_barrier_all ();// barrier synchronization
  return 0;
}
```

FIGURE 2.3: A simple example for OpenSHMEM code. This example shows the OpenSHMEM code for moving data from process 0 to process 1.

2.2.1 OpenSHMEM

A joint community, including science and industry institutions, unifies all SHMEM library development effort to create a standardized, open source SHMEM library API for C and Fortran. OpenSHMEM library supports SPMD-like programming so that all processes involved in an OpenSHMEM program can execute different tasks. The key feature of OpenSHMEM API is to offer a complete set of library calls implementing one-sided (*put*, *get*) and collective operations (*broadcast*, *collection*, *reduction*, *barrier*) and AMOs (*swap*, *compare and swap*, *fetch and operation*) and synchronization operations (*barrier*, *fence*, *quiet*, *wait*), which satisfy the needs for communication. Basically, the synchronization operations come along with data transfer operations (put, get). There are a set of put/get communication interfaces for different data types, e.g., double, int, float, short, byte and so on.

The data objects used in an OpenSHMEM program can be either private to each process or remotely accessible by all other processes. Each process reserves some of its local memory to store the private data objects and thus those objects are only allowed to be accessed locally. The remotely accessible data objects are also called symmetric data objects. Those symmetric variables can be either stored in the global/static memory section of each process or allocated from the symmetric heap (by calling the dynamic memory allocation routine – *shmalloc*) at runtime. Note that, OpenSHMEM one-sided operations are all performed on the symmetric objects. Figure 2.2 illustrates the OpenSHMEM memory model consisting of private and remotely accessible data objects (symmetric objects) in an OpenSHMEM program.

OpenSHMEM programs should start with the initialization function *start_pes* before performing communication/ synchronization amongst the involved processes via the OpenSHMEM library interfaces. When returning from the main function, OpenSHMEM programs finish execution and all the resources associated with the library are released. Like MPI, all processes participating in an OpenSHMEM are identified by unique integers. Each processes can get its own integer via the interface of *shmem_my_pe*. The total number of processes (denoted as n) can be obtained by calling *shmem_n_pes*. Processes are numbered from 0 to n-1. Therefore, a simple OpenSHMEM program is listed in Figure 2.3.

There are many closed source implementations of OpenSHMEM for modern RDMA network hardware such as InfiniBand and Cray's Gemini and Aries. Particularly, the Cray OpenSHMEM implementation provides non-blocking version for the basic put/get calls (written like *X_nb*). The OpenSHMEM synchronization routines should come with the non-blocking RMA calls to ensure the re-usability of memory at the local and remote side. Naturally, the OpenSHMEM assumes the processes progress asynchronously to overlap the computation and communication when performing the non-blocking RMA operations. However, it depends on the implementors to decide if the computation and communication are overlapped or not.

2.2.2 UPC

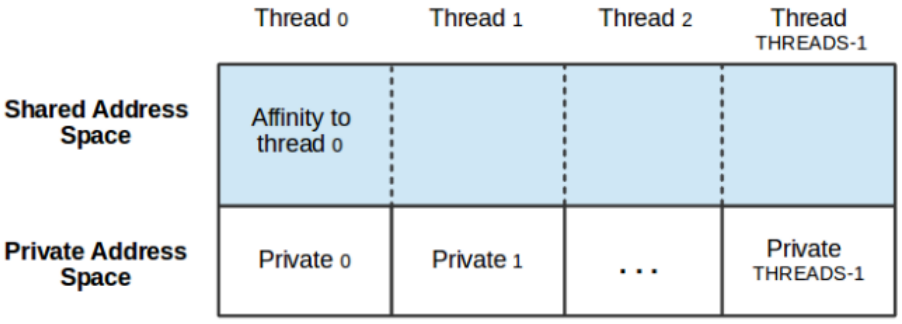


FIGURE 2.4: UPC memory model.

UPC [30] is an explicit parallel extension of the C standard and follows the PGAS programming model. I.e., UPC adds parallelism based on the mature concepts and features of C programming language. The development of UPC is led by a consortium of academia, research and industry. There are many open-source implementations available and tuned for different underlying architectures.

Under UPC programming model, a collection of threads work independently in a SPMD fashion. The total number of threads is available as a global integer constant *THREADS* and can be specified at compile-time or run-time. *MYTHREAD* identifies each thread

```
#include <upc.h>
#include <stdio.h>
int main (int argc, char *argv[])
{
  char source[5] = {0, 1, 2, 3, 4};
  /* private pointer to shared data */
  shared[5]char* target;

  target = (shared[5]char*)upc_all_alloc(THREADS, 5);

  if (MYTHREAD == 0){
    upc_memput(&target[1*5], source, 5);
  }
  upc_barrier;// barrier synchronization
  upc_all_free (target);
  return 0;
}
```

FIGURE 2.5: A simple example for UPC code. This example shows the UPC code for moving data from thread 0 to thread 1.

uniquely as a private integer constant and specifies thread index starting from 0 to $THREADS - 1$. Two memory spaces, the private and the shared, are provided by UPC for the user on top of the distributed shared memory model, which is shown clearly in Figure 2.4. The shared memory space is logically partitioned amongst all participating threads. The portion of the globally shared memory space allotted to a thread is said to have *affinity* to that thread. The elements resided in the shared memory space are allowed to be accessed by all threads. In addition to a portion of the shared memory space, each thread has also its own private space. The private memory is declared and created locally and accessed in a similar way as one does in C. The static and dynamic allocation approaches are supported for shared memory as well as private memory. The shared data objects are statically allocated using a *shared* qualifier. E.g., a single shared object can be created statically with *shared int a*. The shared array in UPC is an array with elements that are distributed on a per-thread basis in a round robin style. Generally, shared array can be statically declared with a sentence like *shared[4] int a[16]*, where 4 is the block size, and 16 is the total size of this distributed-shared array. Likewise, a pointer-to-local (act as the plain C pointer) at a thread only reference addresses in its private space or addresses in the partition of the shared space that has affinity to it. However, a thread may access all locations in the shared address space with a pointer-to-shared, which is a pointer to the data residing in the shared memory space. E.g., *shared[4] int*a* defines a private pointer-to-shared, where 4 is denoted as the block size. The shared memory, that is dynamically allocated using a library function (e.g., *upc_all_alloc*), is identified by a returned pointer-to-shared. With this pointer, the allocated shared memory can be freed through a call to *upc_alloc_free*. Additionally, UPC provides a handful of string functions (e.g., *upc_memput* and *upc_memget*) for data movement.

UPC provides a blocking barrier synchronization routine *upc_barrier*. Figure 2.5 shows a simple UPC example code.

UPC provides a number of utility libraries that provide functionalities or features commonly required for writing parallel applications in modern hardware. Particularly, the non-blocking UPC RMA operations for each of the basic put/get operations (written like *X_nb*) are included in *upc_nb.h*. In library *upc_collective.h*, a set of collective operations, e.g., broadcast, scatter, gather, exchange and reduction, are provided. After including *intrinsics.h*, global AMOs (e.g., *_amo_add_upc* and *_amo_afadd_upc*) are supported to atomically access or update UPC shared variables.

2.3 DART

DART is a plain C-based interface on which the C++ template library DASH is built. DART defines common concepts and terminology and abstract away the specifics of the underlying communication substrate. DART provides a number of interfaces for serving the upper DASH level. DART can be used directly by users or be employed as a general runtime system for other PGAS implementations. The complete DART specification is available online at https://dl.dropboxusercontent.com/u/408013/dart_spec_v2.1.html.

The main task for DART is to establish a partitioned global address space and to provide functions to handle global memory efficiently, such as memory allocation and global pointer dereference and data movement. The DART memory model is shown in Figure 5.3. In addition, DART also provides functions for initialization, synchronization and management of teams. DART provides the functions *dart_init* and *dart_exit* for initializing and terminating a parallel program. DART also features SPMD parallelism. For the simple DART code we can refer to Figure 6.13. The introduction to the functionalities and features provided in DART is elucidated in the chapters below.

Chapter 3

The Suitability and Appliance of MPI-3.0 to Implementing DART

MPI has become the de-facto communication standard for parallel programming, and it is believed to be a popular parallel programming model as its ability of delivering acceptable and scalable performance for those parallel programmers who wish to experiment on various underlying network hardware.

In 1995, the second version of MPI standard added support for RMA operations on top of the two-sided and collective communication models. However, the MPI-2 RMA is not so acceptable as the two-sided communication model is. This is due to the strictly compliance of MPI-2 RMA with the feature of portability specified in MPI standard. In addition, MPI-2 RMA only supports the conservative memory model which lessens the extension space for the usability of hardware abilities to a great extent, such as the automatic cache coherence on a coherent memory subsystem.

PGAS parallel programming model are gaining an increasing interest among the HPC research fields. Considering the prevalence of MPI in writing the scientific applications and its wide portability, researchers in HPC domain have been leading the effort to explore the design opportunities of implementing PGAS models based on MPI. To relax the limitation existed in MPI-2 RMA semantics, MPI-3 standard was released in 2012. It added a substantial extension to the existing RMA model. This extension largely enhanced the usability and effectiveness of the MPI RMA mode being utilized as a runtime system for a variety of PGAS models [42]. Therefore, I leverage the MPI-3 RMA as the underlying communication conduit of DART.

3.1 Applying MPI-3 onto DART

Two kinds of window copies – public and private – are utilized to address the concept of MPI RMA memory model [8]. According to whether the two copies are visible to each other or not, two memory models are identified, i,e., RMA unified and RMA separate.

The public and private window copies are required to be always synchronized explicitly for separate memory model, which is illustrated in Figure 3.1. This figure displays the

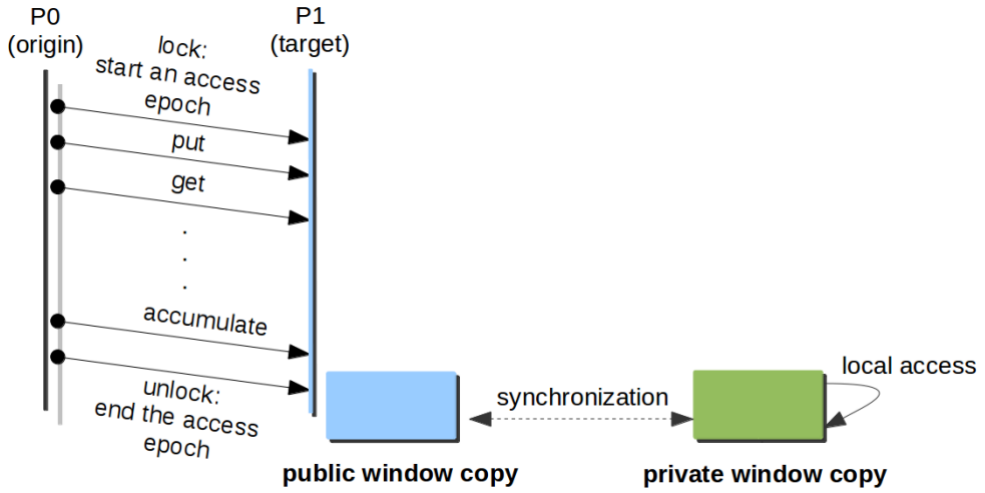


FIGURE 3.1: Schematic description of the public (remote) and private (local) window operations in the separate memory model.

operations on the two window copies and the associativity between them. The local touch to the window is operating on the private window copy; the public window copy is exposed to the remote processes. Noticeably, the requirement for explicit synchronization makes the separate memory model portable to the hardware architecture without coherent memory system.

However, the majority of nowadays hardware features cache consistency. Therefore, in the presence of data consistency, the public and private copies can be maintained consistent implicitly. As a result, the unified memory model emerges to obviate the explicit synchronization between the two copies on systems, where provide data consistency. I.e., the public and private copies are viewed as the logically identical memory (visible to each other) in unified memory model.

According to the Sec. 2.1.3, we learn that MPI RMA supports two kinds of synchronization modes – active and passive target. The passive mode does not involve the target process explicitly in the course of synchronization. Compared to the active target, the passive target mode is more suitable in semantics to support the truly one-sided communication model that provided in DART. Moreover, the passive target mode facilitates

the implementation of applications with dynamic communication patterns. Therefore, DART utilizes the passive synchronization mode.

The MPI passive mode occurs within an access epoch which should be initialized by a MPI lock call and terminated by a corresponding MPI unlock call. Furthermore, passive target mode provides two kinds of lock modes – shared and exclusive. Exclusive lock prevents all concurrent accesses from different remote processes even in case of accessing non-overlapping locations of the target window. Obviously, this behavior completely serializes all accesses to the memory within a window. On contrary, shared lock allows the concurrency of different data accesses. The MPI-2 RMA only supports the RMA separate model which provides conservative semantics in the window access pattern, as a result of the explicit synchronization between public and private copies. These semantical restrictions become an impediment to reaching optimal concurrency with shared lock mode (see [12]). Detailedly, the following common access patterns are forbidden: The remote operations concurrently access the memory (even non-overlapping) encompassed by the target window with other local operations on it. Otherwise, the outcome will be erroneous.

Comparatively, these access limitations are removed in MPI-3 with the support of RMA unified memory model. This model legalizes the above conflict scenarios and thus is perfectly fit for the DART memory system semantically. Technically, I base the DART memory system on the MPI-3 unified memory model by enabling the optimal usage of shared local access mode. This can potentially achieve the improvement in the programming productivity and performance.

3.2 Chapter summary

In this chapter, I mention the unified memory model supported in MPI-3 RMA to address the limitations in the separate memory model. These limitations substantially prevent MPI RMA from being a suitable runtime system for the PGAS programming model. With the unified memory model, I can efficiently apply the MPI passive mode which semantically matches with DART. In the chapters below I show the approaches of designing an efficient DART communication system on the basis of MPI-3 interfaces on large-scale system.

Chapter 4

Design of DART-MPI Team and Group

DART supports process grouping capability, where programmers can create groups consisting of a series of units/processes. To allow and conduct the RMA and collective communications, DART requires programmers to associate a team with a single group. In this view, DART only supports interprocess communications within a group. Once being initialized with *dart_ init*, DART generates a defined universe team – *DART_ TEAM_ ALL* – for all units. In this chapter, I present the preliminary semantics of the DART groups and teams, which plays an integral part in featuring the DART communications. I highlight the group semantics gap between DART and MPI, and present the idea of implementing DART groups and teams on top of MPI groups and communicators. I selectively take advantage of the group and communicator-related functionalities provided by MPI standard. With those functionalities, DART can provide a series of portable and scalable methods for users as well as other PGAS language implementors to describe an appropriate scope for all communication operations.

4.1 Overview of DART team and group concepts

Like MPI group, a DART group is represented by an opaque object (*dart_ group_ t*) and also defines a scope for units in point-to-point and collective communications. A DART group is an ordered collection of units, in the sense that it keeps all units in ascending order in terms of the absolute *unitID* (in *DART_ TEAM_ ALL*) attached with each unit. In this way, each unit in a DART group is associated with a globally unique ID, IDs can be non-contiguous and start from the lowest ID.

A DART team, identified by an integer ID – *teamID*, is an ordered set of units. A team is eligible to create sub-teams. Thus, a unit can belong to multiple teams simultaneously, each of which produces a relative ID for this unit. The *teamID*s do not have to be globally unique. However, the following localized uniqueness guarantees for *teamID* are listed:

1. The same ID is returned to all units composing the new sub-team.
2. The sub-teams' IDs are unique with respect to the parent team.
3. If a unit is belonging to two teams, then it is guaranteed that these two teams will obtain different identifiers.

DART applications are written in SPMD manner. By nature, an SPMD program tends to diverge into multiple branches running simultaneously on different unit sets with different inputs, which generally leads to faster results. In DART, unit sets can be understood as sub-teams. In addition, the 'master-slaver' style [43] is frequently used by SPMD to structure the parallel programs. Therefore, it will be useful for programmers to foretell the new master (normally with relative *unitID* 0) of a newly created sub-team before actually creating it. Therefore, the fact that a DART group is organized by order according to the absolute *unitID*s therein, somewhat encourages a guarantee that the unit with relative *unitID* 0 is the unit with lowest absolute *unitID*.

4.2 Overview of DART team and group interfaces

DART contains routines for accessing information on teams and for manipulating teams and groups, i.e., creating new groups or teams from existing ones, and deleting groups or teams.

All group-related operations are local. Functions *dart_group_init* and *dart_group_fini* are certain to be called when creating a DART subgroup. The *dart_group_init* initializes an empty group to allow operations on it. On the other hand, the *dart_group_fini* reclaims resources that associated with the given group to invalidate it. Two kinds of DART group arithmetic operations, named *dart_group_union* and *dart_group_intersect*, are defined in DART. They function like *MPI_Group_union* and *MPI_Group_intersect* except that the DART counterparts need to keep the output group exhibited as a sorted set based on the absolute *unitID*. Hence, both of them are commutative and associative. Obviously, these two group arithmetic operations address the need for creating subsets (subgroups) and supersets (sup-groups) of existing groups in DART. *dart_group_addmember* and *dart_group_delmember* are defined to insert/delete a unit with the given absolute *unitID* into/from the specified group. An *dart_group_split* is provided to split up the

parent group, chunk by chunk, into several sub-groups of approximately the same size on demand. An *dart_group_getmembers* returns a set of ordered absolute IDs of the units composing the given group.

Therefore, there are two mechanisms to generate a non-empty group. One is provided to build a group from scratch through the function *dart_group_addmember* by adding the desired units one by one. The others are provided to build a group from other existing groups.

Operations that access team are local. The *dart_team_myid* and *dart_team_size* are the typical team access operations, which are defined to get the relative ID of the calling unit in the specified team and the size of the specified team, respectively. The other two team access operations are *dart_team_unit_l2g* and *dart_team_unit_g2l*. They are responsible for determining the relative numbering of the calling unit in the *base group* (associated with *DART_TEAM_ALL* for DART and associated with *MPI_COMM_WORLD* for MPI) and the sub-groups. Operations that manage teams are collective. To be specific, the *dart_team_create* and *dart_team_destroy* are provided in DART to create a new team derived from the known group and invalidate the specified team. Additionally, the *base group* is naturally ordered and available through calling the function *dart_team_group*. With the functionality of *dart_group_split*, The team hierarchy for a program running on a machine of the hierarchical structure [44] could be described.

DART groups and teams are closely interacted with each other. A team is certain to involve a single group (using *dart_team_group*). On the other hand, groups are essentially helper objects representing sets of units, out of which teams can be formed (using *dart_team_create*).

4.3 Design of DART-MPI team and group

In this section, I outline the design of the mapping from DART to MPI-3 in terms of group- and team-related operations respectively. The mapping from DART functions to MPI-3 ones is somehow straightforward. However, there are several semantic gaps between them, which have to be bridged in an effective way.

4.3.1 Group-related operations

To implement the DART group operations based on MPI-3, I foremost need to substantiate the DART group structure by defining a member with the type of *MPI_Group*

in it. Therefore, inside the *dart_group_init*, I can initialize its member with the constant – *MPI_GROUP_EMPTY* which indicates an empty group pre-defined in MPI. I can begin with the empty group to create any DART group. With *dart_group_fini*, the given group can be invalidated by being set to another MPI pre-defined constant – *MPI_GROUP_NULL*.

The *dart_group_addmember* involves two parameters of *group* and *unit* as input, and then outputs an extended group containing the given unit. To be concrete, Figure 4.1 shows us the way I construct a group from scratch by using *dart_group_addmember*. Clearly, the added member (unit) is provided in a form of the absolute *unitID* and group is maintained in ascending order of the absolute *unitID* in each step. Therefore, at the end an ordered group with the unit of the lowest absolute *unitID* – 1 – as the header of the group can be obtained.

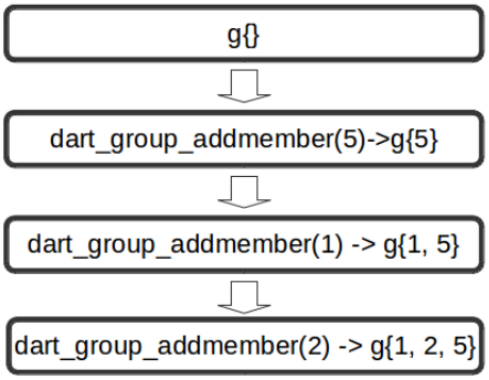


FIGURE 4.1: Schematic example of *dart_group_addmember* operation.

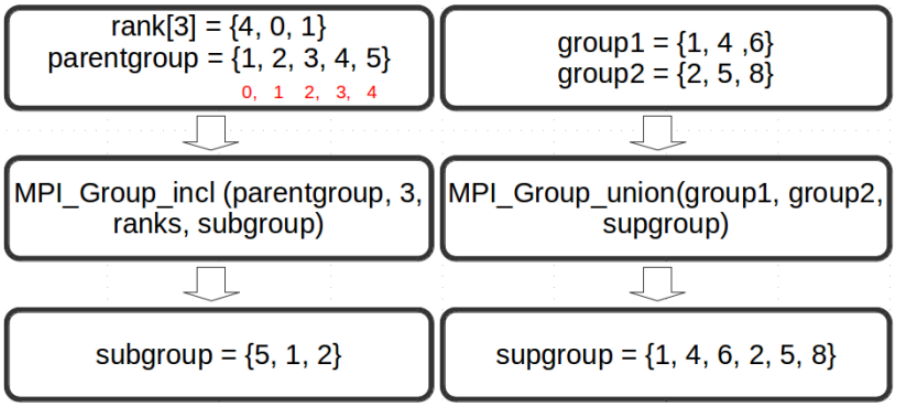


FIGURE 4.2: Schematic examples of *MPI_Group_incl* and *MPI_Group_union* operations. The absolute ranks listed in all groups are written in black, and the relative ranks listed in *parentgroup* are written in red.

Intuitively, an MPI group is created from other previously defined groups, but not from scratch. Take *MPI_Group_incl (parentgroup, n, ranks, subgroup)* for example, shown in Figure 4.2, the *subgroup* is comprised of 3 elements specified by the array *ranks* indicating 3 processes with relative IDs – *ranks*[0], ..., *ranks*[2] in the *parentgroup*. Therefore,

the process with rank 0 in *subgroup* is the process with rank *ranks*[0] in the parent group. Conceptually, this MPI group creation mechanism implies two facts that do not well-fit into the DART group. First, a sub-group is built based on the relative ranks in the parent group rather than the absolute ranks (in *MPI_COMM_WORLD*). Second, the sub-group is determined primarily by order in the *ranks*, which means the sub-group is unlikely to be maintained to be ordered. Besides *MPI_Group_incl*, these two facts also apply to the other three MPI creation functionalities, named *MPI_Group_excl*, MPI_Group_range_incl, MPI_Group_range_excl, respectively. Furthermore, as Figure 4.2 shows, the MPI group union function of *MPI_Group_union (group1, group2, supgroup)* just appends *group*2 onto *group*1, but does not guarantee the ordering of output group *supgroup* based on the absolute ranks. I can conclude that for all practical purposes, the processes in each MPI group are arranged in a random manner. In order to better understand the conceptual difference between MPI group and DART group creation, the processes in the MPI groups shown in Figure 4.2 are also represented as the absolute ranks [41].

As a result of the above differences between DART and MPI in handling groups, it is not feasible to directly map DART group to MPI group. Thus, *dart_group_union (group1, group2, supgroup)* is designed to sort a compound group consisting of the two input groups – *group*1 and *group*2 internally by using a light-weight *mergesort* [45] method. According to the concept of DART group, I can deduce that these two input groups – *group*1 and *group*2 must be both ordered in terms of the absolute *unitID*. Therefore, this *mergesort* is deemed to be an optimal approach, which proceeds with only one round of comparison. In addition, inside the *dart_group_addmember (group1, unitid)*, I firstly perform *MPI_Group_incl (base group, 1, ranks, group2)*, where the sole element of *ranks*[0] is the *unitid* given by *dart_group_addmember* and *group*2 only consists of a single process in *base group* with *ranks*[0]. Secondly I invoke *dart_group_union (group1, group2, supgroup)*, where the *supgroup* contains the unit in *base group* with the *unitid* and this unit is put in place. Therefore, the groups are guaranteed to be ordered once constructed with the interfaces of *dart_group_addmember* and *dart_group_union*.

It is more straightforward to build ordered DART sub-groups based on the orderly DART groups or MPI group constructors. Detailedly, I implement *dart_group_intersect (parentgroup, group, subgroup)* by directly using *MPI_Group_intersect* because the ordering of units in *subgroup* is identical to that in *parentgroup*. For *dart_group_delmember (parentgroup, unitid)*, I first perform an *MPI_Group_incl (base group, 1, ranks, group1)* that does the same thing as it does in the *dart_group_addmember*, then followed by an *MPI_Group_difference (parentgroup, group1, subgroup)*, where the *subgroup* excludes the unit in *base group* with *unitid* from *parentgroup*. All members in *subgroup* are in the same sequence as they are placed in *parentgroup*. Inside the *dart_group_split*

(parentgroup, n, subgroups), I can derive n subgroups from the *parentgroup* by invoking an MPI group constructor – *MPI_ Group_ range_ incl* for n times. In this instance, *MPI_ Group_ range_ incl* provides a triplet of the form (*first relative rank, last relative rank, stride*) indicating relative ranks in *parentgroup* of processes to be included in certain subgroup. To let each subgroup consist of a string of processes with consecutive relative *unitID*s/ranks in *parentgroup*, the *stride* is always set to 1. Figure 4.3 provides an elaborate example on the implementation of *dart_ group_ split*, where I try to split up the *parentgroup* into 3 *subgroups* evenly. Note, that all the above *parentgroup*s are ordered originally. When implementing the interface *dart_ group_ getmember (group,*

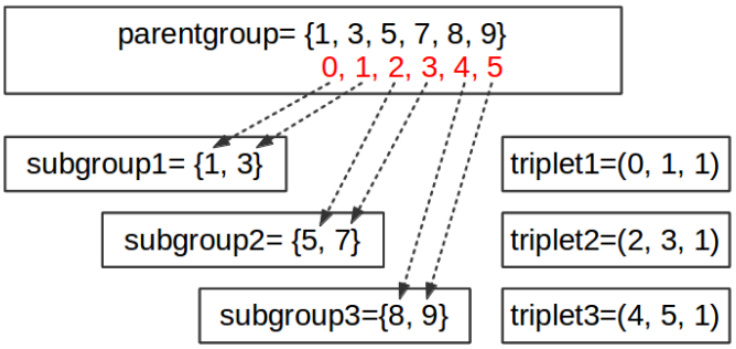


FIGURE 4.3: Schematic example of dart group split-up operation – *dart_ group_ split*. The absolute ranks listed in all groups are written in black, and the relative ranks listed in group *parentgroup* are written in red.

unitids), I first assume that there are n units in *group* and then I define another integer array *r_ ranks* indicating n processes with ranks 0, 1,..., and $n-1$, sequentially. After this, the function of *MPI_ Group_ translate_ ranks (group, r_ ranks, n, base group, unitids)* is invoked, where the output *unitids* is represented as an increasing array containing all the corresponding absolute *unitIDs*/ranks.

Furthermore, Table 4.1 concludes the mapping relationship between MPI and DART group operations briefly.

TABLE 4.1: Correspondence between DART and MPI group operations.

DART group operations	**MPI group operations**
dart_group_init	MPI_GROUP_EMPTY
dart_group_fini	MPI_GROUP_NULL
dart_group_union	MPI_Group_union+mergesort
dart_group_addmember	MPI_Group_incl+dart_group_union
dart_group_intersect	MPI_Group_intersect
dart_group_delmember	MPI_Group_incl+MPI_Group_difference
dart_group_split	Multiple MPI_Group_range_incl
dart_group_getmember	MPI_Group_translate_ranks
dart_team_get_group	MPI_Comm_group

4.3.2 Team-related operations

Team is viewed as one of the most significant concepts defined in DART. The DART team plays a similar role as the MPI communicator in communications. From the unit point of view, a team can be determined uniquely by *teamID*. Therefore, inside the *dart_team_create (parentteam, group, subteam)*, not only should I generate a localized-unique *teamID* for all units in *subteam*, but I also record the corresponding MPI communicator associated with *subteam*. According to Section 4.1, I learn that three localized uniqueness requirements with regard to the generated *teamID*, need to be satisfied. Hence, a straightforward algorithm is proposed, where it keeps a unit-local *next_availteamid* counter (is initialized as 1 since $DART\_TEAM\_ALL$ is generated with *teamID* 0) and performs a maximum all-to-all reduction collectively among all members of the *parentteam*, and the returned result $max\_availteamid$ becomes the *teamID* associated with the *subteam*. After creating the *subteam*, the *next_availteamid* on all units of the *parentteam* is set to $max\_availteamid + 1$. However, the counter $max\_availteamid$ is not touched when a team is destroyed. This implies that a *teamID* will not be adopted repeatedly after the associated team is destroyed. Therefore, *teamID*s are always unique and maintained in sustainably increasing order for each unit.

Intuitively, a table, called *teams* can be used to record the one-to-one correspondence relationship between teams and their related communicators. The table is presented as an array, where $teams[teamID]$ (indicates a certain team specified by *teamID*) is expected to store its corresponding communicator. However, it should be noted that the *teamID* may become extremely large and the size may even go beyond control. Hence, the array – *teams* has to be large enough to meet the demand for gradually increasing *teamID*. This would be inefficient for DART to maintain such a large table when simply using the *teamID* as an index to look up this table.

To mitigate the aforementioned potential problem, the solution is optimized by determining a locally-unique finite integer – *index* – for each *teamID*. However, with this solution, the amount of active teams that are allowed to exist concurrently in a DART application will be limited to a finite number, donated as MAX_NUM. Therefore, initially there are MAX_NUM available *index*s with the value ranging from 0 to $MAX\_NUM - 1$. A free-*index* linked list is created to store all the available *index*s and each node signifies an available *index*. Initially, this linked list contains MAX_NUM nodes. Technically, in order to facilitate the allocation or recycle of an *index*, the free-*index* linked list acts like a stack [46], indicates that all the access operations on this list, including the insertion of new nodes (push) and the removal of existing nodes (pop), always takes place at the same end, here refers to header node. In addition, with this access method, all pop and push operations on the linked list take constant time undoubtedly. In addition, An

allocated-*index* translation table is built to translate *teamID*s to *index*s. When a *teamID* is given, the table is searched for a match. If an entry that describes the required *teamID* is found, then I will use the *index* from that entry to locate the correct communicator that is associated with the given *teamID* in table *teams*. Otherwise, if there is no match, an error will be raised. A load of searches on the allocated-*index* table will be time-consuming if I employ the linear search method. Thus, considering the *teamID*-to-*index* lookup efficiency, I turn to binary search [46]. Binary search requires a sorted collection that allows random access. Therefore, the allocated-*index* table is presented as a static array and each element is a tuple with the form (*teamID*, *index*). The elements are logically stored in order of increasing *teamID*. Furthermore, I do not need to re-sort the table when there is a new tuple inserted. The new tuple can directly be appended onto the already-ordered table for the reason that the *teamID*s is in strictly ascending order for each unit.

Consequently, inside the *dart_team_create*, I first simply pop the header node from the free-*index* linked list and the *index* therein (*allocate_index*) is extracted to link to the newly-generated *teamID* (*new_teamID*). Second, a new record consisting of *allocate_index* and *new_teamID* is appended onto the allocated-*index* table. Finally, I make *teams*[*allocate_index*] equal to *new_teamID*. Inside the *dart_team_destroy*, I first obtain the *index* to be recycled (*recycle_index*) from the allocated-*index* table according to the given *teamID*. Second, a new node with the value of *recycle_index* is created and then pushed into the free-*index* linked list as the new header.

I set *MAX_NUM* to 5 in Figure 4.4, which instantiates the interactive activities taking place between the free-*index* linked list and allocated-*index* table. We can observe that the allocated-*index* table is maintained as an increasing ordered table based on *teamID* after the addition of two new tuples (1,1) and (2,2) or deletion of the existing tuple (1,1). Note, that *DART_TEAM_ALL* has an affinity for *index* 0.

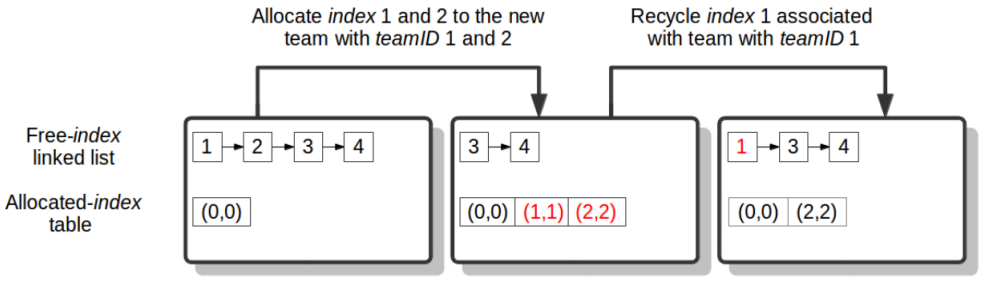


FIGURE 4.4: Interaction between the free-*index* linked list and allocated-*index* table when allocating or recycling an *index*.

The above translation between DART teams and MPI communicators make it possible to map DART team operations to MPI communicator operations properly. To implement the *dart_team_unit_l2g* and *dart_team_unit_g2l*, I employ the same method as

dart_group_getmember, except that the group corresponding to the given team should be obtained before calling the *MPI_Comm_translate_ranks*. Table 4.2 shows us the mapping relationship between DART team and MPI communicator operations in a transparent way, where we can only see the obviously invoked MPI interfaces. Other indirect MPI interfaces invoked by the DART team operations will be explained in the chapters below.

TABLE 4.2: Correspondence between DART team and MPI communicator operations.

DART team operations	**MPI communicator operations**
dart_team_create	MPI_Comm_create
dart_team_destroy	MPI_Comm_free
dart_team_myid	MPI_Comm_rank
dart_team_size	MPI_Comm_size
dart_team_unit_l2g	dart_team_get_group+MPI_Comm_translate_ranks
dart_team_unit_g2l	dart_team_get_group+MPI_Comm_translate_ranks

4.4 Chapter summary

In this chapter, I introduce the groups and teams defined in DART which are conceptually similar to the MPI groups and communicators. However, there are semantic gaps between MPI and DART in the definition of groups, communicators or teams. All units in each DART group are required to be ordered as in the group of *DART_TEAM_ALL*. Compare to the MPI communicator, DART team is presented as an integer and could be continuously increased. Note that, we cannot blame either DART or MPI for the above semantic gaps, which occurs purely for different purposes. To bridge the gaps without violating the semantics of MPI groups and communicators, I merge-sort each new created DART group and attach team with an integer (unlike *teamID*, its value can get controlled internally in DART), with which the corresponding communicator can be determined correctly. The team in DART not only identifies the scope for collective communications, but it also specifies the scope for the appropriate RMA communications happening on certain collective global memory region, which will be described in Chapter 5.

Chapter 5

Design of DART-MPI Global Memory Management

MPI *window* encompasses contiguous blocks of memory that is expected to be exposed to other processes. To allow Remote Memory Access, each process must create a window and make it visible to other processes. However, in DART, only the data items residing in the global memory block have the chance of being accessible to the remote units. As mentioned in Chapter 3, we learn that MPI-3 RMA primitives are applied as the low-level communication subtract of DART implementation system. Obviously, I need to devise a proper method handling the relations between DART global memory regions and MPI windows.

Moore's Law [47], suggests that the processor speed would double every 18 months with the effect of more integrated transistors, is truly sustained over the past decades. However, acceleration of the processor speed fails to be an everlasting workaround to satisfy the Moore's Law in the sense that we take the physical complexity and design expense into consideration. As a result, multi-core processor (also known as CMP) emerges to speed up the computation capability of processor through performing the workload among multiple cores concurrently. Multi-core processor is viewed as a simple but powerful and cost-efficient alternative to continue scaling up the processor speed. With the advantage of multi-core processor, it has been commonly deployed in the nowadays computation clusters. In addition, multi-core processor coupled with parallel programming promise increases in performance and scalability, especially for the data-intensive applications. Multi-core clusters impose new challenge into the software design in a way of highlighting the frequency and performance of intra-node data transferring. To get the optimal intra-node performance, Researchers design the parallel program with multi-core awareness and coordinate the intra-node communication design of a parallel programming model.

I.e., the on-node load/store accesses without providing extra memory-to-memory copies are encouraged, given the ubiquity of CMP.

The MPI library has been scaling and evolving to keep up with the scalable and productive computation hardware. MPI-2 incorporates the one-sided interfaces to avoid the matching affairs inherent in the two-sided operation. Moreover, the upcoming MPI-3 RMA is extended to support the shard memory window. Besides the shared memory window, MPI-3 also supports the dynamically-created window, with which the users can flexibly attach or detach the pre-allocated memory blocks repeatedly. The results provided in earlier works [15, 48] have proved that the MPI-integrated shared memory window could be a promising alternative.

In this chapter, first I map a block of global memory to two separate windows with different intentions. One window is created to enable the processes within a shared memory domain to allocate shared memory for direct load/store accesses. The other window is provided to serve the general inter-process communications via MPI RMA operations. Second, I introduce a translation table to bridge the semantics gap between DART global memory regions and MPI windows. With the translation table, the global memory is eligible to be remotely accessed correctly. Finally, I analyze the two window creation methods provided in MPI synthetically, and conduct a series of experiments to demonstrate that the preferable window has been selected considering the tradeoff between access efficiency and creation overhead.

5.1 Overview of MPI-3 window creation

Four types of windows are supported in MPI-3 RMA and created via calling the routines of *MPI_ Win_ create_ dynamic* (MPI dynamically-created window), *MPI_ Win_ create* (MPI-created window), *MPI_ Win_ allocate_ shared* (MPI shared-memory window) and *MPI_ Win_ allocate*. The *MPI_ Win_ allocate_ shared* routine creates the shared memory windows that can be remotely accessed via direct load/access instructions. The other three routines create windows that can only be remotely accessed by MPI RMA operations. In this section, I mainly explain three MPI-3 window types: MPI-created window, MPI dynamically-created window and MPI shared-memory window, as they play a dominant role in establishing and constructing the DART global memory system.

5.1.1 MPI-created window

Traditionally, the *MPI_ Win_ create* operation is collectively called to create window. Each process specifies an MPI-created window spanning the existing memory that is

supposed to be remotely accessible. This function returns a window object *win* associated with the window. A *win* can be used in the future by each process to perform RMA operations. Under the assumption that a window consists of contiguous memory blocks with equal size, MPI provides the displacement unit argument to allow programmers to specify the memory block size in bytes for each window. Note, that the offsets addressed by remote accesses are always relative to the memory base in the target window. E.g., if a remote access happens to the window exposed by process Pi at offset 0, the first memory block in that window will be touched.

According to the MPI standard [8], MPI provides two routines, called *MPI_ Alloc_ mem* and *MPI_ Free_ mem*, for allocating and freeing some implementation-specific allocated memory segments. On some systems, the performance of RMA operations on the specifically-allocated memory may be better. However, these two MPI functions are still recommended to be utilized to manipulate memory exposed for RMA when no special memory is used.

5.1.2 MPI dynamically-created window

Unlike the traditional MPI-created window, an MPI dynamically-allocated window is understood as a new concept in MPI-3 suggesting that local memory can be explicitly attached to and detached from the associated window object in an asynchronous fashion.

In MPI-3, an *MPI_ Win_ create_ dynamic* is called to generate a window object *d-win* without any memory region attached. The pre-allocated memory segments are not available to be remotely accessed on *d-win* until they are attached to *d-win* using the function *MPI_ Win_ attach*. Likewise, the memory segments can be detached with the routine *MPI_ Win_ detach*. Once those memory segments are detached from *d-win*, they will not be the target of any MPI RMA operation on *d-win* unless they are re-attached again. Notably, any local memory segment can be attached and detached repeatedly, and multiple but non-overlapping memory segments are allowed to be attached to the same *d-win*.

An *MPI_ Get_ address* is invoked to return the address of the given memory location and should be called to validate the RMA operations on *d-win*. This is due to that address of the target memory location is passed directly as window displacement to the MPI RMA operations on *d-win*. Therefore, the target process is required to send the address of a certain attached memory location that locals to it, to the origin process who intends to access it.

Figure 5.1 shows us an overview of the difference between MPI-created window and MPI dynamically-created window. An MPI-created *window* is statically linked to a block of contiguous memory. I.e., the generation of a new *win* spanning the new memory region is

enforced to only serve the RMA operations on it. Conversely, MPI dynamically-created *window* can asynchronously incorporate random memory blocks varying in address and size.

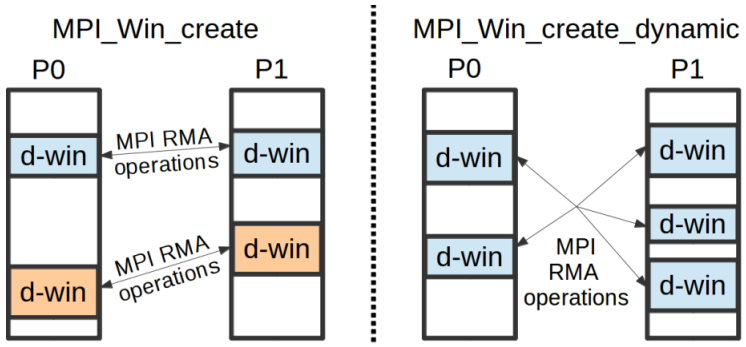


FIGURE 5.1: MPI-created window and MPI dynamically-created window layout.

5.1.3 MPI shared-memory window

The unified memory model is supported in MPI-3 to take advantage of the cache-coherence characteristics reflected in the modern hardware architectures. It is a pre-requisite for the exposure of MPI shared-memory window.

To collectively allocate the shared memory region across all processes in a given communicator, MPI-3 defines a portable interface – *MPI_ Win_ allocate_ shared* to generate a window object *shmem-win* associated with the allocated shared-memory window. In addition, the communicator on which *shmem-win* bases should be a "shared-memory capability" communicator. I.e., it is allowed to build a memory sharing region on top of such special communicator. Therefore, the function *MPI_ Comm_ split_ type*, as an extension of the function of *MPI_ Comm_ split*, identifies sub-communicators on which the shared memory region can be created with the type of *MPI_ COMM_ TYPE_ SHARED*. The function *MPI_ Win_ shared_ query* is provided to locally query the base pointers to the memory segments of the target processes. With the base pointers, the locally allocated memory can be accessed by remote processes with load/store instructions. Such access pattern can make data movements bypass the MPI layer and directly go through memory sharing, which brings in significant performance improvement.

Figure 5.2 shows an example of shared-memory windows on two separate "shared-memory capability" communicators. The memory spanned by those windows can be shared by the processes within the associated communicators with direct load/store instructions.

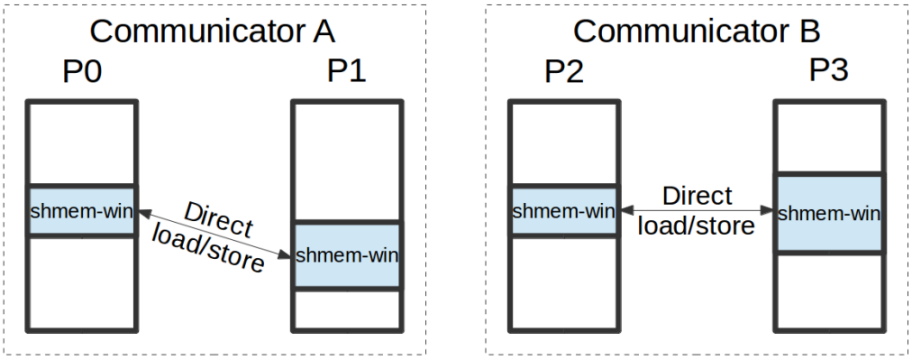


FIGURE 5.2: MPI-3 shared-memory window layout. Communicator A and B are the adequate communicators where provide the opportunity to build the shared-memory windows.

5.2 Overview of DART global memory and global pointer

Building and working with global memory is the focus of DART global memory management. DART uses several terms to identify memory address spaces and the data located in them.

The terms local and global indicate two categories of memory address space. The local memory address space of a unit is managed by the regular OS mechanisms. The data items located in it are addressed by regular pointers and only visible to the local unit. The global memory address space is a virtual abstraction, consisting of the memory segments contributed by the units of an application on demand. The data items are remotely accessible to all units or the units who contribute to them and referred to by global pointers provided by DART. To perform remote accesses to the memory space of another unit, the application needs to specify the identifier of the remote unit as well as the location of remote data on it. Therefore, the DART global pointer is organized by a 32 bit field for determining the remote unit, a 64 bit offset or local address field, a 16 bit flags field and a 16 bit segment identifier field for describing the location of the remote data [41]. Importantly, unlike UPC, the global pointer on the DART level has no phase information associated with it.

The terms private and shared describe the accessibility of data items in DART. The shared datums can be used by remote units to read, write, or perform collective operations. The private datums are however inaccessible to other units. Accordingly, I can assert that the shared datums reside in the global memory space, and the private datums are defined in the local memory space.

The terms non-collective and collective are introduced to differentiate two types of DART global memory. Thus, DART supports two approaches for allocating or freeing global memory. The *dart_ memalloc*, as a typical DART non-collective global memory allocation

call, only allocates a memory region with specified size in the global address space of the calling unit and returns a *non-collective global pointer* to it. The global memory allocated via *dart_ memalloc* are visible to all units (i.e., the memory has implicit associativity with *DART_ TEAM_ ALL*). The *dart_ team_ memalloc_ aligned*, as a typical DART collective global memory allocation call, collectively allocates memory region across the given team. This requires in-step calls from all the units encompassed by the specified team and those involved units need to call this function with the same parameters. Each unit in the given team returns a *collective global pointer* pointing to the beginning of the memory region of this allocation (on the unit with relative *unitID* 0 in the given team). Other units that are irrelevant of the given team are not even aware of this allocation.

The terms aligned and symmetric are used to feature DART collective global memory allocations. On the one hand, a global memory allocation is thought to be symmetric when this operation allocates memory with equal size on all units in the given team. On the other hand, a global memory allocation is expected to be aligned when every unit can locally compute the global pointer to any remote region by simple arithmetic. Therefore, the two mechanisms (collective and non-collective) used for handling DART global memory allocations are illustrated in Figure 5.3. This figure also shows how DART implements a PGAS model using remotely accessible and private data objects.

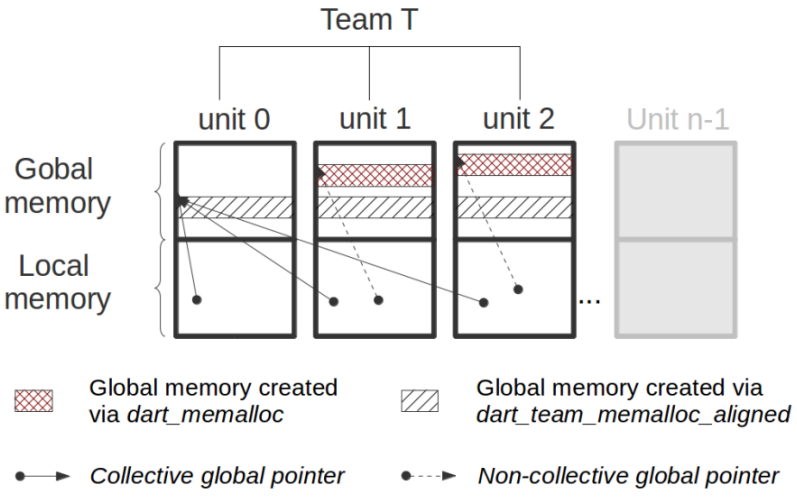


FIGURE 5.3: DART memory model. Two patterns of global memory allocation are supported in DART.

Besides the global memory allocation and freeing operations, there are also local functions for dealing with the global pointers in DART. The *dart_ gptr_ getaddr* is provided to get the local memory address pointed by the specified global pointer for the calling unit. An error will be issued if the calling unit does not have affinity with the specified global pointer. Three global pointer interfaces – *dart_ gptr_ setaddr*, *dart_ gptr_ inaddr* and *dart_ gptr_ setunit* are supported to modify any global pointer. With them, each unit

can flexibly set, increase or decrease the target memory address, or update the target unit in the specified global pointer.

5.3 Design of DART-MPI global memory management

This section, as the focus of this chapter, defines the values of DART-MPI global pointers, and designs a method of building up the DART global memory segments based on the MPI windows, with the aid of translation table. Additionally, considering the efficiency in data accesses, I devise a two-tier global memory architecture, where allows direct load/store accesses as well as the MPI RMA operations.

I assume that all the following collective global memory blocks are allocated across team T (associated with communicator $COMM$) consisting of P units.

5.3.1 Defining DART-MPI global pointer

The shared datum stored in the global memory is reached with the DART global pointers. Having a convention which clearly defines the DART-MPI global pointer is important. Conceptually, there are collective and non-collective global memory blocks in DART, pointed by *collective global pointer* and *non-collective global pointer*, respectively. Therefore, to address a DART-MPI *collective global pointer* associated with team T, the *unitid* is represented by an absolute *unitID*; the *segid* is a locally increasing positive integer number, which can be exploited by a unit to determine any collective global memory segment (has association with this unit) uniquely; the *offset* other than *addr* is utilized, to indicate the relative displacement. The *flags* is equivalent to the *index* pertaining to the team T. To address a DART-MPI *non-collective global pointer*, the *unitid* is also described by an absolute *unitID*; the *segid* is a constant 0; the *offset* rather than *addr* is used, to specify the relative displacement too; the *flags* is simply set to 0 with no special significance. Accordingly, Table 5.1 gives a general picture of the values of the DART-MPI *collective global pointer*s or *non-collective global pointer*s. Obviously, with *segid*, the *collective global pointer*s and *non-collective global pointer*s can be discriminated. Regarding the *offset*, I will describe its value specifically in the sections below.

TABLE 5.1: DART global pointer values.

Field	**Value**
segid	Non-collective: *segid*=0 Collective: *segid*>=1
flags	Non-collective: *flags*=0 Collective: *flags*=*index*
offset	Displacement relative to the start of the memory region
unitid	Absolute *unitID*

5.3.2 Two-tier DART-MPI global memory architecture

On many nowadays multi-core clusters, all units belonging to the same node can fallen into an identical sub-team, where it is allowed to build a shared memory domain collectively. It is critical for us to take the data locality into consideration even in the runtime level, given the efficiency of the intra-node data transfers. In order to acknowledge the multi-core nature of nowadays supercomputing systems, I let the DART-MPI allow for two types of data access remoteness – intra-node and inter-node.

To concurrently support the above-mentioned different degrees of data access remoteness, a two-tier DART-MPI global memory system is entailed. Intuitively, this system makes the locally-allocated memory segment (owned by a process) being accessed by other processes in the same node via direct load/store instructions and accessed by other processes in the different nodes via MPI RMA operations.

Theoretically, there are two ways to make a known memory region available for RMA operations on a certain window. On the one hand, just like the MPI-2 has done, I can generate a MPI-created window and map a block of pre-allocated memory to this window through collectively calling *MPI_ Win_ create*. On the other hand, a memory region can be attached explicitly to an existing dynamically-allocated window via calling *MPI_ Win_ attach*. Calling *MPI_ Win_ create* is costly compared with calling the local operation of *MPI_ Win_ attach* which however behaves much like a no-op [42]. Furthermore, the *MPI_ Win_ create* needs to be called every time when I want to make a certain memory region available for RMA operations. Thus, the repeated calls of *MPI_ Win_ create* will lead to great performance penalty and inconvenience. Likewise, to obliterate the remote accessibility of the memory region on this window, calls to *MPI_ Win_ detach* are cheaper in comparison with calls to *MPI_ Win_free*. Besides the above advantages of the dynamically-allocated windows, Potluri et al. [49] have conducted the experiments to demonstrate that dynamically-allocated *windows* perform as good as the traditional MPI-created windows in terms of put latency with the micro-benchmark evaluation. Therefore, the dynamically-created window is preferable

for DART-MPI, in case where a relatively-large or random amount of exposed memory regions are expected. Otherwise, the MPI-created window would be an acceptable alternative.

To construct the two-tier global memory system on team T, I first split team T into sub-teams by calling *MPI_ Comm_ split_ type* with the key *MPI_ COMM_ TYPE_ SHARED*. Direct load/store access is of course allowed within each of the above sub-teams. After this, I call an *MPI_ Win_ allocate_ shared* to allocate a memory region and create a *shmem-win* spanning the memory segments on each sub-team. Additionally, each unit needs to make its locally-allocated memory segment visible to the units in other sub-teams. One way to do so is to create a *d-win* on the *COMM* and then let each unit attach its locally-allocated memory segment to the *d-win* explicitly. The other scenario is to create a *win* spanning the above allocated memory region. Figure 5.4 shows us the general two-tier global memory system, where there are two overlapping windows sharing the same memory region for different purposes. Accordingly, the units covered by the same *shmem-win* can communicate with each other via memory sharing on *shmem-win*; the units located in different sub-teams should turn to the *d-win* or *win* for completing the remote accesses [50].

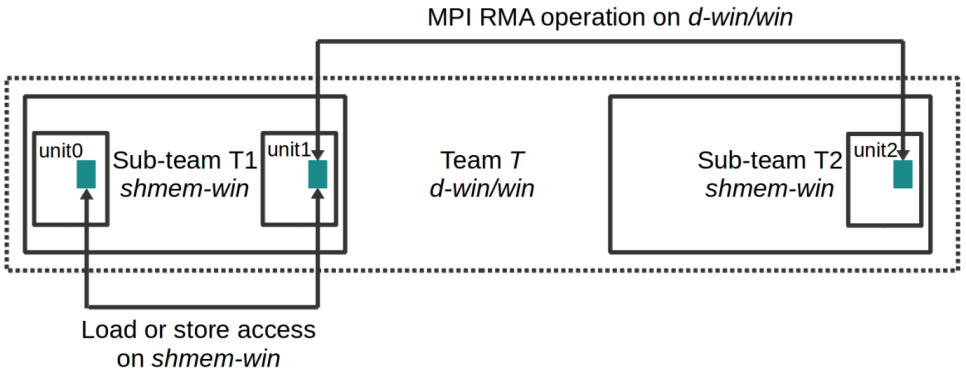


FIGURE 5.4: Two-tier DART-MPI global memory architecture. Nesting of shared-memory window inside dynamically-created window or MPI-created window.

5.3.3 DART-MPI non-collective global memory management

DART non-collective global memory allocation is, just as its name implies, a local operation. It allocates a block of globally accessible memory with given size exclusively for the calling unit. However, the instinct collective property of MPI window creation primitives leads to the participation of all processes in the given communicator. Such semantic discrepancy between DART non-collective global memory allocation and MPI window creation makes it infeasible to build a one-to-one relationship between non-collective global memory blocks and windows. Hence, to make the locally-allocated memory be

accessible to all units, all non-collective global memory blocks will be associated with pre-defined global windows exposed on the *MPI_ COMM_ WORLD* without straight translation operations.

Specifically, to apply the above two-tier architecture, all non-collective global memory blocks are placed within two pre-defined overlapping global windows. Revisiting the Figure 5.4, Once the *dart_ init* is called, I first split *DART_ TEAM_ ALL* into "shared-memory capability" sub-teams. Next, each unit allocates a memory region of sufficient size and creates a *shmem-win* spanning it and exposing it on each sub-team. Note, that such sufficient memory region is allocated and decided once the DART program is initialized, and it will not be resized dynamically in the future till being destroyed. Thereby, a *win* associated with *MPI_ COMM_ WORLD* is generated by using *MPI_ Win_ create*. The *win* exposes the existing memory region in a way of enabling the message transferring among different sub-teams. As a result, these two windows share the static memory region. The data movements can be independently implemented on them in an efficient manner.

In nature, the static memory region is originally free and reserved for each unit to allocate its own non-collective global memory blocks. Therefore, each unit takes responsible for managing the memory region separately. To minimize the external fragmentation, the buddy memory management technique [51] is chose to support the non-collective global memory management system, where it keeps track of free/allocated blocks of memory. The memory allocation or deallocation requests for memory can be sent to the system randomly. When the system receives a request for memory, it will then allocate a free block of memory with specified size or recycle the given memory block.

To be specific, the *offset*s in the *non-collective global pointer*s represent the displacements relative to the base address of the reserved memory region for each unit. Figure 5.5 shows all events that are happened when allocating DART non-collective global memory blocks on each unit. From this figure, we can see that all units manage their free regions of memory independently. The calling unit determines the *unitid* returned in the *non-collective global pointer*. Regarding the values of other field, e.g., *segid* and *flags*, we can refer to Section 5.3.1.

5.3.4 DART-MPI collective global memory management

Unlike the DART non-collective global memory allocation operations, the DART collective global memory allocation operations match the MPI window creation operation semantically. Therefore, the one-to-one correspondence relationship between DART collective global memory blocks and MPI windows can be established dynamically.

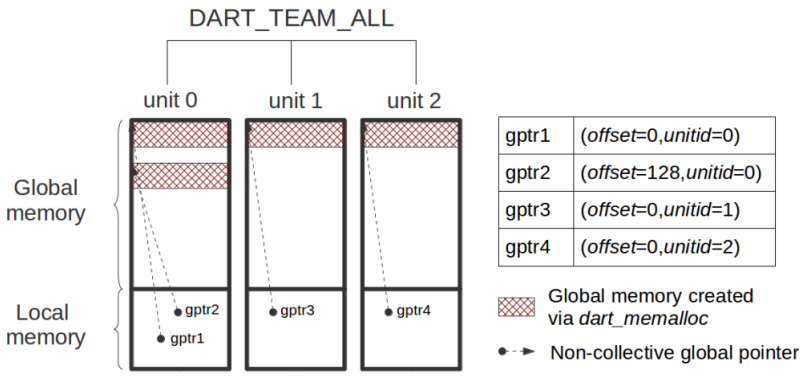


FIGURE 5.5: A schematic of DART non-collective global memory allocations. Each event makes request for memory with 100 bytes in size.

Fundamentally, the collective global memory blocks exposed on team are managed by following the above two-tier global memory architecture. Specifically, when a new team T is created, Team T is first split into sub-teams on which it is possible to enable communication via sharing memory. The collective global memory blocks exposed on team T can be allocated and deallocated on demand. Therefore, a *d-win* associated with team T (spans empty memory) is generated along with team T, indicating one-to-one relationship is built between *d-win* and team T. Such relationship is stored in an array named with *dart_win_lists*. Therefore, the *index* associated with team T can also be a perfect index into the array *dart_win_lists*. The *d-win* can potentially be utilized to complete all data movements where the units are located in different sub-teams. Given a collective global memory region (exposed on team T) of given size is allocated, I need to create a *shmem-win* spanning the memory of the specified size exposed on each of the above expected sub-teams. Here, each *shmem-win* is generated to validate all message transfers happened within the related sub-team. On top of that, each unit of the team T should attach the locally-allocated memory segment to the *d-win* explicitly to make it accessible to the units in the different sub-teams, by using *MPI_Win_attach*.

According to Sect. 5.3.1, we learn that a collective global memory segment can be determined uniquely based on *segid* in the related *collective global pointer*. Thereby, the translation between collective global memory blocks and windows actually manifests as the translation between *segid*s and window objects – *shmem-win*s. Therefore, a translation table should be introduced to visualize the relationship between *segid*s and *shmem-win*s for each unit. To be consistent with the definition of *segid* in *collective global pointer*, *segid* can be utilized as the key of the translation table. As a result of the locally uniqueness of the *segid*s, a sole translation table is bonded with each unit in the lifetime of a DART program. I.e., each unit uses a sole translation table to record all collective global memory segments exposed on all possible teams. A region of collective global memory exposed on team T determines a unique *segid* for each unit in team T upon allocated.

Besides, the related *shmem-win* is also included into the translation table.

In addition, Sect. 5.1.2 tells us that, the address of locally-attached memory segment for a unit can be used by other units as a remote access entry to it. In order to let all locally-attached memory segments be ready for remote accesses within the *d-win* by all units in team T, each unit in team T is involved in a collective operation – *MPI_ Allgather* , which collects the beginning address of the memory segments that local to other units. Thus, an array *disp-set* with size of P is defined in the translation table to store those separate addresses, which are represented as the data with the type of *MPI_ Aint*. As an example, when unit i in the team T is targeted with a *collective global pointer*, then the i-th item in the related *disp-set* is then obtained and can be utilized in the future to locate any memory location of unit i within the *d-win*. Likewise, another array *baseptr-set* is entailed to record the address for local memory segments and the process-local address for remote memory segments within the *shmem-win*. The size of array *baseptr-set* for each unit equals to the number of processes spanned by the *shmem-win*. Therefore, once a region of collective global memory is created, the corresponding *disp-set* and *baseptr-set* as well as *shmem-win* are entered into the translation table. However, the usage of dynamically-allocated window implies an obvious drawback, that is, an *MPI_ Allgather* working on the *MPI_ Aint* data is added and will bring extra overhead. The *offset* returned in the generated *collective global pointer* is always initialized to 0, signifying the base of the allocated memory block. Generally, the *offset*s in the *collective global pointer*s mean the displacements relative to the beginning address of the collective global memory regions.

Figure 5.6 reflects the events that happen when the collective global memory regions are created randomly. Here, assuming that *DART_ TEAM_ ALL* includes three units and a sub-team with *teamID* 1 (pertaining to *index* 1) consists of two units with absolute *unitID* 1 and 2. First, I allocate a region of collective global memory exposed on sub-team 1, then unit 1 and 2 return a *collective global pointer* (gptr1) pointing to the beginning address of the allocated memory region. Meantime, the *shmem-win*1 is created to associate with the allocated memory region. Second, I allocate another collective memory region exposed on *DART_ TEAM_ ALL*. This is accompanied by the creation of *shmem-win*2. Take unit 2 for example, two records including the *segid*, associated *disp-set*, *baseptr-set* and *shmem-win* are formed and inserted into its translation table. We can observe that the *segid* returned in the *collective global pointer* is always a unique integer. Regarding the *flags*, we can refer to the Fig. 4.4. Note that, the *unitid* returned in the *collective global pointer* for each unit is identical and denotes the unit with relative *unitID* 0 in the associated team (lowest absolute *unitID*). Moreover, the returned *unitid* is always presented in the form of absolute *unitID*.

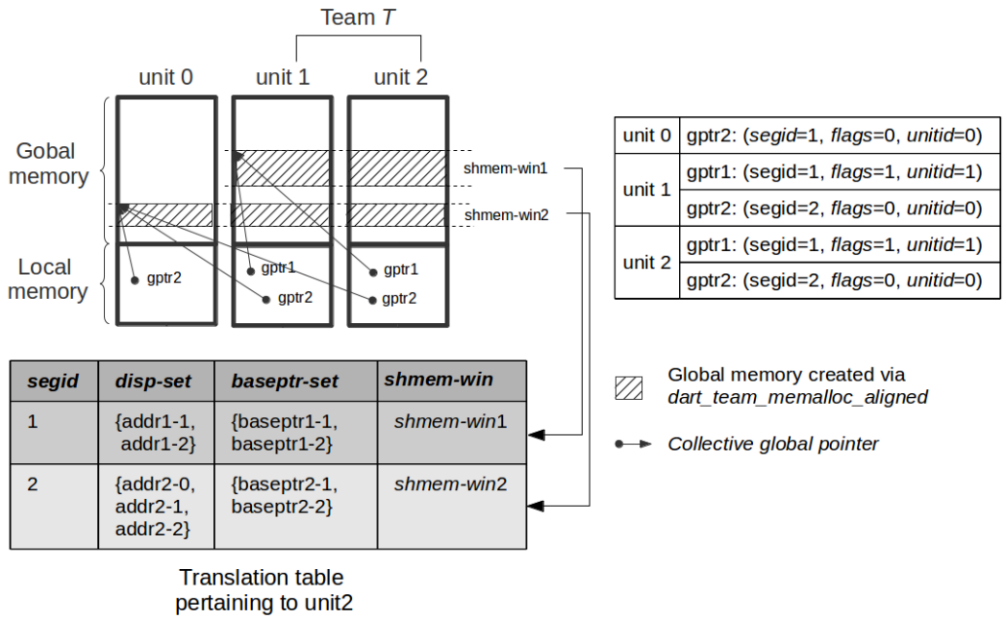


FIGURE 5.6: A schematic of DART-MPI collective global memory allocations. The elements in *disp-set* and *baseptr-set* are generally represented as addri-j/baseptri-j where i signifies the segment ID and j signifies the absolute target *unitID*. Note that the units in team T are located in the same node and unit 0 is placed in a different node.

5.3.5 Trade-off between using *d-win* and *win* for collective global memory management

This section presents experimental results and uses two micro-benchmarks to weigh up the cost of *d-win* with that of *win* when applying them to the DART collective global memory management. In the tests, the processes are grouped in blocks of 24 per node and the results presented are the average time over 2000 measure rounds. Each round ends with an *MPI_ Barrier* aiming to make sure all processes have finished the measured operation. The measurement errors are estimated from the statistical standard deviation, which is typically less than 10%. In addition, several rounds are run as a warm-up before the actual timing phase. The warm-up rounds are done to eliminate the initialization overheads from the actual measurement. Note, that all benchmarks are compiled with the Cray complier.

Theoretically, when allocating a region of collective global memory, there are two scenarios: one where *d-win* is used and one where *win* is used. On the one hand, to enable the remote communications on *d-win*, an *MPI_ Allgather* collective operation is entailed to collect the address (be expressed as an integer data with the type of *MPI_ Aint*) of the newly-allocated memory segment from each involved process. On the other hand, if *win* rather than *d-win* is applied, the operation of *MPI_ Win_ create* needs to be invoked instead.

The first test evaluates the variance in overhead of *MPI_ Win_ create* in terms of different message sizes. Figure 5.7 reflects the performance of *MPI_ Win_ create* with 8 running processes and presents an ascending curve. I.e., the overhead of *MPI_ Win_ create* grows steadily as the message size is increased and is lowest at the window size of 1 bytes. Next, I compare the performance of *MPI_ Allgather* as well as *MPI_ Win_ create*, as the number of processes is increased contiguously on Cray XC40 system (refer to Appendix A). The results shown in Figure 5.8 show such evaluation. This evaluation assumes that the transferred messages in *MPI_ Allgather* are of *MPI_ Aint* type with the size of 8 bytes on Cray XC40 system, and the created window size is always set to 1 bytes in *MPI_ Win_ create*. Obviously shown in Fig. 5.8 the *MPI_ Allgather* can consistently perform and scale much better than *MPI_ Win_ create* for the varying process counts, let alone when the created window size is larger than 1 byte. As far as the collective global memory management concerned, using *d-win* can produce some extra overhead in a certain degree. Nevertheless, the evaluations prove that using *d-win* makes window a better fit for the DART-MPI collective global memory management than using *win*.

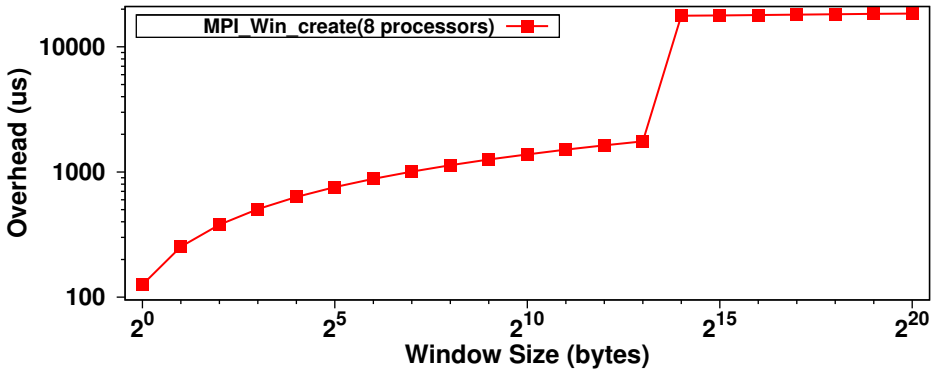


FIGURE 5.7: The variance in *MPI_ Win_ create* overhead.

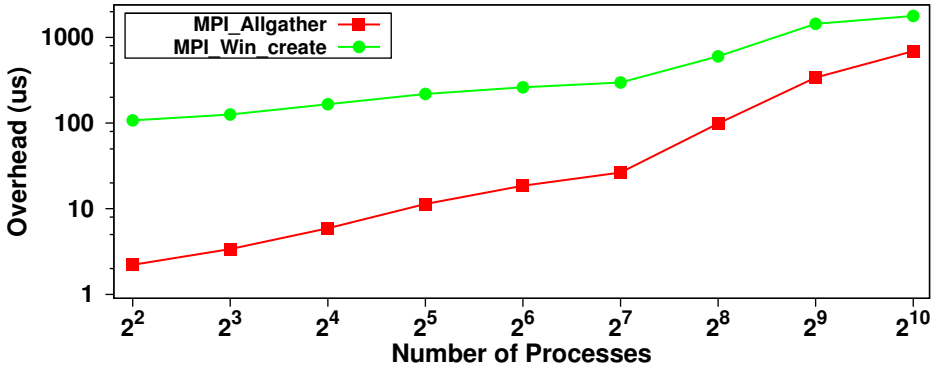


FIGURE 5.8: A comparison of *MPI_ Allgather* and *MPI_ Win_ create* in time performance.

5.4 Chapter Summary

In this chapter, I first explain two new MPI-3 window types: dynamically-allocated window and shared-memory window. With dynamically-allocated window, the repeated coarse-grained window creation/destroy operations can be avoided effectively and be replaced by multiple local attach/detach operations. The shared-memory window can be leveraged to enable the shared-memory programming for intra-node case. Second, I introduce the concept of collective and non-collective global memory regions defined in DART. The global memory is allocated to be accessed by remote units and thus conceptually matches the MPI window within certain access epoch. With a global pointer (collective or non-collective), the desired remote memory location can be determined precisely. My contributions are reflected in three aspects below:

- define and describe the DART global pointers (collective or non-collective) by giving a determined meaning to each field therein.
- draw up a two-tier global memory architecture by nesting shared-memory window inside dynamically-created window or MPI-created window according to the type of global memory.
- conduct a comparative study to prove that using dynamically-created window for DART collective global memory management exhibits better performance than using MPI-created window.

Essentially, the work in this chapter forms a fundamental ground for designing DART-MPI RMA communication operations, which will be discussed in the Chapters 6 and 7.

Chapter 6

Design of DART-MPI Communication Operations

Recently, the applications in scientific domains have been witnessing a rapid growth of communication-intensive as well as computation-intensive over these years [52]. Furthermore, an in-depth profiling study [53] is conducted to assess the performance of supercomputing systems on data-intensive application using the Graph 500 benchmark [54]. This study demonstrates that communication overhead accounts for up to 80% of the execution time of the application, which is apparently communication-dominated. Equally, on the road to constantly pursue better scaling and parallelism, the communication performance matters more. This is also shown in this study. Hence, high-efficient computation and communication performance is entailed in gaining a scaling application. The emergence of multi- or many-core processors fuels an explosive growth of processing capability in current-generation high-end systems. According to the new TOP500 Supercomputer report [1], of the top one Supercomputers as of June 2015 – Tianhe-2, exemplifies this fact with core counts exceeding 3,000,000. To achieve the best use of computing cores, a raw problem is suggested to be divided and then spread among those distributed cores, leads to substantial computation speedups. However, the improvement in communication performance fails to keep pace with that in computational capability.

All in all, the performance of parallel applications closely depends on the communication performance. Putting efforts on the improvement of communication performance is essential and will pay off. This, coupled with the prevalence of MPI, make MPI researchers conduct long-term optimization works on the MPI point-to-point [55], RMA [56–59] and collective operations [60–65], respectively.

DART – as a PGAS runtime system – is designed to provide a number of communication primitives including one-sided and collective operations. Given the high-portability of

Table 6.1: DART RMA operations.

DART RMA Calls			**Guarantee**
Communication	blocking	put	return till the completion at both the origin and target
		get	
	non-blocking	put	return directly without waiting for the completion at either the origin or target
		get	
Synchronization	dart_wait		return when the requested data transfer/s is/are completed both at the origin and target

MPI, DART communication operations are thoroughly implemented based on the MPI-3 communication model. Besides, the well-optimized MPI-3 communication operations can definitively facilitate the DART implementation and profit the DART communication performance as well.

In this chapter, first I outline all DART communication and synchronization operations, their functionalities and the completion guarantees offered by these operations. Second, I discuss the methods of dealing with the semantic mismatches, e.g., how to accurately hide the MPI-specific RMA synchronization operations inside of the DART interfaces from the DART users. Meantime, on top of the two-tier global memory system, I exploit the data locality-aware design for DART RMA operations with the purpose of letting DART achieve high performance (e.g., low latency and high bandwidth) for remote accesses. Third, I provide the correspondence relationship of RMA communication and synchronization interfaces between DART and MPI. Fourth, I introduce the dereference method for *non-collective global pointer* and *collective global pointer* with the aid of translation table. Finally, I analyze the performance of DART RMA communications with micro-benchmarks and an application kernel – Random Access (RA). In the experiments, not only I compare the DART RMA and the native MPI, but I also compare it with two other typical PGAS languages – UPC and OpenSHMEM.

6.1 Overview of DART RMA and collective operations

The DART communication class contains functions that perform one-sided and collective communications. On the one hand, DART one-sided communications are classified into blocking (i.e., *dart_get_blocking* and *dart_put_blocking*) and non-blocking (i.e., *dart_get* and *dart_put*) communication operations. In addition, DART provides the one-sided synchronization operations, i.e., *dart_wait* and *dart_waitall*, for non-blocking communication operations to ensure the completion of the data transfers. Here I briefly outline the semantics of the DART RMA operations (the completion guarantees are offered) in Table 6.1. On the other hand, DART collective communications are blocking,

and provided for synchronization or data exchanges within a team, such as *dart_ bcast*, *dart_ scatter*, *dart_ barrier* and so on. The semantics of DART collective routines is the same as that of MPI blocking collective counterparts.

6.2 Design for DART-MPI RMA operations

The MPI RMA communication operations coupled with explicit passive synchronization operations (with shared lock) form a theoretic foundation for building the DART RMA model, as stated in Sec. 3.1. 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, I need to fill the semantic gap between MPI RMA and DART RMA in terms of communication and synchronization operations.

On the one hand, 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*). In the next chapter I use an asynchronous progress engine to complement the DART non-blocking RMA communications. This can help DART to support truly asynchronous RMA communications.

On the other hand, 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* that have been mentioned in the background), 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 synchronization calls embedded is in direct proportion to that of

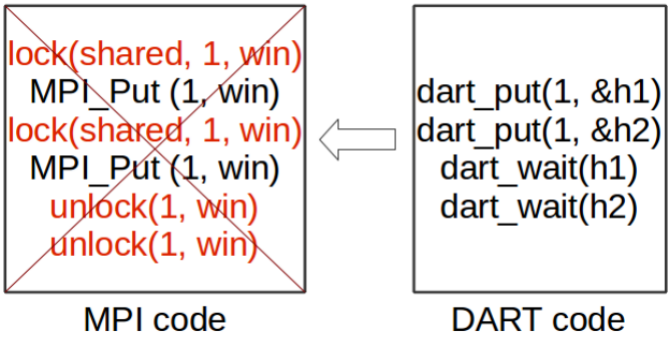


FIGURE 6.1: An unsupported overlapping scenario based on MPI single lock operations.

MPI RMA communication calls. In fact, the MPI RMA synchronization operation adds a substantial overhead [66]. 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 6.1 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. 6.1 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.

MPI-3 supports a global lock model providing the lock_all/unlock_all 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, I will explain how to safely expose these two types of global memory independently based on this method.

Chap. 5 tells us that a global *win* (MPI-created window object) and a global *shmem-win* (shared memory window object) are created once a DART program is initiated. 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, I 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. Compared 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 I 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.

However, MPI wait routines (e.g., *MPI_ Wait*) only wait on the specific request handles produced by the MPI request-version RMA communication operations. Importantly, the request-version operations are only valid within the access epoch with MPI passive target mode, which does not conflict with DART semantics. As mentioned before, DART adopts the MPI RMA passive target mode rather than the active mode. Compared to the *flush* synchronization method, this provides a mechanism allowing for the finer-grained operation completion which seems a natural fit for the DART blocking RMA operations in semantics. However, using wait routines can only guarantee local completion at the origin. I.e., it can guarantee that the local buffer at the origin can be reused for *MPI_ Rput* and the remote data has been delivered to the local buffer for *MPI_ Rget*. This implies that an *MPI_ Win_ flush* needs to be invoked before *MPI_ Wait* to ensure the remote completion of data transfer triggered by an *dart_ Rput* operation. As for *MPI_ Rget* operation, an invocation of *MPI_ Wait* suffices to ensure its remote completion. Therefore, an *MPI_ Put* followed by an *MPI_ Win_ flush* is more straightforward and efficient for implementing DART blocking put operations.

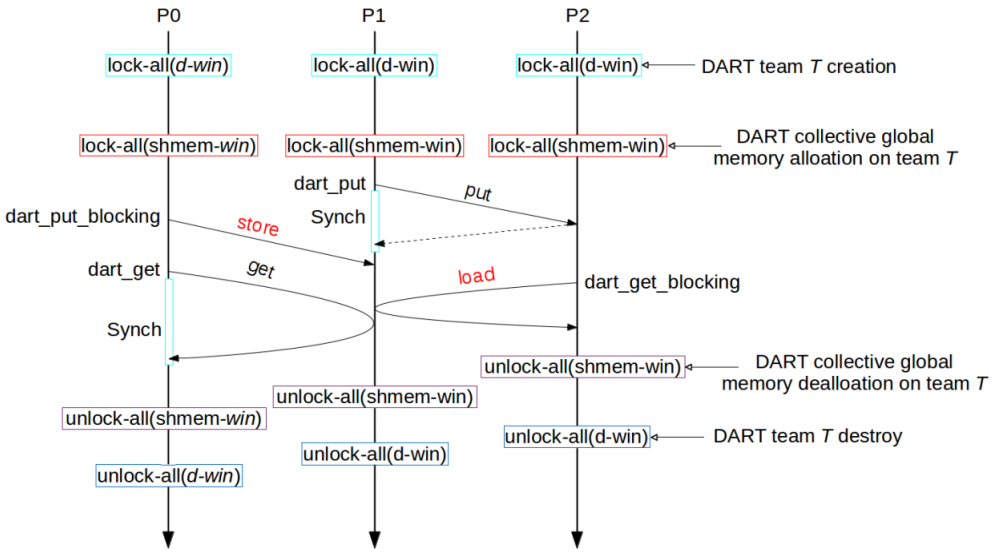


FIGURE 6.2: Example of DART-MPI RMA operations within two passive epochs. Here, the two passive epochs are started by a call to *MPI_ Win_ lock_ all* and ended by a call to *MPI_ Win_ unlock_ all*. The Synch means the operation of *MPI_ Win_ flush*.

TABLE 6.2: Correspondence between DART and MPI RMA operations.

DART RMA Calls			**MPI RMA Calls**
Communication	blocking	put	inter-node: MPI_Put+MPI_Win_flush
			intra-node: store instructions
		get	inter-node: MPI_Rget+MPI_Wait
			intra-node: load instructions
	non-blocking	put	MPI_Put
		get	MPI_Get
Synchronization	dart_wait		MPI_Win_flush

The DART synchronization routines are expected to ensure the remote and local completion of DART non-blocking RMA communications. 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*. In addition, Figure 6.2 thoroughly shows the exemplary DART RMA events happening on the collective global memory region across team T with MPI global lock mechanism. Here I assume that team T consists of three processes locating within one node.

When possible (i.e., intra-node), the shared-memory windows described in Fig. 5.4 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*. Hence we can observe that DART spontaneously has data locality in mind when performing the RMA operations. Accordingly, the correspondence relationship between the DART RMA and MPI RMA operations is synthetically delivered in Table 6.2.

6.3 Design for DART-MPI collective operations

I implement the DART collective interfaces straightforwardly by using the MPI-3 blocking collective counterparts. Note, that the correct communicator needs to be acquired before calling the corresponding MPI-3 collective routine. It can be obtained by first getting the correct *index* according to the given *teamID* and then referencing the *teams* based on the *index*.

Furthermore, a previous optimization [67] on the MPI broadcast operations are also able to be applied to the DART broadcast implementation. This optimization improves performance by saving bandwidth use in cases where transferring long messages or cases where transferring medium messages with non-power-of-two processes.

6.4 DART-MPI global pointer dereference method

All the DART one-sided communications are performed based on the DART global pointers. In detail, a DART global pointer identifies a remote data location with its members – target unit (*unitid*), segmentation ID (*segid*), a specific *offset* and *flags*. The significances (see Sec. 5.3.1) assigned to the members determine the DART global pointer dereference method. Additionally, the type of DART global pointer (collective or non-collective) is differentiated via the value of *segid*.

Note, that the relative target ranks in the given communicators are entailed for launching the MPI RMA operations. Therefore, MPI one-sided operations – put and get are both performed on relative ranks (*unitID*s) in the given communicators (teams). However, the DART one-sided communication operations work on absolute *unitID*s. To use MPI as an underlying communication substrate, this gap necessitates the translation from DART absolute ranks/*unitID*s to the relative ranks/*unitID*s in the given communicators/teams.

Given there is a *collective global pointer* associated with team T, its dereference method is elaborated in Figure 6.3. This figure shows a critical operation which translates the absolute *unitID* 6 to the relative *unitID* 3 in T. In the case of intra-node communication, the dereference procedure is showed in the yellow path. I.e., I query the appropriate *shmem-win* that covers the expected target location from the translation table based on

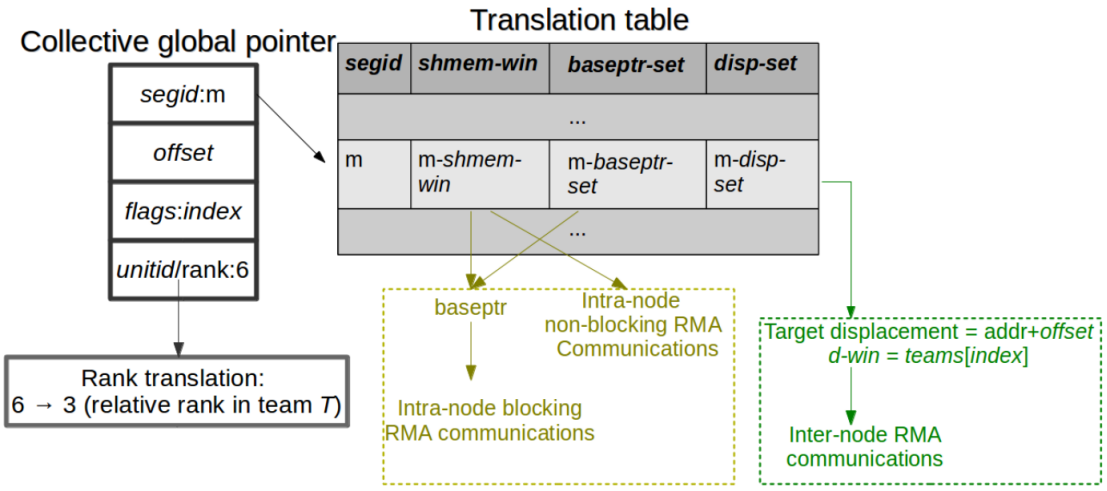


FIGURE 6.3: Description on the dereference of a *collective global pointer*.

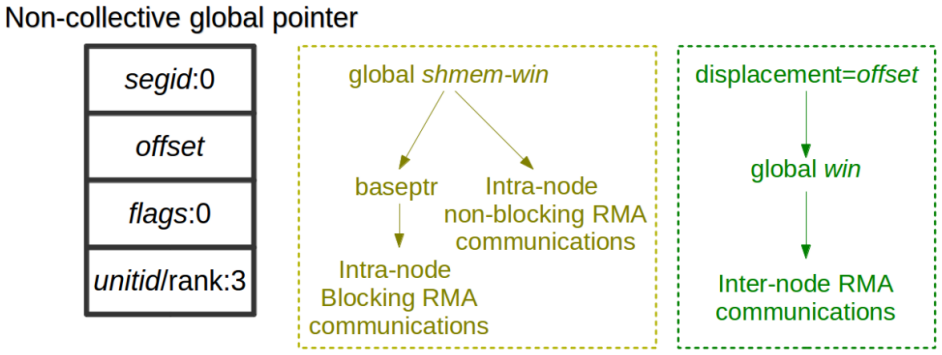


FIGURE 6.4: Description on the dereference of a *non-collective global pointer*.

segid. In the blocking scenario, the pointer (visit the *baseptr-set* in translation table) coupled with *offset* enable correct read or write operations. Otherwise, I directly issue the MPI RMA operations on *shmem-win*. In the case of inter-node communication, the blue path is followed. In this sense, I firstly query the *disp-set*, which indicates the beginning address of the window segment of each unit in team T, from the translation table according to *segid*, then get the correct *d-win* from the array *dart_win_lists* based on *index* (equivalent to *flags*). Finally I access the remote data by calling MPI RMA operations pertaining to *d-win*, where the value of $offset + m\text{-}disp\text{-}set[3]$ is passed as parameter *target_disp*.

Unlike the *collective global pointer*, an *non-collective global pointer* is generated to remotely locate the global memory spanned by two pre-defined overlapping windows on *MPI_COMM_WORLD*. Therefore, they can be trivially dereferenced without the unit translations. Besides the absence of unit translation, a *non-collective global pointer* is closely associated with a MPI-created window instead of dynamically-allocated window to perform the inter-node RMA operations. Hence, I can neglect the query into the translation table when referring to the non-collective global pointer as well. Such dereference

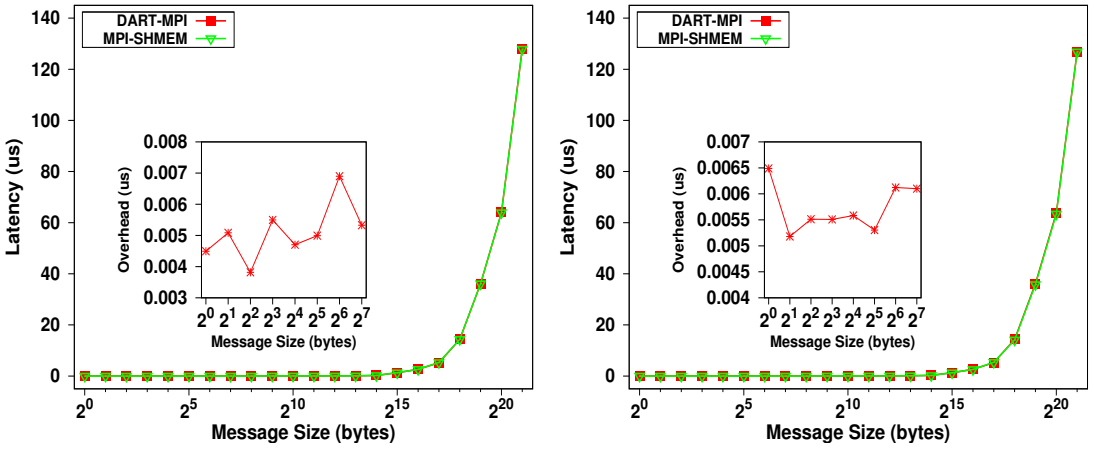
flow diagram is graphically sketched in Figure 6.4.

6.5 Performance evaluation

I now present all performance evaluation experiments and analyze those results thoroughly. A set of benchmarks, including micro-benchmarks and the widely-studied Random Access benchmark, are conducted fully in this section. All benchmarks are carried out on the Cray XC40 system. Foremost, I am interested in the implementation overhead of DART intra-node blocking RMA operations with respect to the pure MPI-3 RMA shared-memory extensions. I then investigate the performance features of the DART-MPI (combines MPI-3 RMA shared-memory extensions and RMA operations) and compare it to the native Cray MPI's RMA communication. In addition, I compare DART-MPI with two important HPC PGAS implementations: UPC and OpenSHMEM, which are both fully implemented and tuned on the Cray XC40 system. All low-level communication benchmarks are performed based on OSU Micro Benchmark [68] and averaged over 10000 executions. I do not show the error bars for their plots, as the standard deviation from the mean is always relatively small.

6.5.1 The DART intra-node blocking RMA versus the shared-memory RMA operations – implementation overhead

To be fair, Figure 6.5 shows the DART implementation overhead by comparing the latency of intra-node blocking RMA operations using tests written with DART interfaces



(A) Blocking put (intra-node) (B) Blocking get (intra-node)

FIGURE 6.5: Latency comparison between DART intra-node blocking RMA and MPI-3 shared-memory RMA operations.

and MPI-3 shared memory extensions, respectively. Clearly illustrated by the inner plot in Fig. 6.5, the overhead for DART intra-node blocking put implementation is very low with a standard deviation of a few percent only. Similarly, DART intra-node blocking get implementation gains a very small overhead (around 5 ns) in general. Therefore, the implementation overhead in DART is not noticeable with respect to the native RMA communication supported with direct load/store accesses in MPI. Additionally, such marginal performance difference will even be negligible when the larger messages are transferred, as shown by the outer plot in Fig. 6.5.

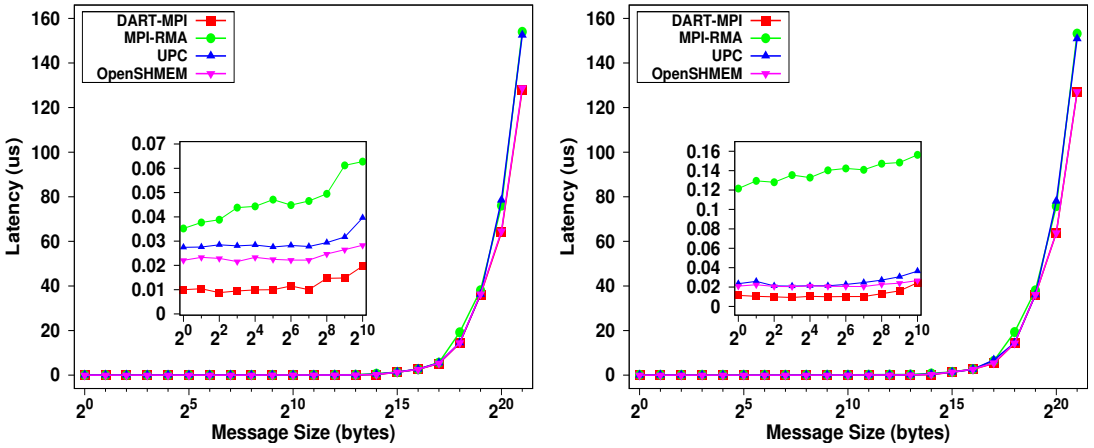
6.5.2 Latency and bandwidth evaluations

In this section, I basically assess the raw communication performance in two typical cases, named inter-node (communication across nodes) and intra-node (communication within node), respectively. I first test the average latencies of blocking operations of DART-MPI and the counterparts of blocking operations of MPI, UPC and OpenSHMEM as well. Second, I evaluate how the blocking put and get operations perform when increasing logical distance between two involved processes. On the other hand, I get the bandwidth results by means of proceeding streamed non-blocking RMA operations, referred to the "flood" test [6], where the sender sends a large number of overlapped get/put requests and synchronizes all of them in the end with a single synchronization operation. This test is useful in revealing the maximum achievable bandwidth of the communication system.

Comparing latency and bandwidth between DART-MPI RMA and MPI RMA communication with passive target mode is somehow unfair since DART wraps the heavy-weight synchronization calls (i.e., global lock calls) inside the global memory interfaces. This means they are excluded from any DART RMA-related calls. Therefore, as for MPI tests, the lock synchronization calls are not timed.

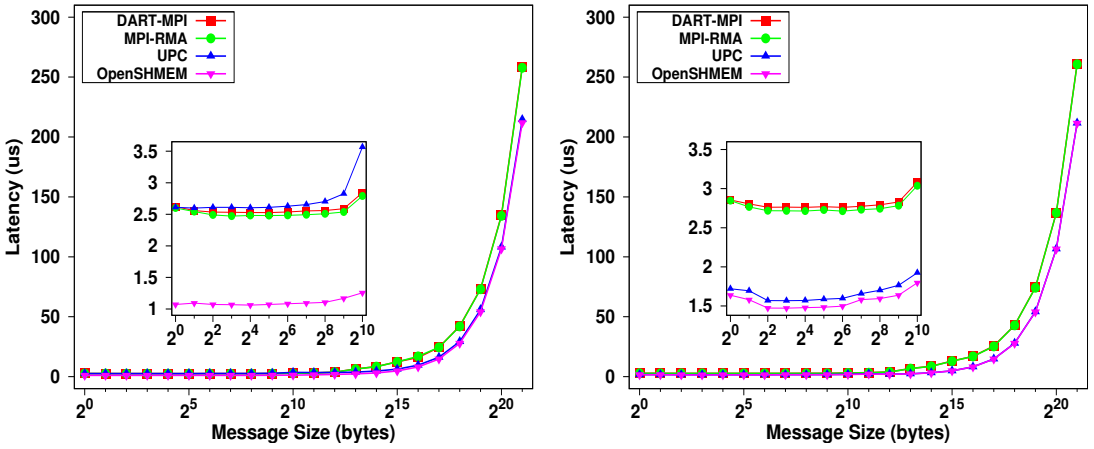
6.5.2.1 Latency

Since RMA model decouples the communication and synchronization calls, all latency results are gathered by guaranteeing both the remote and local completion with synchronization. I measure MPI-3 RMA with the *MPI_ Put* or *MPI_ Rget* communication operation and followed by the *flush* synchronization call (all the communications happen within the window allocated by calling *MPI_ Win_ allocate*). As for UPC, I use a single shared array and the blocking function – *upc_ memput* or *upc_ memget*. In OpenSHMEM, I define a global/static array of sufficient size of each process, and then issue the blocking RMA interface, i.e., *shmem_ putmem* or *shmem_ getmem*.



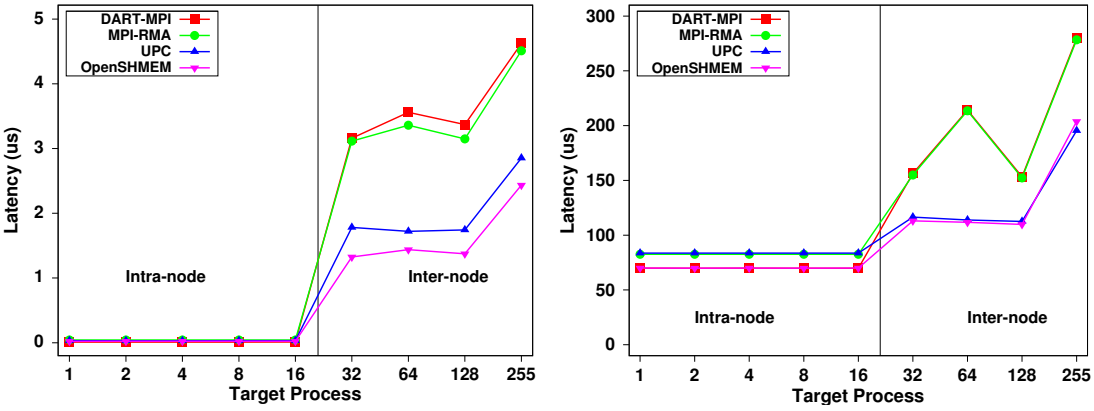
(A) Blocking put (intra-node) (B) Blocking get (intra-node)

FIGURE 6.6: Intra-node blocking put/get latency on 2 cores.



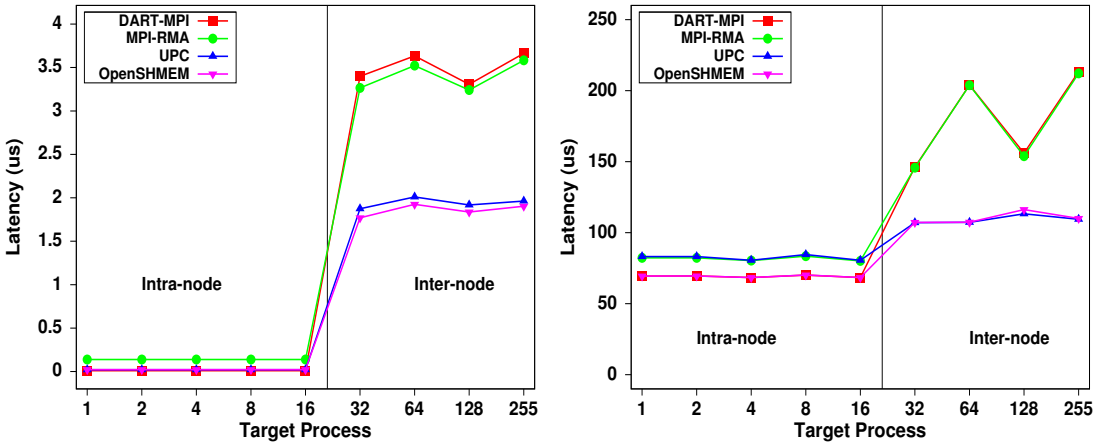
(A) Blocking put (inter-node) (B) Blocking get (inter-node)

FIGURE 6.7: Inter-node blocking put/get latency on 2 cores.



(A) Blocking put (8 bytes) (B) Blocking put (1 Mb)

FIGURE 6.8: Blocking put latency as a function of logically increasing distance between two involved processes on 256 cores.



(A) Blocking get (8 bytes) (B) Blocking get (1 Mb)

FIGURE 6.9: Blocking get latency as a function of logically increasing distance between two involved processes on 256 cores.

Figures 6.6 and 6.7 show the average latencies performance of intra- and inter-node blocking put and get operations for the message sizes ranging from 1 byte to 2 MB. A fact, that is DART-MPI can get significant benefit from the usage of shared-memory window, is clearly shown in Fig. 6.6. For the intra-node communication, it is observed that DART-MPI can achieve up to 93.1% improvement in latency for put and 78.7% for get compared with native MPI implementation and has ∼50% lower latency than the UPC and OpenSHMEM for the blocking RMA operations at best. For the inter-node communication, DART-MPI and MPI performs worse than OpenSHMEM in all cases. Specifically, UPC put has higher latency than MPI and DART put when transferring small message across nodes. DART-MPI curves are almost consistent with the native MPI. This is due to that DART-MPI RMA turns to the MPI RMA operations rather than direct load/store accesses when working on the distributed nodes.

Next, I evaluate the performance of blocking RMA operations in a way of letting process 0 send messages of fixed size to the target processes varying from 1 to 255. Note, that the job consists of 256 processes in total, which corresponds to 11 nodes on Cray XC40 system. Figures 6.8 and 6.9 show the performance variance of blocking put and get operations for short message of 8 bytes and long message of 1 MB as a function of logically increasing distance between the origin and target process.

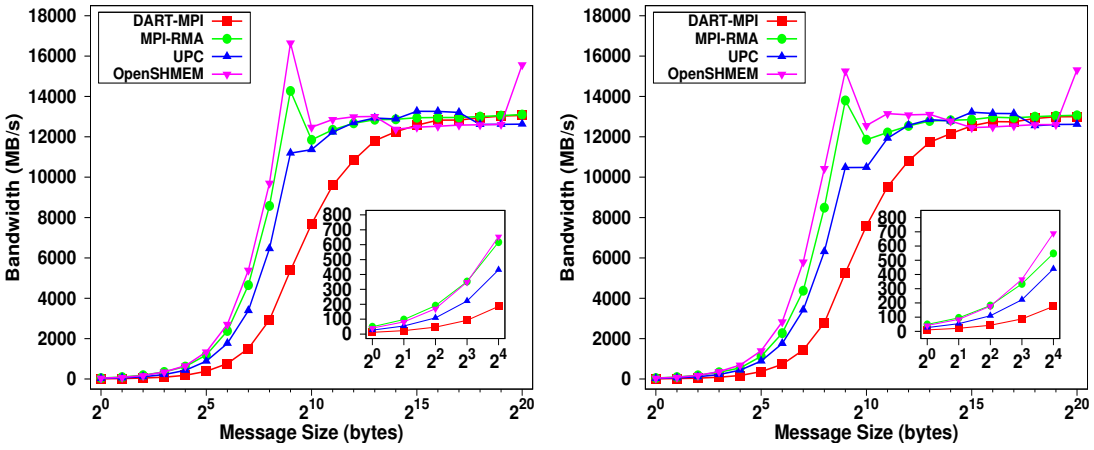
As expected the latency keeps constant when messages are transferred within one node. However, all latency curves rise substantially at a logical distance between 16 and 32, where the process 0 starts targeting a different node. The curves for DART-MPI closely sit above those for native MPI in inter-node scenario. This is due to that DART-MPI falls back on MPI RMA interfaces when communicating on distributed nodes. Those comparison curves offer a preliminary proof that DART-MPI RMA is competitive for

intra-node blocking communications while exhibiting the similar performance as the native MPI inter-node RMA, in terms of latency.

6.5.2.2 Bandwidth

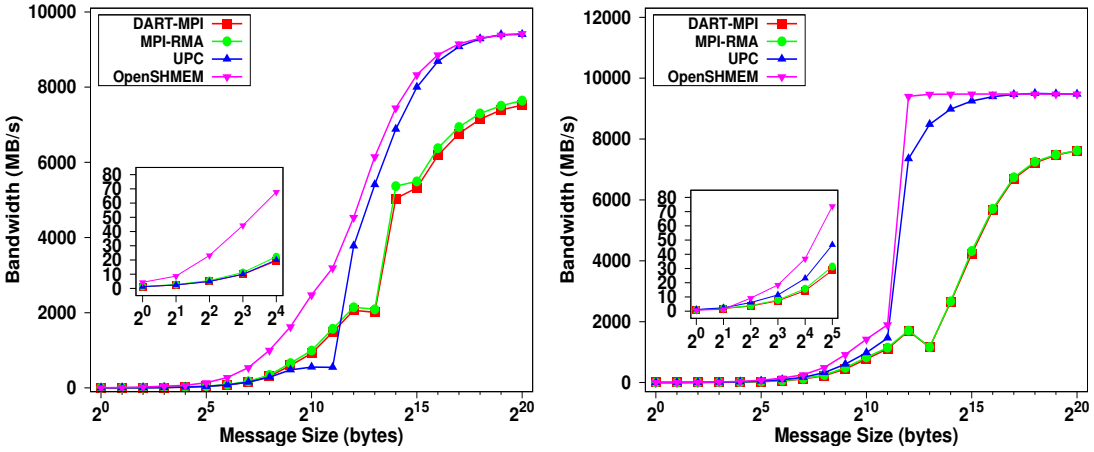
The "flood" test benchmarks multiple non-blocking RMA communication functions. Cray's proprietary *upc_memput_nb* or *upc_memget_nb* and a followed *upc_sync* are used for UPC. For OpenSHMEM, I use the *shmem_putmem_nb* or *shmem_getmem_nb* interface and synchronized by an *shmem_fence* or *shmem_quiet*.

Figures 6.10 and 6.11 show the bandwidth results for various RMA operations. The DART non-blocking operations are implemented on top of MPI RMA operations. Thus,



(A) Non-blocking put (intra-node) (B) Non-blocking get (intra-node)

FIGURE 6.10: Intra-node non-blocking put/get bandwidth on 2 cores.



(A) Non-blocking put (inter-node) (B) Non-blocking get (inter-node)

FIGURE 6.11: Inter-node non-blocking put/get bandwidth on 2 cores.

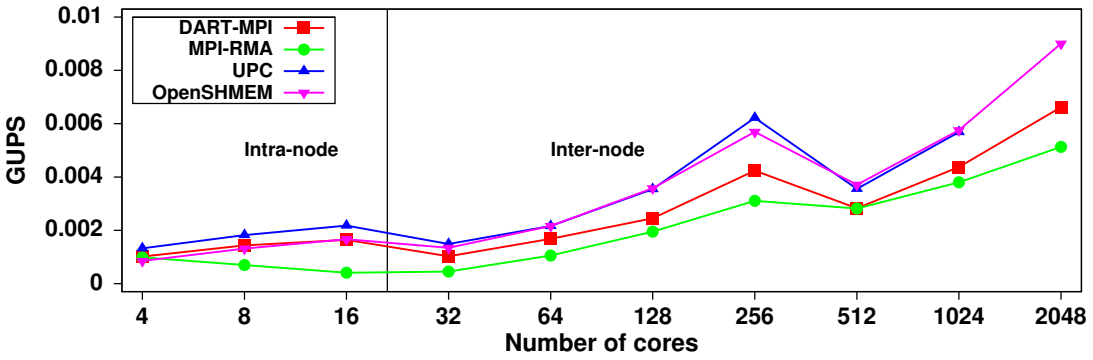


FIGURE 6.12: Random Access performance comparison.

as shown in the Fig. 6.10, there is a bandwidth gap between DART-MPI and MPI RMA due to the implementation overhead of DART non-blocking RMA with respect to the pure MPI RMA. However, such gap becomes negligible in inter-node (refer to 6.11) or long message case (refer to 6.10). The performance of DART-MPI (also MPI) is comparable to UPC and OpenSHMEM in terms of intra-node bandwidth.

The bandwidth performance curves of DART-MPI are almost overlapping with those of MPI in inter-node case, as the Fig. 6.11 shows. However, the bandwidth of DART-MPI and MPI RMA show no advantage over that of UPC and OpenSHMEM, even for large messages. The inter-node message protocol used in Cray MPI switches from eager to rendezvous. This is the direct justification for the sudden drop at message size of around 8 KB.

6.5.3 Random Access (RA)

In this section I present the results through comparing DART and MPI RMA as well as PGAS implementations (e.g., UPC and OpenSHMEM) using Random Access benchmark. It is required to run on 86 nodes with 24 cores in each node.

Random Access (RA) [69] is classified into the HPC Challenge (HPCC) benchmarks developed for the HPCS program. These benchmarks give an insightful understanding of performance of a high-end computing system and indicate how the system will perform across a spectrum of real-world applications [70], such as turbulence simulation, climate simulation or other simulations with societal impact.

Giga Updates Per Second (GUPS) is calculated by identifying the number of memory locations that can be randomly updated in one second, divided by 1 billion. An update indicates a read-modify-write operation to a large distributed table of 64-bit integer values (i.e., 8 bytes on Cray XC40 system).

I evaluate GUPS using the RA code [71] for OpenSHMEM to check the performance of the testbed system concerning the random memory access. For a fair comparison, I implement the DART, MPI and UPC versions of RA by following the same communication pattern. This pattern only involves small data movements (8 bytes) and mainly performs three steps: 1. Read a 64-bit integer; 2. Write a 64-bit integer; 3. Atomically update a 64-bit integer. The above communication coupled with local computations form the kernel that needs to be timed.

Figure 6.12 shows the GUPS performance of the DART, MPI, UPC and OpenSHMEM implementations of RA benchmarks varying the number of processes between 2 and 2048. The UPC benchmark runs terminated with segmentation fault for 2048 processes. Hence, its related result is not shown. Here, I only benchmark the comparison results for power of two numbers of processes. DART-MPI achieves a consistent speedup over MPI, mostly due to the fraction of direct load/store accesses involved in the intra-node data transfers.

DART-MPI performs slightly better than OpenSHMEM when RA runs on a single node (i.e., the number of processes is less than 32). However, we can see that the performance of DART-MPI version suffers at larger number of processes (larger than 16). This is due to the fact that the volume of the inter-node data transfers increases as the growth of the running processes. The inter-node get communication time performance of DART-MPI is poor relative to that of UPC and OpenSHMEM for small messages (e.g., 8 bytes), as obvious from Fig. 6.7(b). Accordingly, a clear gap between the performance of UPC and OpenSHMEM and that of DART-MPI emerges when the application is executed on distributed nodes. In addition, the atomic operation contributes partly to such performance variance between the DART, UPC and OpenSHMEM implementations. Specifically, I use *MPI_Fetch_and_op* with *MPI_SUM* operation and *flush* for MPI-3 and DART-MPI. For UPC and OpenSHMEM, I use *_amo_afadd_upc* and *shmem_long_fadd*, respectively. Noticeably, there are sudden drops occurring at the number of cores of 512 for all versions of RA implementation, shown in the Fig. 6.12. In this test, inter-chassis communications through Cray XC Rank-2 copper network take place when 512 cores are launched, which would greatly degrade GUPS performance, in comparison to the inter-blade (intra-chassis) communications through Cray XC Rank-1 backplane network (refer to App. A). This is the reason for the sudden drops in GUPS when varying the number of cores from 256 to 512.

```
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 6.13: 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 6.14: 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.

6.6 Chapter summary

In this chapter, I have described the data locality-aware design for DART-MPI RMA communication operations based on the nested window structure (see Fig. 5.4). Basically, the contributions of this work are fourfold:

- guarantee the correctness of DART-MPI RMA

- achieve very small DART implementation overhead for the intra-node blocking RMA communication operations
- retain the performance of the native MPI-3 RMA communication operations
- encapsulate the MPI code complexity without violating the semantics of either DART-MPI RMA or MPI RMA

I have evaluated in depth the performance improvement offered by the design using several communication micro-benchmarks and a Random Access benchmark. The results show that the DART intra-node blocking RMA communications are implemented without adding noticeable overhead over the native shared-memory RMA operations supported in MPI-3. Compared with the MPI RMA operations, the DART-MPI RMA can even yield a more than 50% improvement in latency for intra-node blocking operations. Further, the RA evaluation reveals that the DART-MPI RMA is competitive especially when the communication occurs within one node. The DART RMA implementation consistently gains more GUPS than the pure MPI RMA version for up to 2048 processes.

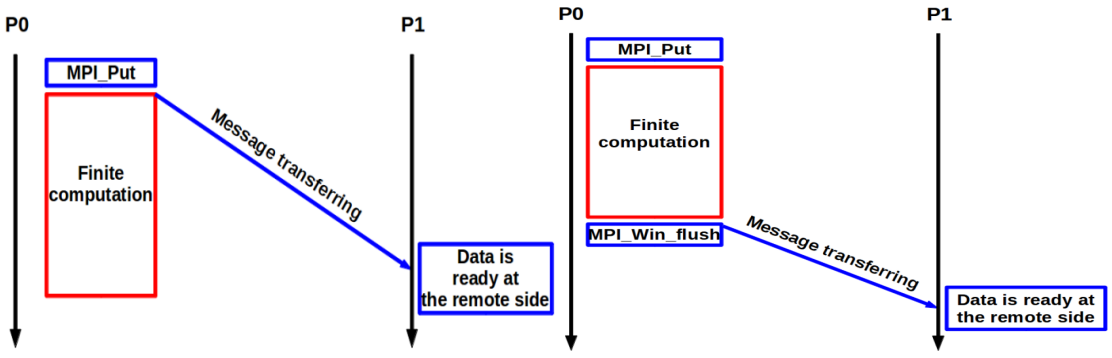
Figures 6.13 and 6.14 show the MPI and DART codes that are used to consciously send a message from the origin to another random target process with the data locality in mind. Specifically, the underlined code lines, highlighted in Fig. 6.14, are required for handling the intra-node data transfers. It is clearly observed that in MPI code, the window creation/destroy operations as well as the access epoch start/end operations should be dealt with explicitly and carefully. In comparison to the MPI code, the DART code is obviously more concise and easier-to-read. This is due to that DART-MPI has internally taken over the responsibility for the locality-awareness by hiding the complexity from the user. Therefore, DART-MPI has reached the goal: make itself easier to programmers for writing efficient applications with the data locality in mind.

Chapter 7

Design of DART-MPI Asynchronous Progression Engine

Towards asynchronous communication operations, MPI standard defines an imprecise progress rule for implementors [72]. However, different interpretations (*strict* or *weak*) of progress rule could lead to differing progress patterns of the non-blocking communication operations (include RMA communication routines).

Figure 7.1, take *MPI_ Put* for example, describes the view of how MPI RMA communication operations implement according to the *strict* and *weak* interpretation, respectively. As Fig. 7.1(a) has shown, once the non-blocking put operation has been posted on the origin process (P0), the data transfer can be enabled independent of the further MPI synchronization calls (e.g., *flush*es) at the P0 side. This pattern supports the truly asynchronous completion of communications (i.e., asynchronous progression) by offering the overlap of communication and computation. On contrary, in Fig. 7.1(b), the put



(A) *Strict* interpretation (B) *Weak* interpretation

FIGURE 7.1: Progress patterns of *MPI_ Put* in terms of *strict* and *weak* interpretation.

communication is delayed until the ensuring *flush* call happens at the origin. This interpretation depends on MPI synchronization call from P0 to make explicit progress. *MPI_ Get* exhibits the similar behavior difference as *MPI_ Put* does under the above two interpretations. The MPI programmers should be aware that MPI RMA communications do not overlap with computation with certainty in all MPI implementations. Clearly, allowing overlap of communication and computation in parallel applications, especially for long messages, is beneficial. This helps to reduce host processor overhead by making the host processor less involved in the transmission or reception of data and to hide latency by letting CPU contribute to the computation in the interim. Therefore, an implementation adheres to the *strict* interpretation has performance advantage potential over one that supports the *weak* interpretation for the applications with some degree of overlap potential [73]. Invoking non-blocking communication operations is an intuitive way to satisfy the overlap potential.

Traditionally, two methods – multi-threading-based and kernel thread-based – are well-studied by scientists to support for communication/computation overlap to minimize the host overhead [74]. In multi-threading-based approach, we spawn threads for each process, and the thread handles all data transfer requests originating from this process asynchronously. It is commonly utilized by many MPI implementations (e.g., MPICH, MVAPICH and Intel MPI). However, keeping thread safety in multi-threading environment is challenging and risky as a result of locking mechanism. In kernel thread-based approach, we partition cores on a node between application threads, OS kernel threads and system service daemon threads (e.g., Cray MPI [75]). At least one core per node is reserved as the MPI kernel thread for handling the interrupts delivered by NIC (Aries NIC for Cray XC40 system) and making progress on the outstanding communications. Noticeably, each interrupt triggers a context switch and thus will significantly degrade performance if it frequently occurs. Additionally, an innovative approach – process-based [11, 74] is devised to designate random number of cores per node to be the processes handling the communication progression. It has been proved to be a promising alternative method supporting asynchronous communications for current multi-/many-core architecture, compared with the conventional methods.

Given the performance properties, PGAS languages require of the RMA interfaces to hide network latencies by overlapping communication and computation. Therefore, supporting the asynchronous progression in DART RMA is inevitable.

In this chapter, first I design the DART-MPI asynchronous progression engine (supports C interfaces) with the process-based approach so that make DART-MPIs RMA truly support asynchronous communication to some degree. I encapsulate the asynchronous

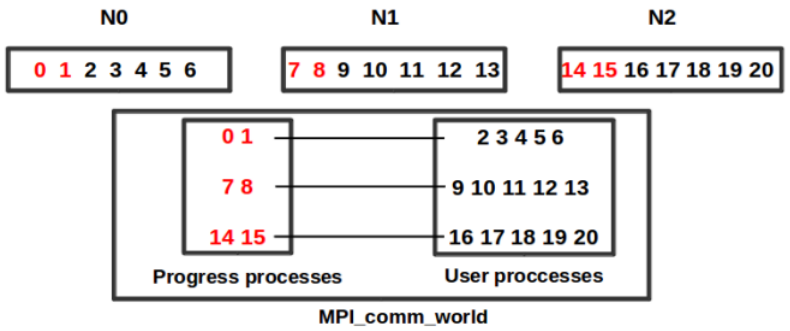


FIGURE 7.2: DART process layout with asynchronous progression enabled. The two progress processes are marked with read ink and serve the RMA requests originating from other on-node progresses. A node signifies a shared memory domain.

progression code inside all DART RMA-related interfaces. Therefore, the DART applications require no code change and can transparently achieve overlap of communication and computation without extra code complexity. Importantly, the asynchronous progression approach is designed to achieve low implementation overhead (i.e., minimize the implementation overhead brought by the asynchronous progression code.) on the basis of ensuring correctness. Second, I theoretically analyze the efficiency and suitability of applying the DART asynchronous non-blocking RMA communication operations on parallel applications. Finally, I evaluate the performance of DART non-blocking RMA communication operations with micro-benchmarks and a five-point stencil kernel benchmark. For the application kernel benchmark, not only I compare the DART-MPI RMA and the native MPI, but I also compare it with two other typical PGAS implementations – UPC and OpenSHMEM.

7.1 Designing the DART-MPI progress engine

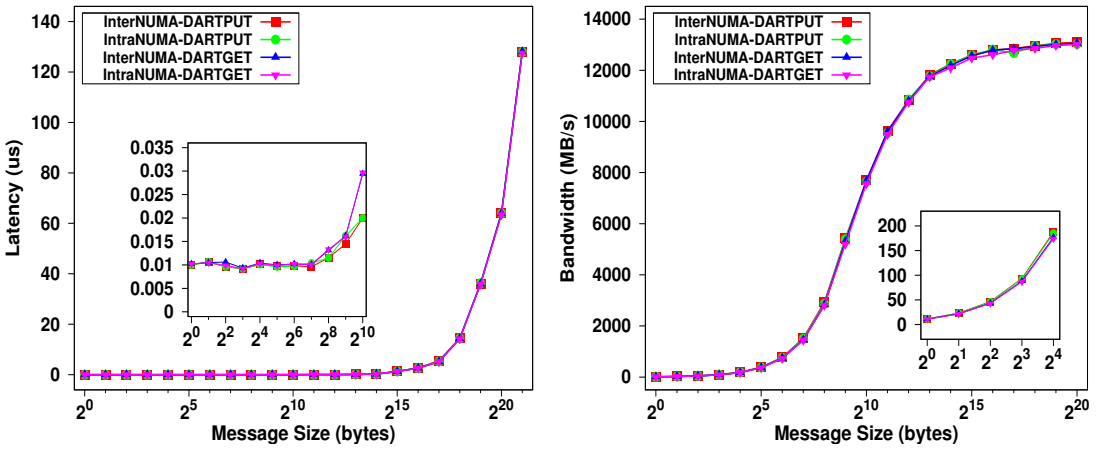
In this section, I outline the design details which address the process-based implementation issues regarding the asynchronous progression engine in DART. Foremost, I partition cores on a node between progress processes and user application processes. Next, I introduce the way of setting up global memory across the partitioned cores. Finally, I internally launch the asynchronous progression engine by intercepting and overriding the DART RMA functions. That is, the progress processes are timely notified in terms of the operation characteristics and then react accordingly.

7.1.1 Core partitioning

On the premise that the DART global memory architecture is applied, (see Fig. 5.4) processes within the same shared memory domain are possible to communicate via direct

load/store accesses. Detailedly, the user memory regions are directly mapped into the on-node progress processes' memory address spaces. This property of DART system positively affects the appliance of the process-based approach in a way of reducing the time complexity. Consequently, with the mapped memory address, RMA operations can be redirected and handled by the progress processes without the verbose message transfers between the application processes and the progress processes within the same shared-memory domain.

At DART initialization time (i.e., in *dart_ init*), one launches applications with a number of processes. Several processes within each node are then designated to be the progress processes for the other processes located in the same node. Figure 7.2 shows a specific instance where I use two progress processes within each node and each of them is pinned to a core. Actually this asynchronous progression engine allows one to generate random number of progress processes internally on-demand. Importantly, after excluding the progress processes, the remaining processes comprise the *DART_ TEAM_ ALL* that are only visible to the user applications. Apparently, *DART_ TEAM_ ALL* here denotes a subset of *MPI_ COMM_ WORLD* and thus the progress processes are only visible to the DART system instead of the users. Consequently, the progress processes are prohibited to be involved in all DART interfaces except the *dart_ init* or *dart_ exit*. As the Fig. 7.2 shows, the DART progress processes are first deployed appropriately in *dart_ init*. Those progress processes then skip all intermediate DART interfaces in any application until they meet the final call to *dart_ exit*. Specifically, inside the *dart_ exit*, the progress processes are busy waiting for the commands delivered by other on-node user processes in an *MPI_ Iprobe* loop. The asynchronous progression engine ensures that the progress processes react properly according to the command characteristics, which I will discuss



(A) Blocking RMA latency (B) Nonblocking RMA bandwidth

FIGURE 7.3: DART RMA performance comparison between intraNUMA and inter-NUMA.

in the sections below.

Regarding the creation of sub-teams, the engine needs to keep the progress processes in the parent team be a part of any of its sub-teams. This is done internally inside DART system. In other words, the progress processes (that created at the initialization time) are incorporated in each team to enable the asynchronous progression for the data transfers within this team. Noticeably, the progress processes are reserved to serve the asynchronous progression in their lifetime.

It can be observed that the two on-node progress processes are ranked contiguously by revisiting the Fig. 7.2. Therefore, how far is the physical distance between the two progress processes depends on the methodology of task placement on cores (e.g., cyclic, blocked and so on). Hence, they are distributed either the same NUMA node or across NUMA nodes. However, DART-MPI asynchronous progression engine disregards the location of on-node progress processes compared to other application processes. That is, an application process is not only bound to the progress processes within the same NUMA domain and thus able to request all on-node progress processes. Take Cary XC40 for example, the communication performance between across NUMA nodes and within NUMA node make no big difference in terms of latency and bandwidth, as obvious in Figure 7.3 (using the same benchmark with Sect. 6.5.2). Therefore, it is may be not worth hooking each application process on to the physically closest progress process at the expense of exacerbating the code complexity.

To load balance the task distributed across multiple progress processes within each node to some degree, I let each progress process turn to one of the on-node progress processes in a cyclic pattern. This can prevent the application process from simply sticking to a progress process. Additionally, the progress processes are designated to on-node application processes at DART initialization time as evenly as possible.

7.1.2 Global memory setup

The progress processes are natively designed to be invisible to the target applications. However, I here let progress processes being aware of any global memory allocation since the progress processes, after all, would be a proxy for the application processes to asynchronously perform the RMAs to the target global memory segments.

The global memory region could be allocated by invoking *dart_ memalloc* (i.e., non-collective global memory allocation) or allocated via *dart_ team_ memalloc_ aligned* (i.e., collective global memory allocation). The memory region reserved for non-collective global memory allocation is statically built up at DART initialization time. Meantime, the progress processes are intuitively involved in the collective window creation operation.

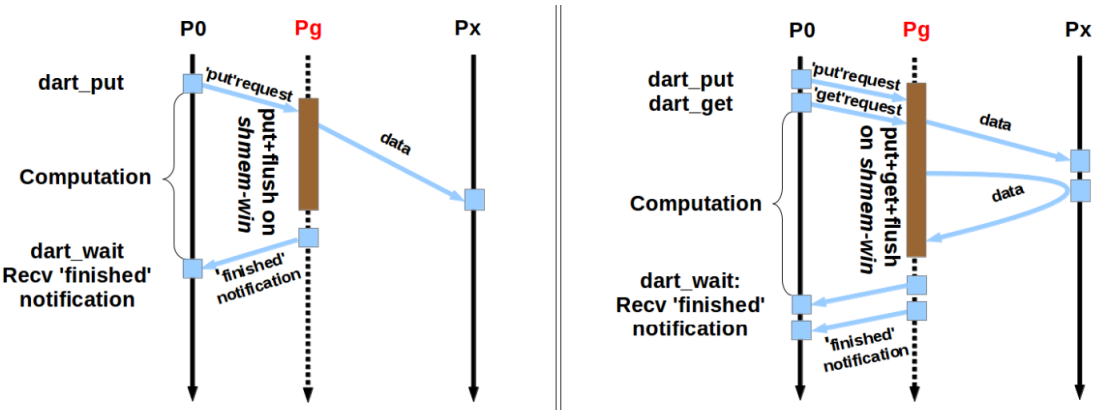


FIGURE 7.4: DART intra-node RMA non-blocking communication protocols using process-based approach. P0 is the origin which initiates the RMA operations, Px is the destination. P0 and Px are in the same node, Pg (marked with red ink) denotes the progress process corresponding to P0. Left: Pg consumes one RMA request. Right: Pg consumes two contiguous RMA requests.

Next, I allocate the collective global memory region dynamically by collectively invoking *dart_team_memalloc_aligned* on certain team. This interface is originally visible to application processes instead of progress processes. Therefore, one of the participated application processes internally notifies all on-node progress processes. After consuming the intra-node notifications, the progress processes are involved in the collective global memory allocation with the received parameter (i.e., *index*). In this case, a notifying operation is equivalent to a small intra-node message transferring. It is simply performed by using pair-wise *MPI_Send* and *MPI_Recv*, which will not cost a lot. Clearly, the progress processes are internally involved in all global memory allocations (window creation) within DART system. To minimize the space complexity, I allocate memory of 0 byte in the progress processes' memory space as a result of memory mapping. This is feasible since the progress processes' memory space would never be accessed by users. Therefore, there is no extra memory allocation wasting from the application point of view.

Importantly, the target global memory segments should be accessible to the progress processes after the window is generated. Therefore, the progress processes must immediately start a global lock epoch to all processes related to this window and end it when the window is destroyed. Note that, there is no other simultaneous epoch on the same window in the interim.

7.1.3 Asynchronous non-blocking RMA communication operations

DART non-blocking RMA operations can be activated successfully as the processes are granted access to the window within the above global lock epoch. Figure 7.4 and 7.5

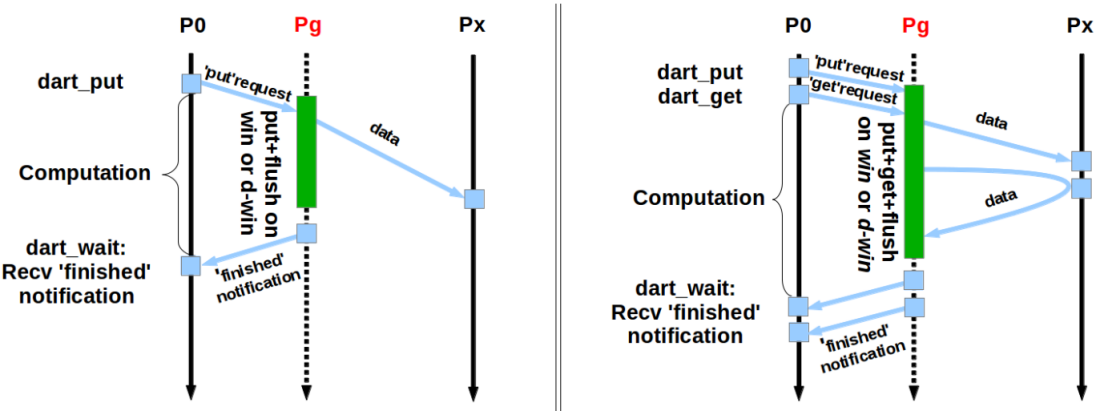


FIGURE 7.5: DART inter-node RMA non-blocking communication protocols using process-based approach. P0 is the origin which initiates the RMA operations, Px is the destination. P0 and Px are in different nodes, Pg (marked with red ink) denotes the progress process corresponding to P0. Left: Pg consumes one RMA request. Right: Pg consumes two contiguous RMA requests.

give insights into the asynchronous progression design of the DART non-blocking RMA communication operations in intra-node and inter-node case. In detail, first of all, the origin chooses an on-node progress process. Instead of actually performing the RMA operations, it sends the RMA (i.e., put or get) requests to its current progress process. This progress process then takes over the actual message transfers meant for the given destination Px on *sharedmem-win* or RMA window (i.e., *win* for non-collective global memory and *d-win* for collective global memory) according to whether the target process is an on-node application process or not. In this sense, the origin can do its own computation tasks independently from the initiated RMA communications. It is clearly observed that the asynchronous progression engine also keeps the data locality in mind in order to achieve high performance. The *dart_ wait* operation at the origin is simply busy-waiting for the finished notification from its progress process.

Figures 7.4 and 7.5 show two common scenarios where the RMA requests are consumed by progress process brokenly (shown in left) and consecutively (shown in right), respectively. The progress process would like to *flush* the target immediately if it finds no RMA requests followed closely. Otherwise, if the RMA requests are consecutively sent to the progress process, ideally the progress process will build a backlog of RMA requests. They are processed in batch by progress process when it receives a 'wait' request or becomes idle (i.e., probe nothing). In this case, the data transfers fail to start as soon as the RMA calls occur. However,the progression in the accumulative RMA communications can be activated automatically when the progress process probes no message, which provides overlap of communication and computation to some extent. Furthermore, this behavior can somehow bring performance improvement through amortizing a *flush* synchronization call with multiple RMA operations. I introduce a specific request with characteristics

TABLE 7.1: Packet structure. The communication informations are contained in the packet to enable an RMA communication for the progress process.

Field	**Value**
dest	The target process ID
index	Denote the involved team
origin_ offset	The offset relative to the beginning of origin memory segment
target_ offset	The displacement relative to the beginning of target memory segment
data_ size	The transferred data size
segid	0: Indicate the non-collective global memory $\geq$ 1: Indicate the specific collective global memory
is_ shmem	0: origin and target are in different nodes 1: origin and target are in the same node

of 'wait' for ensuring that the desired RMA communications get started immediately (without unreasonable delay) by the progress process when *dart_ wait* is invoked. I.e., the origin sends a 'wait' request to its progress process when invoking the *dart_ wait*.

The application processes, rather than the progress processes, are originally aware of the raw information on the DART RMA communications. Therefore, an application process should encapsulate the raw information first and then send it as a packet to its corresponding progress process. Essentially, the packet is determined in a way of abstracting the input data provided by DART RMA operations. The input data basically consists of three attributes, that is, global pointer, message size, and origin address, where the global pointer exposes the target unit/process ID, *segid*, target address offset and *index*. The message size, *segid* and *index* should be straightforwardly included into the packet. Besides, the progress process should perform RMA communications with locality-awareness in mind. Therefore, a signal value indicating that data transfer occurs within node or across nodes is entailed in the packet. The *segid*, *index* and the signal value can make progress process easily get the *sharedmem_ win* or *win* or *d-win* if possible. The target process ID in the global pointer signifies the absolute process ID with respect to *DART_ TEAM_ ALL*. However, the progress process is out of the scope of *DART_ TEAM_ ALL*. Therefore, the target process ID needs to be translated into the correct ID which is understood by the progress process and then the correct ID is incorporated into the packet. I include the origin address offset relative to the beginning of the origin's memory segment into the packet. With the origin address offset, the progress process can calculate the raw origin address after obtaining a local pointer pointing to the beginning of the origin's memory segment. In the case of touching the attached memory (i.e., spanned by d-win), getting initial displacement relative to the beginning of the remote memory segment is essential. Hence, the correct displacement

```
1: Input:global pointer p, message size m, target process r
2: function dart_put(p, m, r)
3:     g ← TranslatetoPG {Obtain its on-node progress process}
4:     packet ← EncodeComm(p, m, r)
5:     MPI_Send(packet ...g)
6:     handle ← MPI_Irecv(NULL ...g) {Indicates control message}
7:     return handle
8:     end function
9: function dart_get(p, m, r)
10:    g ← TranslatetoPG;
11:    packet ← EncodeComm(p, m, r)
12:    MPI_Send(packet ...g)
13:    handle ← MPI_Irecv(NULL ...g)
14:    return handle
15:    end function
16: procedure dart_wait(handle)
17:    MPI_Send(NULL ...g);
18:    MPI_Wait(handle)
19:    end procedure
```

FIGURE 7.6: Pseudo-code for application processes.

```
1: procedure progress()
2:     while running do
3:         header ← Iprobe()
4:         if header.flag then
5:             switch (header.messageType)
6:             case PUT:
7:                 MPI_Recv(packet, ...header.o) {header.o denotes the origin process}
8:                 buf ← DecodeObuf(packet.origin_offset)
9:                 r ← packet.dest
10:                target_offset ← packet.target_offset
11:                win ← DecodeWin(packet.is_shmem, packet.segid, packet.index)
12:                MPI_Put(buf, r, target_offset, ...win)
13:                request.r ← r
14:                request.win ← win
15:                request.o ← header.o
16:                q ← AddQueue(request)
17:                break
18:                end case
19:            case GET:
20:                MPI_Recv(packet, ...header.o)
21:                buf ← DecodeObuf(packet.origin_offset)
22:                r ← packet.dest
23:                target_offset ← packet.target_offset
24:                win ← DecodeWin(packet.is_shmem, packet.segid, packet.index)
25:                MPI_Get(buf, r, target_offset, ...win)
26:                request.r ← r
27:                request.win ← win
28:                request.o ← header.o
29:                q ← AddQueue(request)
30:                break
31:                end case
32:            case WAIT:
33:                MPI_Recv(NULL, ...header.o)
34:                while q is not an empty queue do
35:                    request ← DeQueue
36:                    MPI_Flush(request.r, request.win)
37:                    MPI_Send(NULL, request.o)
38:                end while
39:                end case
40:            end switch
41:        else
42:            while q is not an empty queue do
43:                request ← DeQueue
44:                MPI_Flush(request.r, request.win)
45:                MPI_Send(NULL, request.o)
46:            end while
47:        end if
48:    end while
49:    end procedure
```

FIGURE 7.7: Pseudo-code for progress processes.

that is passed to the RMA operation is the sum of the initial displacement and the raw target offset. It should also be included in the packet. Accordingly, the packet is represented as the data structure shown in Table 7.1. This packet is identified by MPI as a derived datatype through a call to *MPI_ Type_ create_ struct*.

To demonstrate the interactive activities between application processes and the corresponding progress processes, Figures 7.6 and 7.7 show the pseudo-code executed by application processes and progress processes, respectively. Fig. 7.6 describes the algorithm for each of the DART put, get and wait primitives straightforwardly. Obviously, the progress process procedure shown in Fig. 7.7 is invoked as necessary to make progress on outstanding put and get communications. Here, I only describe the procedures or

functions that are only directly related to RMA communications and leave out the procedures handling the collective memory allocation/deallocation, team creation/destroy operations and termination routine (i.e., *dart_ exit*).

One point regarding the asynchronous progression engine needs to be optimized when the performance and practicality are considered. Smaller message transfers in MPI can avoid the network latency hiding issue by using the eager protocol (i.e., through buffering small messages) [76]. Therefore, the eager protocol for RMA communications potentially involves overlap of computation and communication. However, the rendezvous protocol for long message transfers does not allow any overlap between computation and communication. Obviously, the performance for long messages could get substantial benefit from the asynchronous progression. Therefore, I need to weigh the bandwidth performance and overlap achieved by DART non-blocking RMA operations with the asynchronous progression enabled, especially for small messages. This will be explained in Section 7.2.1.

7.1.4 Suitability analysis

It is important to present the suitability of this DART-MPI asynchronous progression approach for the real-world applications. As mentioned in Chapters 3 and 6, I use the passive target mode for DART RMA communications. However, MPI implementations do not guarantee truly passive RMA operations in terms of the definition of MPI standards. This means that the remote target might be involved in an RMA operation only by explicitly making MPI calls (any MPI call) to ensure communication progression on its side. Actually, the study [74] mentioned before adopted the process-based approach to realize the truly passive RMA (i.e., the RMA can proceed timely even when the target is stuck with a complex computational task). Besides, another study [77] tries to implement the truly passive RMA by using InfiniBand Atomics. Achieving truly passive RMA operations is critical for the irregular parallel algorithms.

Revising the Fig. 7.1(b), we can observe that the RMA communications are essentially initiated by letting the origin explicitly issue the synchronization calls even when the target process is ready. In this case, the actual data transfer will be deferred and fails to overlap with the following computation after employing the above two studies for truly passive RMA. Detailedly, the performance of applications including the traditional regular parallel algorithms (e.g., stencil computation, FFT and so on) may benefit from this DART asynchronous progression approach rather than the above two studies. In addition, the regular parallel algorithms feature balanced communication patterns and still play an active role in the engineering and scientific communities.

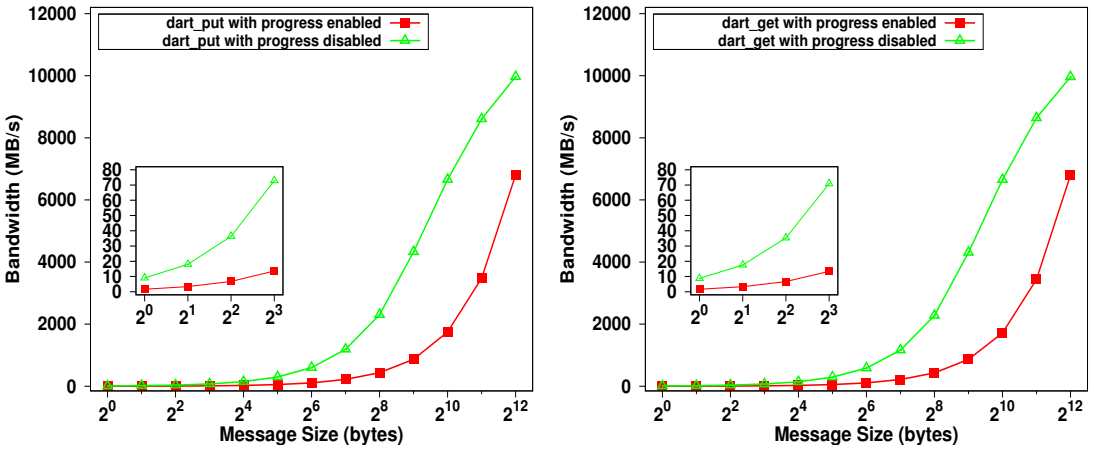
7.2 Performance evaluation

In this section, I present a comprehensive experimental evaluation and analysis of the asynchronous progress feature discussed in the preceding sections by running a set of benchmarks on Cray XC40 system. The first step in identifying the performance of DART-MPI progress engine is to examine the impact on performance of non-blocking RMA communications, in terms of micro-benchmark characteristic – bandwidth performance, when the asynchronous progression is enabled. Meanwhile, I show the effect of the optimization scheme introduced in Sect. 7.1.3. Following that, I evaluate the ability to improve the application performance through overlapping with computation for the non-blocking RMA communication operations provided by DART-MPI progress engine (using the optimization scheme). I should add that, the asynchronous progression feature is disabled in Cray MPI by default and thus should be turn on explicitly by setting environment variables at compiler time. (Surprisingly, after setting environment variables as needed, it seems that the MPI asynchronous progression does work for the MPI non-blocking point-to-point communications rather than RAM communications.) For a more comparative study, a 2D 5-point stencil kernel benchmark is run and analyzed on DART, Cray MPI, UPC and OpenSHMEM implementations to investigate the synthetic impact of progression on the overlap of computation and communication.

The following tests are all performed with non-blocking RMA communications only. I evaluate DART non-blocking RMA operations with two progress processes per node. Reserving two progress processes is sufficient to the testing of micro-benchmarks as there are a small number of processes communicating. On the other hand, employing two progress processes is enough for the testing of application benchmark since a good number of computing processes (i.e., application processes) need to be guaranteed to share the required workload.

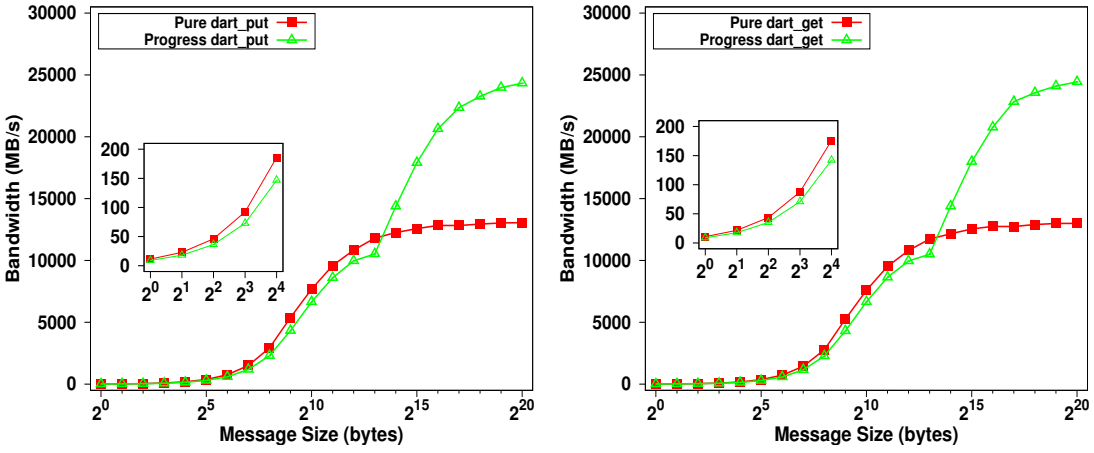
7.2.1 Performance evaluation for the asynchronous progression implementation

Essentially, the performance of RMA communications needs to be retained in pursuit of the highest overlap. I.e., the adverse impact of the asynchronous progression implementation on the communication performance (i.e., bandwidth) of non-blocking RMA operations, especially for small messages, should be minimized. In this section, I first target the effect of the optimization effort by performing the DART non-blocking RMA operations. By experiments, I set the message threshold for enabling the asynchronous progression as 4 KB (this value strongly depends on the underlying MPI message passing protocol). Figure 7.8 specifically target the intra-node RMA communication with respect



(A) Intra-put (B) Intra-get

FIGURE 7.8: Bandwidth performance of DART non-blocking intra-node RMA operations with and without progression for small messages.

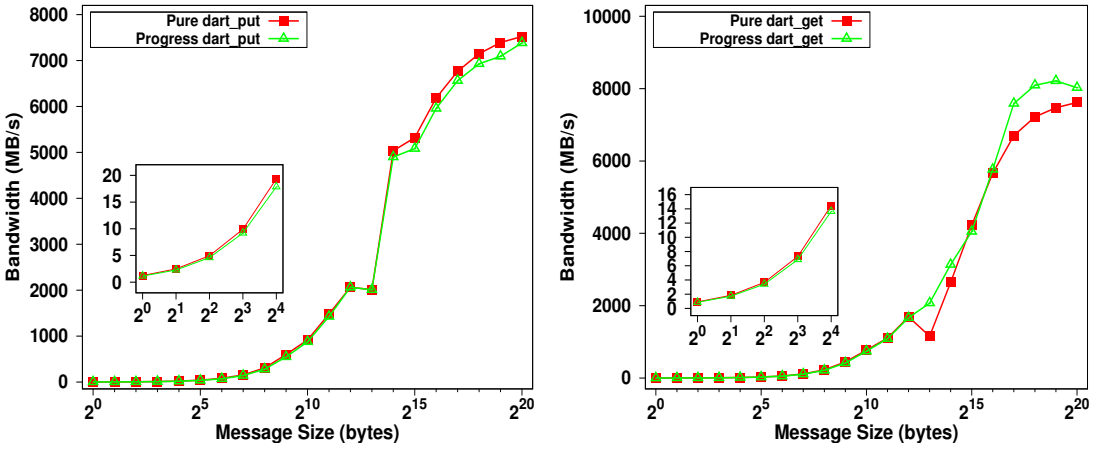


(A) Intra-put (B) Intra-get

FIGURE 7.9: Bandwidth performance comparison between pure and progress DART non-blocking intra-node RMA operations.

to small messages (not larger than 4 KB) and indicates that two bandwidth curves have an obvious and consistent gap. Therefore, the bandwidth of RMA communications for small messages could be reduced by half if the progression is enabled in this case. The increase of overlap comes at the expense of significantly degrading performance for small messages. Therefore, enabling progression for small messages does not pay off eventually.

Next, Figures 7.9 and 7.10 portray the overall bandwidth results of the non-blocking RMA communications before (pure DART-MPI RMA) and after (progress DART-MPI RMA) featuring the asynchronous progression, in the intra-node and inter-node case, respectively. Clearly, the comparison curves of the intra-node put show the similar trend as that of the intra-node get. At message sizes smaller than 16 KB, we can see



(A) Inter-put (B) Inter-get

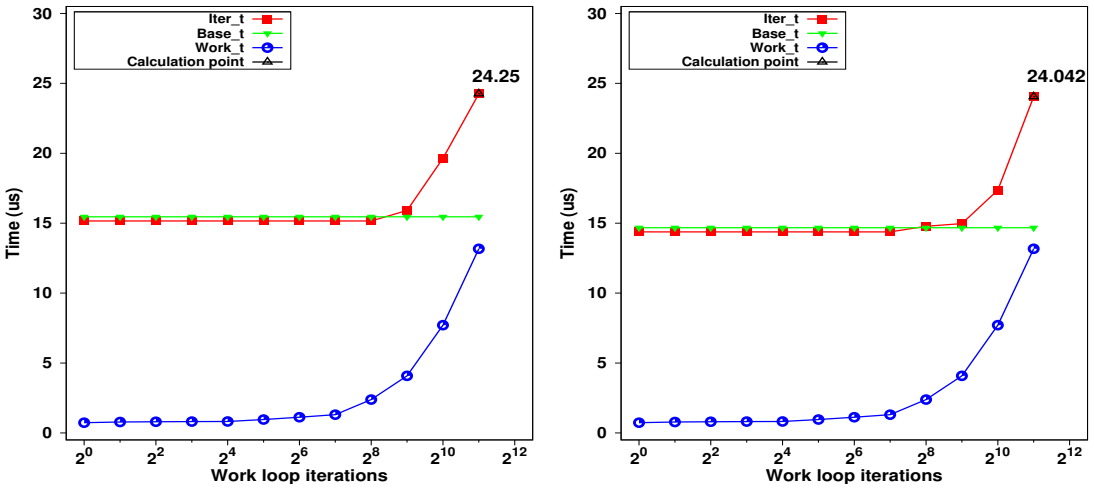
FIGURE 7.10: Bandwidth performance comparison between pure and progress DART non-blocking inter-node RMA operations.

$\sim$ 20% reduction on average for put in bandwidth and $\sim$ 19% for get after featuring asynchronous progression. Noticeably, the curves for progress DART-MPI RMA goes up suddenly and sit far above the curves for pure DART-MPI RMA after it crosses over the message with size of 16 KB. Such performance behavior occurs since the progress processes speed up the message processing progression. On the other hand, in the inter-node case, the progress inter-node put shows a range of bandwidth reduction from 0.7% to 10%, compared with the pure inter-node put. As the Fig. 7.10(b) shows, the progress inter-node get shows up to 7% reduction in bandwidth at the message sizes equal to or smaller than 4 KB. In this case, the overhead imposed by progression is negligible. Once the message size is increased to 8 KB, progress DART-MPI inter-node get shows clear bandwidth improvement and different behavior than pure DART-MPI get as a result of the asynchronous progression.

7.2.2 Host processor *overhead* and application *availability*

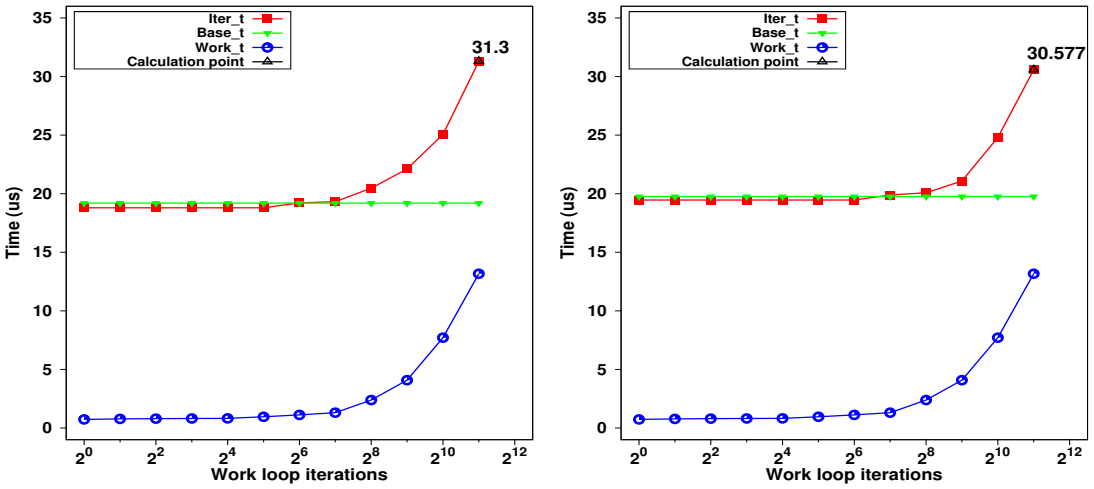
In this section, I examine the overlap degree of communication and computation when using DART non-blocking RMA operations with asynchronous progression enabled, based on the measurements modified from Sandia MPI Micro-Benchmark Suite (SMB) [78]. Meantime, I check the potential of DART non-blocking RMA communication operations to improve the application performance through overlapping with computation, compared with the Cray-MPI RMA (without asynchronous progression).

Specifically, I use a test component which is contained in the SMB. In this component, two basic metrics – the host processor *overhead* and application *availability* [79] are measured for non-blocking MPI send and receive operations. To target the non-blocking



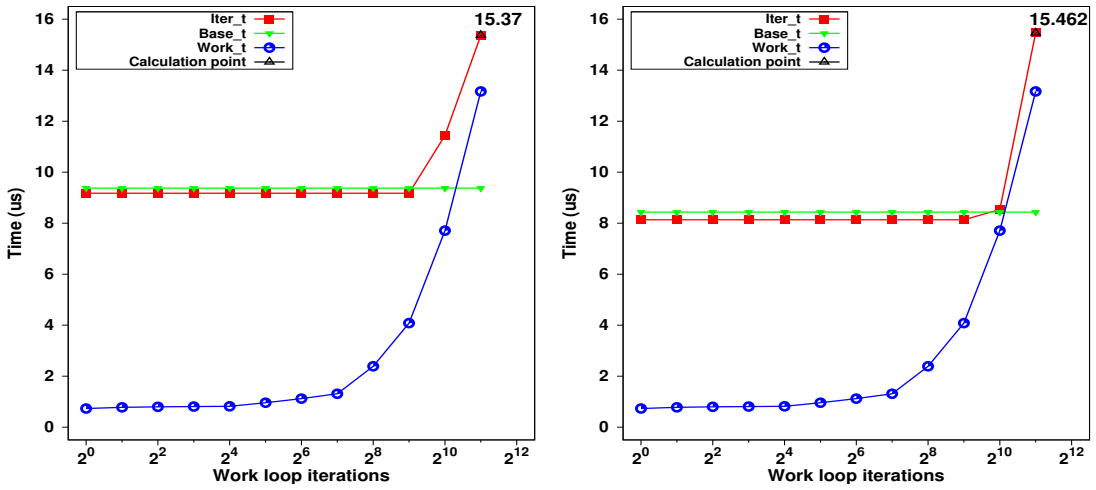
(A) Intra-node non-blocking put (B) Intra-node non-blocking get

FIGURE 7.11: The *overhead* and application *availability* achieved with MPI intra-node RMA operations for 16 KB message sizes. The measured *overhead* is significant and the application *availability* is 28.3% for put and 25.9% for get. Without asynchronous progression.



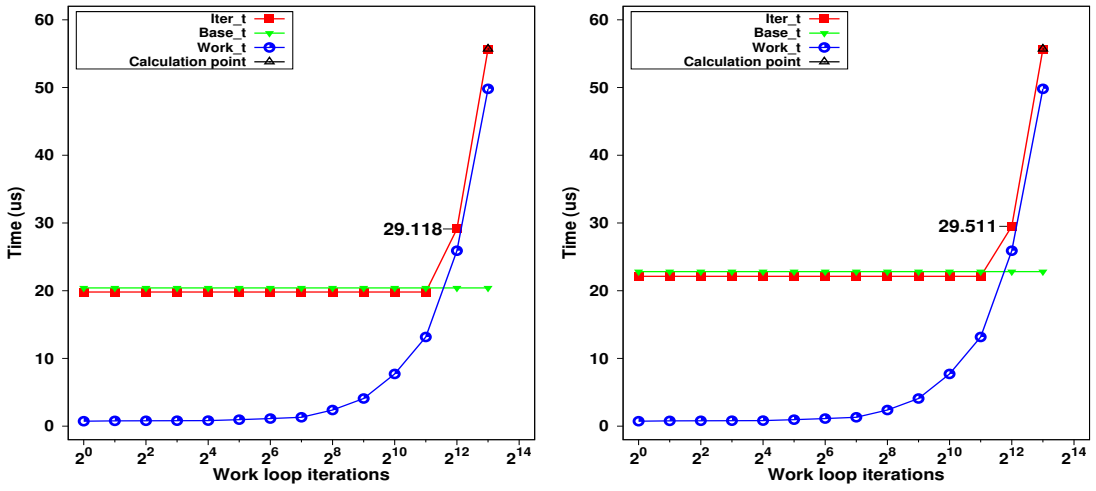
(A) Inter-node non-blocking put (B) Inter-node non-blocking get

FIGURE 7.12: The *overhead* and application *availability* achieved with MPI inter-node RMA operations for 16 KB message sizes. The measured *overhead* is significant and the application *availability* is 10.6% for put and 11.9% for get. Without asynchronous progression.



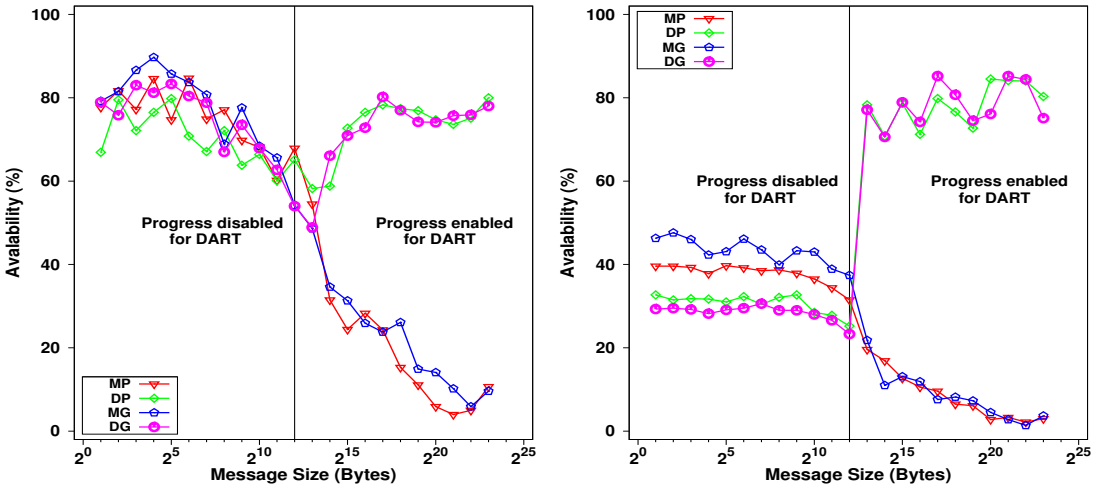
(A) Intra-node non-blocking put (B) Intra-node non-blocking get

FIGURE 7.13: The *overhead* and application *availability* achieved with DART non-blocking intra-node RMA operations for 16 KB message sizes. The measured *overhead* is low and the application *availability* is 76.5% for put and 72.8% for get. With asynchronous progression.



(A) Inter-node non-blocking put (B) Inter-node non-blocking get

FIGURE 7.14: The *overhead* and application *availability* achieved with DART non-blocking inter-node RMA operations for 16 KB message sizes. The measured *overhead* is low and the application *availability* is 71.2% for put and 74.2% for get. With asynchronous progression.



(A) Intra-node non-blocking RMA (B) Inter-node non-blocking RMA

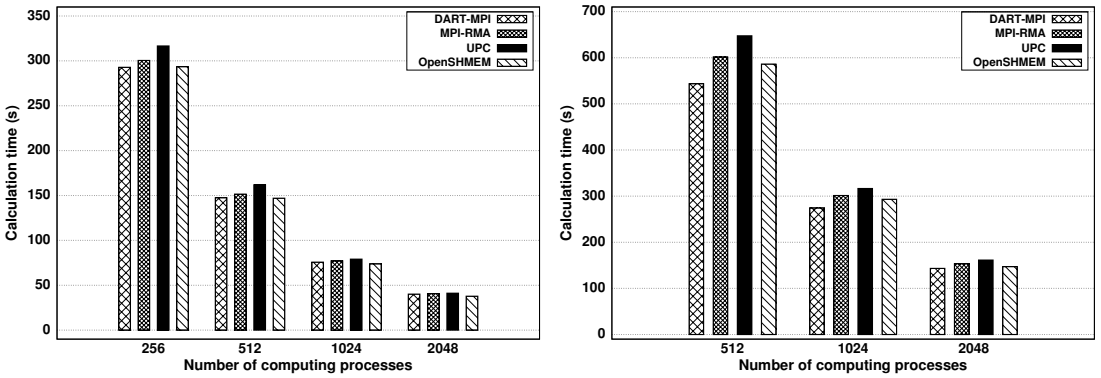
FIGURE 7.15: A comparison of application *availability* between DART non-blocking and MPI RMA operations. "M" denotes MPI and "D" denotes DART. "P" denotes Put operation and "G" denotes Get operation.

RMA operations, I adjust this measurement by replacing the send or receive routines with the non-blocking RMA routines. The definition to the *overhead* and application *availability* nevertheless stays unchanged. I.e., the *overhead* is defined as the amount of time that a process is engaged in the transfer of each message and thus cannot perform other operations in the interim. On the other hand, application *availability* is defined to be the fraction of total transfer time that the application is free to perform non-MPI related work [78].

The test measures the time to a loop which performs a non-blocking RMA operation of a given size, some works and then waits for the data transfer to complete by using a synchronization call (e.g., *dart_wait* for DART-MPI, *MPI_Win_flush* for MPI) in each iteration. The loop stops until the total amount of time ($iter\_t$) hits more than 1.5 times the data transfer time ($base\_t$) as the work performed ($work\_t$) is increased for each iteration. Note that, the *overhead* and application *availability* are calculated at the loop stop point with the method of $overhead = iter\_t - work\_t$ and $availability = 1 - overhead/base\_t$.

Taking message size of 16 KB for example, Figures 7.11 and 7.12 illustrate the *overhead* and application *availability* when using MPI RMA communication operations, in intra-node and inter-node case. The *overhead* and application *availability* implied in these figures are baseline values without featuring the asynchronous progression.

Likewise, the *overhead* and application *availability* achieved with DART non-blocking RMA operations at message size of 16 KB in the intra-node and inter-node cases are shown in Figures 7.13 and 7.14. We can clearly see that DART non-blocking RMA



(A) 32768×32768 (B) 65536×65536

FIGURE 7.16: Five-point stencil kernel performance comparison.

communication operations support smaller *overhead* and higher application *availability* in all cases, compared to the baseline values. Obviously, at the work loop iterations of 2048, DART non-blocking RMA operations deliver less total time than MPI RMA operations in the case of intra-node by partly hiding the data transfer time. Besides, DART non-blocking RMA operations can even show less total time at 4096 than MPI RMA operations at 2048 for inter-node data transfers. Accordingly, enabling the asynchronous progression potentially speeds up the execution time of an application.

Figure 7.15 comparatively reports the application *availability* achieved with the DART non-blocking and MPI RMA operations as a function of the message size by repeating the above measurement. DART non-blocking RMA operations support slightly lower application *availability* when the transferred message sizes are not larger than 4 KB, which is in my expectation since the asynchronous progression feature is disabled in this case. Nevertheless, it is still possible for them to be overlapped with computation as the eager protocol is employed. Clearly, DART non-blocking RMA can get consistently better application *availability* than MPI RMA once the message sizes exceed 4 KB. Specifically, MPI RMA operations show very poor application *availability* for message sizes larger than 1 MB.

7.2.3 Five-point stencil kernel benchmark

In this section, I calculate a 2D heat equation by applying a five-point stencil on a square grid based on a 2D heat solver (provided in a research paper [15]). This equation is solved in an iterative way with the Gauss-Seidel method. The N×N cell grid is decomposed evenly by rows among P distributed processes. I.e., this algorithm is based on the row-wise block-striped data decomposition scheme. Each cell of the grid is a 4-byte floating point number. Extra halo zones are used to exchange boundary elements between

neighbors. A total of $4 \times N \times (2 \times P - 2)$ bytes of data per iteration are transmitted using non-blocking put operations. With those halo data, all the inner grid cells can get updated successfully. I run the stencil kernel until convergence of solution. The time recorded in the benchmark includes the execution time of the Gauss-Seidel solver (local computation part) and the non-overlapping portion of communication time for halo exchange.

I run this benchmark for DART, native MPI, UPC and OpenSHMEM implementations on the 32768×32768 and 65536×65536 cell grids, respectively. Note, that I use the same number of computing processes in all implementations and dedicate two extra progress processes to the DART-MPI asynchronous progression. We can refer to Sect. 6.5.2.2 as for the RMA communication and synchronization routines that are used in the UPC and OpenSHMEM implementations. However, the run time variability is not measured by benchmarking the application multiple times, which would be expensive.

As Figure 7.16(a) shows, the comparison results of a 32768×32768 cell grid processes revolve around the strong scaling performance with the increased number of computing processes (ranging from 256 to 2048). As expected, the DART implementation consistently performs better than the MPI implementation since DART non-blocking put operation is allowed to be overlapped with computation when proceeds. Detailedly, the DART implementation provides a speedup of 1.025x, 1.027x, 1.019x and 1.014x on 256, 512, 1024 and 2048 computing processes respectively. We can clearly observe that the calculation time performance speedup is reduced as the number of computing processes become larger (larger than 512). The workloads allocated on each processing process are lessened as the larger number of computing processes is employed. However, the transferred message size implied by a round of halo exchange stays unchanged. Therefore, the communication overhead may contribute more to the overall calculation time than the local computation overhead does. I.e., a large portion of the network latency may fail to be hidden. This will in turn weaken the positive effect of the asynchronous progression on the overall performance and then leads to a decline in the speedup. In addition, the DART implementation shows better performance than UPC and to a smaller extent than OpenSHMEM until the number of processing processes is beyond 1024.

To target larger workload, Figure 7.16(b) shows comparison results of a 65536×65536 cell grid distributed across 512, 1024 and 2048 computing processes. It is obvious that DART-MPI RMA shows superior strong scaling performance than that of MPI, UPC and OpenSHMEM. Specifically, the DART implementation provides a speedup of 1.11x, 1.09x and 1.07x over MPI implementation on 512, 1024 and 2048 computing processes. Such performance advantage degrades slightly when the number of computing processes grows,

as the case in Fig. 7.16(a). The above two figures suggest that the asynchronous progression can speed up the application performance. However, the performance of DART-MPI version suffers in the scenario where the communication time plays a dominant role in the calculation time. In overall, the calculation time shown in all implementations is decreasing as the number of the computing processes is increased.

7.3 Chapter summary

In this chapter, I have illustrated the design of enabling the asynchronous communications of DART non-blocking RMA for large message sizes by using the progress process-based approach. Such asynchronous progression can provide better communication and computation degree of overlap and meantime is designed to take the data locality into consideration.

The contributions of this work address the following three aspects:

- partition cores on a node between application processes and progress processes, by which the non-blocking RMA operations for long messages are intercepted and handled independent of the synchronization calls on the origin side.
- retain the bandwidth performance of the native RMA communication in MPI-3. Meantime, better overlap can be achieved.
- hide the MPI code complexity (see Figures 7.7 and 7.6) from the user. The asynchronous progression can be enabled without changes to the application codes.

I have given a detailed evaluation on the performance improvement offered by the design using a bandwidth micro-benchmark, a Sandia MPI Micro-Benchmark (SMB) and a 2D five-point stencil kernel benchmark. The results demonstrate that the overlap can be achieved without sacrificing the performance of the native MPI RMA communications in bandwidth. According to the SMB evaluation for host processor *overhead*, the DART non-blocking RMA can provides lower host processor *overhead* and higher *availability* than the MPI RMA does. Furthermore, the 2D five-point stencil evaluation shows that using DART non-blocking RMA operations with asynchronous progression enabled produces less calculation time than using MPI RMA interfaces for up to 2048 computing processes. On average, the calculation time performance can be improved by 2.1% on the 32768×32768 cell grid and by 8.4% on the 65536×65536 cell grid. Besides, the performance of the DART implementation can even match that of OpenSHMEM implementation and be superior to that of UPC implementation.

Chapter 8

Application-Level Evaluation

Heat transfer involves the exchange of thermal energy through a physical medium or between physical systems. It is triggered by a temperature difference. Thermal energy transfers from the higher temperature object to the lower temperature object. Basically, there are three fundamental modes transferring heat from one place to another: conduction, convection and radiation. Heat transfer problems are categorized as being one-dimensional (1D), two-dimensional (2D) and three-dimensional (3D). Generally, the medium through which the heat transfer is three-dimensional in the real world.

Heat transfer plays an important role in engineering communities due to several industrial applications such as the cooling of electronic equipment and heat recovery systems, in scientific communities such as global warming, in many manufacturing processes as well, melding and casting, for example. Besides, heat transfer problems are emphasized in daily life concerning the home appliances, such as refrigeration, air-conditioning, ovens and household heaters. Obviously, modeling and solving heat transfer problems are of great scientific and industrial interest. The numerical simulation of the heat transfer phenomenon by iteratively solving a Partial Differential Equation (PDE) is prevailing. However, simulating large 3D heat transfer problems involves a considerable amount of computation and is thus expensively time-consuming. As hinted before, DD schemes can be applied on the parallel solution of complex heat transfer problems to address the time-consuming issue [80–82].

However, data dependency leads to data transfers between neighboring processes. In this sense, the performance bottleneck of the simulation applications could be the communication overhead, which actually closely depends on the applied communication-related directives provided by parallel programming languages or libraries. Specifically, MPI is widely and frequently used in the numerical simulation due to its characteristics of scalability and portability.

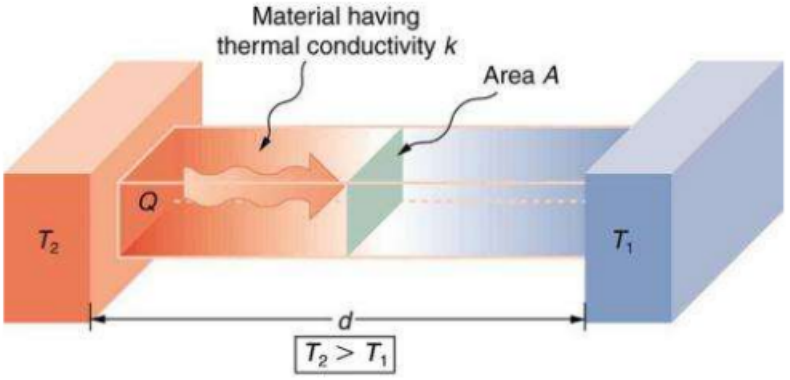


FIGURE 8.1: Heat conduction [4]

8.1 Numerical simulation of 3D heat conduction problem

In this section I present a comprehensive picture of the comparative results between DART-MPI RMA and MPI RMA through a numerical simulation of 3D heat conduction problem.

The heat conduction[4] is a mode of heat transfer owing to molecular activity and occurs in any material (e.g., solids, fluids and gases), shown in Figure 8.1. I simulate the phenomenon of 3D heat conduction in solids with temperature-dependent thermal diffusivity based on an application code parallelized with MPI [83]. In this benchmark, the 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 benchmark solves the PDE over a 3D grid by using the Finite Differential 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. Such communication pattern is shown schematically in Figure 8.2.

Each process exchanges the border data with its corresponding neighbors and the halo cells are reserved to receive the exchanged data. This 3D heat conduction benchmark 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

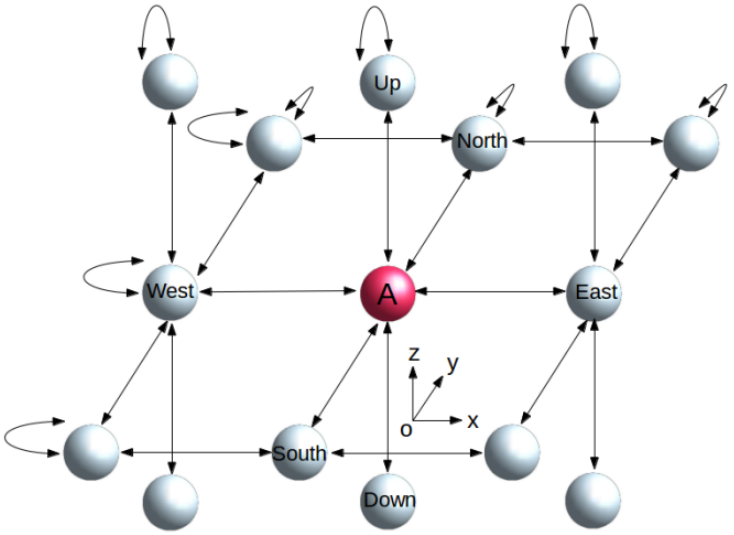


FIGURE 8.2: The communication pattern in the 3D heat conduction algorithm. Each sphere signifies a process. Assuming the process A has all the six neighbors.

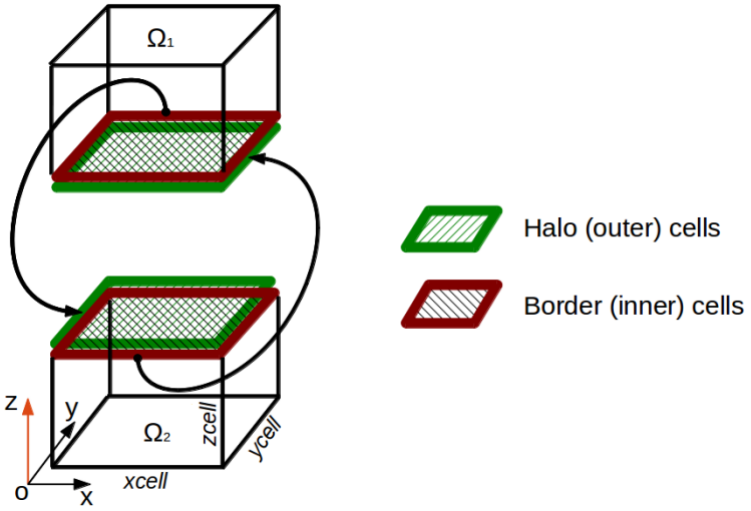


FIGURE 8.3: Halo cells exchange in 3D grid. Two get communications of the matrix data (a total of $xcell \times ycell$ grid cells) happen on Oxy plane between neighboring processes in the direction of z-axis, where sub-domain Ω_2 is the *Down* neighbor of sub-domain Ω_1.

(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 to $size\_z/domain\_z$. Figure 8.3, takes the direction of z-axis for example, to explain the way of boundary data exchange between two neighboring processes.

For my test, I port this implementation to DART one-sided directives. Also, I implement this 3D heat conduction algorithm based on MPI one-sided interfaces using passive

```
/* 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 8.4: Pseudo-code for the halo cells exchange with get and barrier operations.

target mode as the memory synchronization mechanism for a fair comparison. The halo cells exchange is achieved by using blocking get operation. Revisiting Fig. 8.2 we can deduce that six blocking get operations will be invoked for each process. Particularly, *MPI_ Rget* is invoked first and then followed by *MPI_ Wait* in MPI implementation and *dart_ get_ blocking* is used in DART implementation. Note that, within each calculation iteration, after one round of halo cells exchange with certain neighbor an explicit process synchronization, such as barrier (*MPI_ Barrier* in MPI and *dart_ barrier* in DART-MPI), is naively inserted in my 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, I only measure the overhead introduced by the barrier operations instead of the time to wait between processes. Figure 8.4 concisely shows the critical parts of the halo cells exchange code within each calculation iteration. In addition, I 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. I plot the average data results of 25 runs with small execution time variation reported.

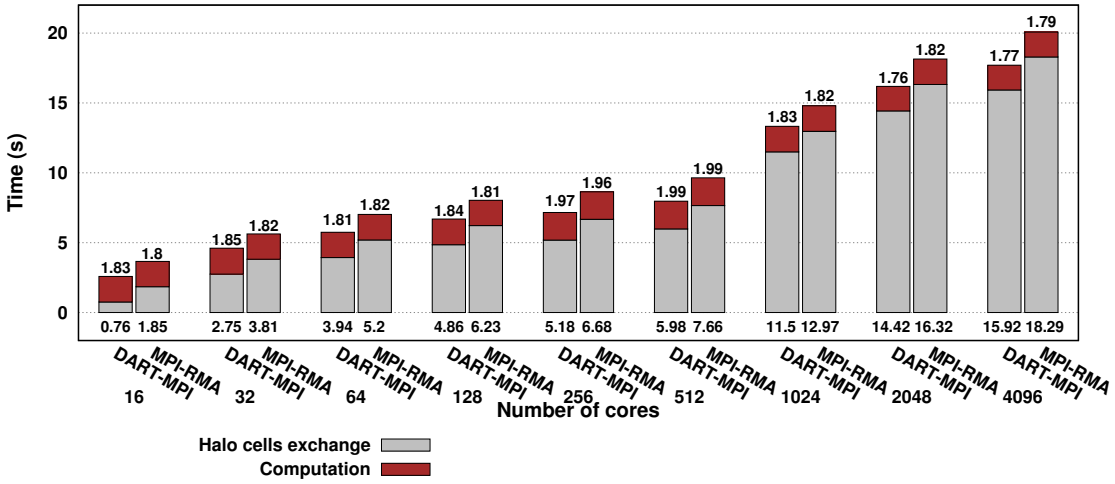


FIGURE 8.5: Weak scaling for one-sided DART and MPI. The grey bar signifies the halo cells exchange time, the value of which is written on the bottom. The brown bar signifies the computation overhead, the value of which is written on the top.

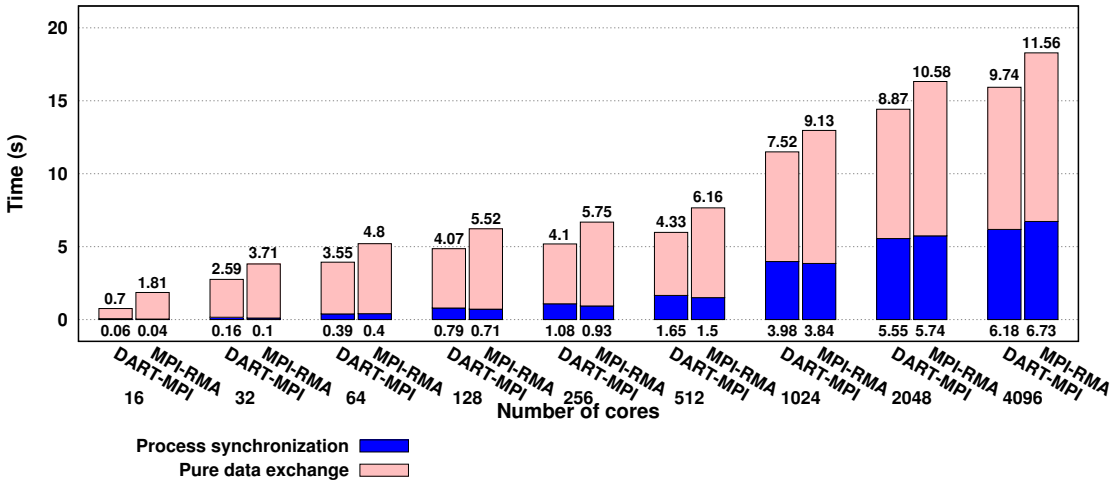


FIGURE 8.6: Breakdown of halo cells exchange time for the weak scaling problem. The blue bar signifies the overall process synchronization overhead, the value of which is written on the bottom. The pink bar signifies the pure data exchange overhead, the value of which is written on the top. The sum equals to the halo cells exchange overhead that I measured in this benchmark.

Figure 8.5 gives the quantitative results showing the performance of DART-MPI RMA and MPI RMA 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 and is applied with grid sizes varying from (64×64×128) to (256×2048×256). The overall calculation is dominated by the halo cells exchange time. 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 MPI implementation is pretty much the

same. The DART implementation can provide a relative speedup of 1.38x over MPI on average in terms of the halo cells exchange time performance. Such speedup is mostly attributed to the enabling of direct load/store access within one node in the DART implementation. Particularly, when only the on-node performance is considered, that is 16 cores, then the improvement can be obviously seen, *cf.* 0.76s for DART-MPI, and 1.85s for MPI. In addition, the consistent improvement seen is expected because most of the data exchanges still happen within one node in this evaluation. Such fact highlights and visualizes the advantage of the memory sharing mechanism in intra-node case employed by DART implementation. Furthermore, 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 RMA uses the same communication infrastructure as MPI RMA internally in the inter-node case.

On the other hand, shown in Fig. 8.5, 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 8.6 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 through Cray XC Rank-3 optical network take place when 1024 cores are launched, which would greatly degrade data exchange performance, in comparison to the inter-blade (intra-chassis) communications through Cray XC Rank-1 backplane network and inter-chassis (intra-group) communications through Cray XC Rank-2 copper network (refer to App. A). 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, I would add that the Fig. 8.6 sufficiently illustrates that the data locality-aware implementation of DART RMA results in the improvement in the halo cells exchange time.

The strong scaling calculation mode can also be applied to compare the performance of the above two implementations with a fixed problem size as the number of cores varies.

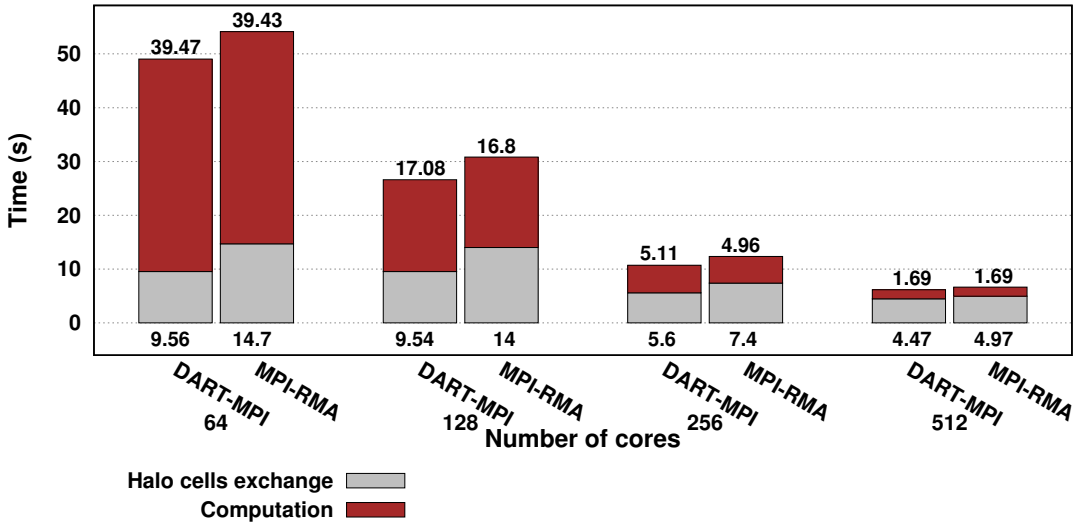


FIGURE 8.7: Strong scaling for one-sided DART and MPI. Refers to Fig. 8.5 for the annotation.

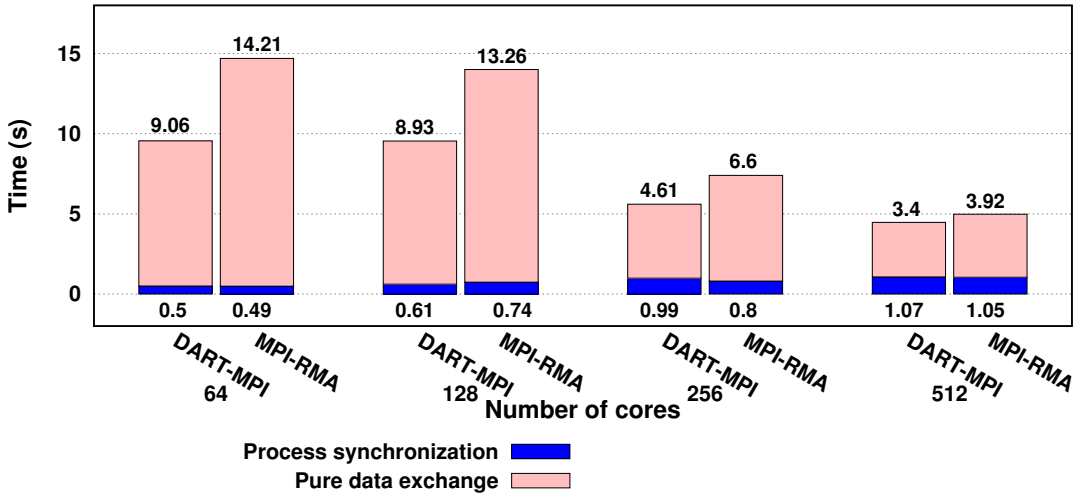


FIGURE 8.8: Breakdown of halo cells exchange time for the strong scaling problem. Refers to Fig. 8.6 for the annotation.

Shown in Figure 8.7 are the quantitative results for a strong scaling analysis on different core counts with grid size (128×256×512). The computation time is shortening as the increase in the number of cores given the problem size keeps constant. Detailedly, the DART implementation provides a speedup of 1.54x, 1.47x, 1.32x and 1.11x in halo cells exchanges on 64 (4×4×4), 128 (4×4×8), 256 (4×8×8) and 512 (8×8×8) processes respectively. This suggests that MPI RMA performs worse than DART-MPI RMA regardless of the message size per RMA communication. Obviously, there is a decline in the halo cells exchange performance speedup as the number of cores grows, as the case in Fig. 8.5. Likewise, Figure 8.8 is provided to show the breakdown of halo cells exchange

time in terms of pure data exchange and process synchronization for this strong scaling scenario. Obviously, the pure data exchange (implemented based on RMA operations) decreases as the higher number of cores is employed. Although the fact that the number of inter-node RMA communications increases as the number of cores doubles (its contribution to the growth in the pure data exchange overhead is very small when ranging the number of cores from 64 to 512, obviously shown in Fig. 8.6), smaller message carried in each RMA message or less RMA messages on certain direction could speed up the halo cells exchange performance. In detail, the value of *zcell* is reduced from 128 to 64 when increasing the number of cores from 64 to 128. This means that the message size carried in each RMA communication on the Oxz and Oyz planes (see Fig. 8.4) is reduced from 1024 to 512 bytes given each cell is a 8-byte number. Revisiting Figures 6.6 and 6.7, we can observe an obvious performance gap in latency between the message size of 512 and 1024 bytes in both the intra-node and inter-node cases for the get blocking operations. The value of *ycell* will be reduced by half when ranging the core counts from 128 to 256. Accordingly, the number of RMA communication operations on the Oyz and Oxy planes will declines, which in turn decreases the halo cell exchange overhead.

8.2 Chapter summary

In this chapter the performance of one-sided MPI and DART are compared using a 3D heat conduction application benchmark up to large number of cores. One-sided MPI comparatively performs poorly for different number of cores and grid sizes according to the strong-scaling and weak-scaling calculation tests. The DART-MPI implementation is proved to be efficient on Cray XC40 system and speeds up the communication (halo cells exchange) by 27% on average over MPI implementation in this application-level evaluation. 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 retaining the MPI inter-node RMA performance. Besides the high-portability and ease-of-program, DART-MPI RMA owns the feature of high performance which is convincingly demonstrated in this chapter. Therefore, when the native implementations of PGAS model are not feasible to implement or easily available on a new hardware, DART, as an MPI-based runtime system for PGAS model, could be an acceptable alternative.

Chapter 9

Conclusions and Future Research Directions

9.1 Summary of research contributions

Parallel computing facilitates and enhances the large-scale simulations in engineering and business. Parallel programming is a software method of enabling parallelism and validates the communication and synchronization necessary for parallel applications. Modern high-performance clusters are equipped with multi-/many-core processors, where embody the feature of the hierarchical memory architecture and the computer processing capabilities are strengthened. Therefore, communication performance could become the bottleneck to achieving high-speed distributed simulations. Partitioned Global Address Space (PGAS) model is ease-to-use, and matches the hierarchical hardware by explicitly exposing the data locality to the user. However, the common low-level RMA communication systems (such as GASNet and ARMCI) serving the PGAS model are supported by the vendors for specific target platforms and thus lack of portability. MPI, as a highly-portable parallel programming model, supports a set of RMA operations and tunes their features and capability in MPI-3 standard. It is naturally believed that MPI can be used as the runtime system of higher-level PGAS model. However, MPI does not intuitively expose the data locality to user applications and provides a rich of interfaces with different semantics or functionalities. Choosing the proper MPI interfaces for writing efficient parallel programs on current hardware is challenging for programmers.

This dissertation is focused on designing an efficient (data locality-aware) and portable runtime system (i.e., DART) for the higher-level PGAS model. This runtime system provides ease-to-use interfaces for the user and is internally implemented based on the MPI communication layer. Hence, the complexity of MPI code is encapsulated inside

this runtime system and completely hidden from programmers. The main contributions of this dissertation are:

- MPI-3 RMA instead of MPI-2 RMA is chosen as the communication conduit of the DART system – I have provided insights into the suitability of implementing one-sided DART based on the MPI-3 unified memory model in Chapter 3. The MPI-3 RMA extends MPI-2's separate memory model with a unified memory model, which provides relaxed semantics so that enhance the usability and effectiveness of the RMA (especially for the RMA operations occurring within the passive epoch with shared lock mode) on machines with automatic cache-coherent architectures. Therefore, it is feasible to implement the one-sided DART based on MPI-3 unified memory model and utilize the passive synchronization mode which is closer in semantics to the truly one-sided communication operations than the active mode for the DART RMA.

- Bridging DART team/group and MPI communicator/group in semantics – In Chapter 4, I have proposed a method that builds the one-to-one relationship between DART team/group and MPI communicator/group. A new DART group should be sorted and thus not straightforwardly created based on MPI group, which would lead to the disorder of DART group. DART team can be uniquely identified by an integer value, called *teamID*. This *teamID* is then bound to another internally controlled value which can be used as an index to the team-to-communicator translation table.

- Designing the DART global memory model with the data locality in mind – This have been achieved by nesting the MPI shared memory window inside the MPI RMA window. In Chapter 5, I have shown that using MPI-integrated shared memory model is a possible solution to take the data locality into consideration. The MPI RMA window could either be the MPI dynamically-created window or the traditional MPI-created window. With dynamically-allocated window, the repeated coarse-grained window creation/destroy operations can be avoided effectively when the collective global memory regions are frequently allocated/deallocated. Although a collective communication operation – *MPI_Allgather* – is entailed when the dynamically-created window is applied, a comparative study has proved that using dynamically-created window for collective global memory management shows higher scalability and better performance than using MPI-created window.

- Designing a data locality-aware communication scheme for DART RMA – In Chapter 6, I have designed DART RMA communication operations with data-locality

awareness based on the nested window structure. Basically, the MPI shared-memory window serves for the intra-node data transfers while the MPI RMA window serves for the inter-node data transfers. I have described the way of using MPI passive target mode to guarantee the correctness of DART RMA operations without sacrificing the performance of MPI RMA operations. The results show that the performance of DART RMA gets improved in latency (more than 50%) in intra-node blocking case, compared to the native MPI RMA operations. Plus, using DART RMA interfaces, the Random Access benchmark consistently delivers better performance than using the native MPI RMA interfaces.

- Enabling asynchronous progression for DART non-blocking RMA communication operations – I have employed the progress process-based approach to design the DART non-blocking RMA operations featuring the asynchronous progression in Chapter 7. Such asynchronous progression can provide better computation-to-communication overlap while retaining the bandwidth performance of the native MPI RMA. The SMB evaluation results show that the DART non-blocking RMA can provide lower host processor *overhead* and higher application *availability* than the MPI RMA does. In addition, compared to MPI put operation, less calculation time can be achieved when using DART non-blocking put operation from the evaluation results for a five-point stencil kernel benchmark.

- Application-level performance evaluation for one-sided MPI and DART – In Chapter 8, I have demonstrated the benefits of one-sided DART for a 3D heat conduction application on high-end system with halo cells exchange time speedup of more than 20% over one-sided MPI using up to 4096 processes. It is proved that the appliance of MPI-3 integrated shared-memory feature in the DART RMA implementation contributes to the communication improvement in this application benchmark. Furthermore, the heat conduction problem is commonly encountered in various engineering applications. It can thus be concluded that DART will be beneficial for a wide spectrum of engineering applications.

9.2 Future research directions

In this dissertation, I have illustrated the design details of DART. DART demonstrates an MPI-based one-sided communication library which works as a runtime system for higher-level PGAS programming models. However, the pursuit of writing large-scale and high-performance parallel applications drives me to explore a number of interesting research topics in the future.

Harvesting richer and more robust features from MPI – Currently, DART defines a minimal set of interfaces that just support PGAS model in a simple way. This work is just a first step in the standardization of DART based on MPI. MPI has not stopped evolving to add more features and adapt to new parallel hardware. Thus, it is useful for DART to take advantage of the powerful functionalities provided in MPI.

MPI provides derived datatypes mechanism for specifying arbitrary, noncontiguous data layout. This mechanism can be utilized to bring significant speedups for some complex parallel applications which involve non-contiguous data communications [84]. Therefore, letting DART one-sided communications allow non-contiguous data transfers with the derived datatypes is indispensable. In addition, the guarantee for the local completion of non-blocking RMA operations should be met in DART to allow finer-grained overlap, as the case in MPI.

The non-blocking collective operations brought in MPI-3 have been proved as an efficient mechanism to bring performance benefits for some applications due to the overlap between communication and computation [85, 86]. Besides, the user applications are able to tolerate the process skew or load imbalance to a certain extent using the non-blocking collective operations. Accordingly, the non-blocking collective operations should be supported in DART on top of blocking collective operations to allow scalable parallel applications.

Support for remote completion notification – 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 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 [87].

Collective communications with data-locality awareness – Basically, the collective communication operations are implemented on top of point-to-point communication functions. The DART one-sided interfaces could be the feasible alternatives to efficiently form the basis for the implementation of collective operations using the light-weight notification scheme (mentioned above). Therefore, with locality-aware one-sided interfaces in DART, the collective communications can be accordingly implemented with data-locality awareness where the communications happen within one node are distinguished from those happen across nodes.

Appendix A

Experimental Setup

I have carried out all the benchmark experiments on the Cray XC40 system, named Hazel Hen at HLRS. 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 and the two NUMA domains are interconnected with each other through QPI. The different compute nodes are interconnected via a Cray Aries network using Dragonfly topology. Figure A.1 shows a single compute node architecture. The Hazel Hen system features a hierarchical network topology. Detailedly, the Cray XC Rank-1 network is used for communications happening across different compute

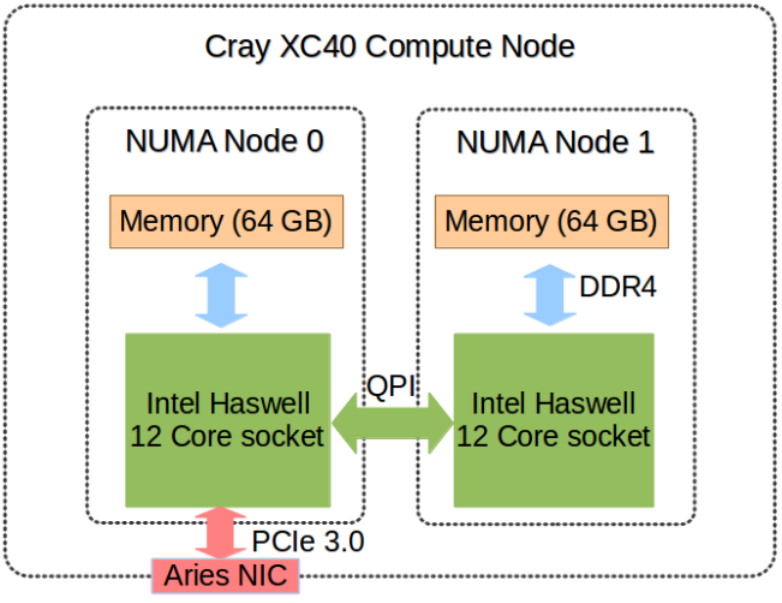


FIGURE A.1: Cray XC40 compute node architecture.

blades within a chassis over backplane. Each chassis consists of 16 compute blades. I.e., there are 64 nodes and thus a total of 1536 cores within a chassis. 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.

All the experiments in this thesis are based on the Cray Programming Environment 5.2.56, where the Cray Compilation Environment (CCE) supports optimized standards for multiple languages and programming models. The Cray Programming Environment supports UPC (through the compiler flag *-h upc*) and OpenSHMEM (as a library) and also supports asynchronous progress for non-blocking RMA. On the HLRS's wiki page [88], we can find a full technical documentation describing the system's hardware and software architecture.

Moreover, in all experiments the tasks are assigned to cores in SMP style [89], which means one node needs to be filled before going to next. Note, that each process is pinned to a physical core.

Bibliography

[1] TOP500 - List Statistics - Nov. 2015. http://www.top500.org/statistics/list/. Accessed: Nov. 2015.

[2] R. Buyya. *High Performance Cluster Computing: Systems and Architectures*, volume 1. Prentice Hall, 1999.

[3] OpenSHMEM Application Programming Interface v1.1 draft. Technical report, Feb. 2014.

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

[5] Partitioned Global Address Space. http://www.pgas.org/, 2015.

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

[7] J. Nieplocha and B. Carpenter. ARMCI: A Portable Remote Memory Copy Library for Distributed Array Libraries and Compiler Run-time Systems. Technical report, 1999.

[8] MPI Forum. MPI: A Message-Passing Interface Standard. Version 3.0. Technical report, Sep. 2012. Available at: http://www.mpi-forum.org.

[9] J. Jose, M. Luo, S. Sur, and D. K. Panda. Unifying UPC and MPI Runtimes: Experience with MVAPICH. In *In Fourth Conference on Partitioned Global Address Space Programming Model*. IEEE Computer Society, 2010.

[10] J. Dinan, P. Balaji, J. R. Hammond, S. Krishnamoorthy, and V. Tipparaju. Supporting the Global Arrays PGAS Model Using MPI One-Sided Communication. In *IPDPS*, pages 739–750. IEEE Computer Society, 2012. ISBN 978-1-4673-0975-2.

[11] J. Daily, A. Vishnu, B. Palmer, and H. van Dam. PGAS Models Using an MPI Runtime: Design Alternatives and Performance Evaluation. In *The International*

Conference for High Performance Computing, Network, Storage and Analysis. IEEE Computer Society, 2013.

[12] D. Bonachea and J. Duell. Problems with Using MPI 1.1 and 2.0 as Compilation Targets for Parallel Language Implementations. *International Journal of High Performance Computing and Networking*, 1(1-3):91–99, 2004. ISSN 1740-0562. doi: 10.1504/IJHPCN.2004.007569.

[13] G. R. Luecke, S. Spanoyannis, and M. Kraeva. The Performance and Scalability of SHMEM and MPI-2 One-sided Routines on a SGI Origin 2000 and a Cray T3E-600. *Concurrency - Practice and Experience*, 16(10):1037–1060, 2004.

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

[15] J. R. Hammond, S. Ghosh, and B. M. Chapman. Implementing OpenSHMEM Using MPI-3 One-Sided Communication. In Stephen W. Poole, Oscar R. Hernandez, and Pavel Shamis, editors, *OpenSHMEM*, volume 8356 of *Lecture Notes in Computer Science*, pages 44–58. Springer, 2014. ISBN 978-3-319-05214-4.

[16] 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*, pages 542–552, 2014. doi: 10.1007/978-3-319-14313-2_46.

[17] DASH Project Documentation. http://doc.dash-project.org/. Accessed: Apr. 2016.

[18] B. Barney. Introduction to Parallel Computing. https://computing.llnl.gov/tutorials/parallel_comp/, . Accessed: Nov. 2015.

[19] W. D. Gropp. Parallel Computing and Domain Decomposition. In *Fifth Conference on Domain Decomposition Methods for Partial Differential Equations*, pages 349–361, 1992.

[20] L. Chai, Q. Gao, and D. K. Panda. Understanding the Impact of Multi-Core Architecture in Cluster Computing: A Case Study with Intel Dual-Core System. In *Cluster Computing and the Grid, 2007. CCGRID 2007. Seventh IEEE International Symposium on*, pages 471–478, May 2007. doi: 10.1109/CCGRID.2007.119.

[21] B. Barney. POSIX Threads Programming. https://computing.llnl.gov/tutorials/pthreads/, . Accessed: Nov. 2015.

[22] OpenMP Application Programming Interface Version 4.5. Technical report, Nov. 2015.

[23] B. Barney. OpenMP. https://computing.llnl.gov/tutorials/openMP/, . Accessed: Nov. 2015.

[24] MPI Forum. MPI: A Message-Passing Interface Standard. Version 2.2. Technical report, Sep. 2009. Available at: http://www.mpi-forum.org.

[25] MPICH Overview. http://www.mpich.org/about/overview/. Accessed: Apr. 2016.

[26] W. Pfeiffer and A. Stamatakis. Hybrid MPI/Pthreads Parallelization of the RAxML Phylogenetics Code. In *Ninth IEEE International Workshop on High Performance Computational Biology (HiCOMB)* , 2010.

[27] E. L. Lusk and A. Chan. Early Experiments with the OpenMP/MPI Hybrid Programming Model. In Rudolf Eigenmann and Bronis R. de Supinski, editors, *IWOMP*, volume 5004 of *Lecture Notes in Computer Science*, pages 36–47. Springer, 2008. ISBN 978-3-540-79560-5.

[28] R. Rabenseifner, G. Hager, and G. Jost. Hybrid MPI/OpenMP Parallel Programming on Clusters of Multi-Core SMP Nodes. In *Proceedings of the 17th Euromicro International Conference on Parallel, Distributed, and Network-Based Processing (PDP 2009)*, pages 427–436, Feb. 2009.

[29] W. Gropp. MPI, Hybrid Programming, and Shared Memory . Technical report, 2014.

[30] W. Carlson, J. Draper, D. Culler, K. Yelick, E. Brooks, and K. Warren. Introduction to UPC and Language Specification. Technical Report CCS-TR-99-157, IDA Center for Computing Sciences, 1999.

[31] R. W. Numrich and J. Reid. Co-array Fortran for Parallel Programming. *SIGPLAN Fortran Forum*, 17(2):1–31, Aug. 1998.

[32] J. Mellor-Crummey, L. Adhianto, W. N. III Scherer, and G. Jin. A New Vision for Coarray Fortran. In *Proceedings of the 3rd Conf. on Partitioned Global Address Space Programming Models, PGAS'09*, pages 1–9, 2009.

[33] B. L. Chamberlain, D. Callahan, and H. P. Zima. Parallel Programmability and the Chapel Language. *International Journal of High Performance Computing Applications*, 21(3):291–312, Aug. 2007.

[34] V. Saraswat, B. Bloom, I. Peshansky, O. Tardieu, and D. Grove. X10 Language Specification. Technical report, IBM, Jan. 2012.

[35] A. A. Buss, Harshvardhan, I. Papadopoulos, O. Pearce, T. G. Smith, G. Tanase, N. Thomas, X. Xu, M. Bianco, N. M. Amato, and L. Rauchwerger. STAPL: Standard Template Adaptive Parallel Library. In *SYSTOR*, ACM International Conference Proceeding Series. ACM, 2010. ISBN 978-1-60558-908-4.

[36] K. A. Yelick, L. Semenzato, G. Pike, C. Miyamoto, B. Liblit, A. Krishnamurthy, P. N. Hilfinger, S. L. Graham, D. Gay, P. Colella, and A. Aiken. Titanium: A High-performance Java Dialect. *Concurrency - Practice and Experience*, 10(11-13): 825–836, 1998.

[37] V. Saraswat, G. Almasi, G. Bikshandi, C. Cascaval, D. Grove, D. Cunningham, O. Tardieu, I. Peshansky, and S. Kodali. The Asynchronous Partitioned Global Address Space Model. In *Proc. First Workshop Advances in Message Passing*, 2010.

[38] X10RT. X10: Performance and Productivity at Scale. http://x10-lang.org/documentation/x10rt.html. Accessed: Apr. 2016.

[39] S. Poole, O. Hernandez, J. Kuehn, G. Shipman, A. Curtis, and K. Feind. OpenSHMEM - Toward a Unified RMA Model. In David Padua, editor, *Encyclopedia of Parallel Computing*, pages 1379–1391. Springer US, 2011.

[40] J. Nieplocha, R. J. Harrison, and R. J. Littleeld. Global Arrays: A Nonuniform Memory Access Programming Model for High-performance Computers. *Journal of Supercomputing*, 10:169–189, 1996.

[41] 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 Allen D. Malony and Jeff R. Hammond, editors, *PGAS*, pages 3:1–3:11. ACM, 2014. ISBN 978-1-4503-3247-7.

[42] J. Dinan, P. Balaji, D. Buntinas, D. Goodell, W. Gropp, and R. Thakur. An Implementation and Evaluation of the MPI 3.0 One-sided Communication Interface. In *Preprint ANL/MCS-P4014-0113* . IEEE Computer Society, 2013.

[43] S. Sahni and G. Vairaktarakis. The Master–slave Paradigm in Parallel Computer and Industrial Settings. *Journal of Global Optimization*, pages 357–377, Sep. 1996.

[44] A. A. Kamil and K. A. Yelick. Hierarchical Additions to the SPMD Programming Model. Technical Report UCB/EECS-2012-20, EECS Department, University of California, Berkeley, 2012.

[45] S. Qin. Merge Sort Algorithm. http://cs.fit.edu/~pkc/classes/writing/hw13/song.pdf. Accessed: Apr. 2016.

[46] C. A. Shaffer. *Data Structures and Algorithm Analysis*. Dover Publishing, 3rd ed. edition, Jun. 2012.

[47] R. R. Schaller. Moore's Law: Past, Present and Future. *Spectrum, IEEE*, 34(6): 52–59, Jun. 1997. ISSN 0018-9235. doi: 10.1109/6.591665.

[48] 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 Jesper Larsson Träff, Siegfried Benkner, and Jack J. Dongarra, editors, *EuroMPI*, volume 7490 of *Lecture Notes in Computer Science*, pages 132–141. Springer, 2012. ISBN 978-3-642-33517-4.

[49] S. Potluri, S. Sur, D. Bureddy, and D. K. Panda. Design and Implementation of Key Proposed MPI-3 One-Sided Communication Semantics on InfiniBand. In Yiannis Cotronis, Anthony Danalis, Dimitrios S. Nikolopoulos, and Jack Dongarra, editors, *EuroMPI*, volume 6960 of *Lecture Notes in Computer Science*, pages 321–324. Springer, 2011. ISBN 978-3-642-24448-3.

[50] H. Zhou, K. Idrees, and J. Gracia. Leveraging MPI-3 Shared-Memory Extensions for Efficient PGAS Runtime Systems. In Jesper Larsson Träff, Sascha Hunold, and Francesco Versaci, editors, *Euro-Par*, volume 9233 of *Lecture Notes in Computer Science*, pages 373–384. Springer, 2015. ISBN 978-3-662-48095-3.

[51] G. S. Brodal, E. D. Demaine, and J. I. Munro. Fast Allocation and Deallocation with an Improved Buddy System. *Acta Inf.*, 41(4-5):273–291, 2005.

[52] K. Vaidyanathan, L. Chai, W. Huang, and D. K. Panda. Efficient Asynchronous Memory Copy Operations on Multi- Core Systems and I/OAT. In *Proceedings of the IEEE International Conference on Cluster Computing (Cluster'07)*, Sep. 2007.

[53] A. Anghel, G. Rodríguez, B. Prisacari, C. Minkenberg, and G. Dittmann. Quantifying Communication in Graph Analytics. In Julian M. Kunkel and Thomas Ludwig, editors, *ISC*, volume 9137 of *Lecture Notes in Computer Science*, pages 472–487. Springer, 2015. ISBN 978-3-319-20118-4.

[54] Graph 500 Brief Introducation. http://www.graph500.org/. Accessed: Aug. 2015.

[55] H. W. Jin, S. Sur, L. Chai, and D. K. Panda. LiMIC: Support for High-Performance MPI Intra-Node Communication on Linux Cluster. 2005.

[56] 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. ISBN 0-7695-2132-0.

[57] 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*, 25(1-2): 3–14, 2010.

[58] R. Gerstenberger, M. Besta, and T. Hoefler. Enabling Highly-scalable Remote Memory Access Programming with MPI-3 One Sided. In William Gropp and Satoshi Matsuoka, editors, *SC*, pages 53:1–53:12. ACM, 2013. ISBN 978-1-4503-2378-9.

[59] 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*, volume 6960 of *Lecture Notes in Computer Science*, pages 99–109. Springer, 2011. ISBN 978-3-642-24448-3.

[60] R. Thakur, R. Rabenseifner, and W. Gropp. Optimization of Collective Communication Operations in MPICH. *IJHPCA*, 19(1):49–66, 2005.

[61] T. Ma, T. Hérault, G. Bosilca, and J. J. Dongarra. Process Distance-Aware Adaptive MPI Collective Communications. In *CLUSTER*, pages 196–204. IEEE Computer Society, 2011. ISBN 978-1-4577-1355-2.

[62] 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. ISBN 0-7695-2132-0.

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

[64] S. Sur, U. Bondhugula, A. R. Mamidala, H.-W. Jin, and D. K. Panda. High Performance RDMA Based All-to-All Broadcast for InfiniBand Clusters. In *HiPC*, volume 3769 of *Lecture Notes in Computer Science*, pages 148–157. Springer, 2005. ISBN 3-540-30936-5.

[65] A. R. Mamidala, J. Liu, and D. K. panda. Efficient Barrier and Allreduce InfiniBand Clusters Using Hardware Multicast and Adaptive Algorithms. 2004.

[66] R. Thakur, W. D. Gropp, and B. R. Toonen. Minimizing Synchronization Overhead in the Implementation of MPI One-Sided Communication. In *PVM/MPI*, volume 3241 of *Lecture Notes in Computer Science*, pages 57–67. Springer, 2004. ISBN 3-540-23163-3.

[67] H. Zhou, V. Marjanovic, C. Niethammer, and J. Gracia. A Bandwidth-saving Optimization for MPI Broadcast Collective Operation. In *Proceedings of the International Conference on Parallel Processing Workshops, ICPPW*, Sep. 2015.

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

[69] RandomAccess GUPS (Giga Updates Per Second). http://icl.cs.utk.edu/projectsfiles/hpcc/RandomAccess. Accessed: Nov. 2015.

[70] P. Luszczek, J. Dongarra, D. Koester, R. Rabenseifner, B. Lucas, J. Kepner, J. McCalpin, D. Bailey, and D. Takahashi. Introduction to the HPC Challenge Benchmark Suite. Technical report, Mar. 2005.

[71] P. Shamis, M. G. Venkata, S. W. Poole, A. Welch, and T. Curtis. Designing a High Performance OpenSHMEM Implementation Using Universal Common Communication Substrate as a Communication Middleware. In *OpenSHMEM*, volume 8356 of *Lecture Notes in Computer Science*, pages 1–13. Springer, 2014. ISBN 978-3-319-05214-4.

[72] R. Brightwell, R. Riesen, and K. D. Underwood. Analyzing the Impact of Overlap, Offload, and Independent Progress for Message Passing Interface Applications. *IJHPCA*, 19(2):103–117, 2005.

[73] M. Wittmann, G. Hager, T. Zeiser, and G. Wellein. Asynchronous MPI for the Masses. *CoRR*, abs/1302.4280, 2013.

[74] M. Si, A. J. Peña, J. R. Hammond, P. Balaji, M. Takagi, and Y. Ishikawa. Casper: An Asynchronous Progress Model for MPI RMA on Many-Core Architectures. In *IPDPS*, pages 665–676. IEEE Computer Society, 2015. ISBN 978-1-4799-8649-1.

[75] H. Pritchard, D. Roweth, D. Henseler, and P. Cassella. Leveraging the Cray Linux Environment Core Specialization Feature to Realize MPI Asynchronous Progress on Cray XE Systems. In *Proceedings of the Cray User Group Conference*, May 2012.

[76] Understanding MPI on Cray XE6. In *Cray XE6 Performance Workshop*, 2013.

[77] M. Li, S. Potluri, K. Hamidouche, J. Jose, and D. K. Panda. Efficient and Truly Passive MPI-3 RMA Using InfiniBand Atomics. In *EuroMPI*, pages 91–96. ACM, 2013. ISBN 978-1-4503-1903-4.

[78] Host Processor Overhead. http://www.cs.sandia.gov/smb/overhead.html. Accessed: Nov. 2015.

[79] D. Doerfler and R. Brightwell. Measuring MPI Send and Receive Overhead and Application Availability in High Performance Network Interfaces. In *13th European PVM/MPI User's Group Meeting*, pages 331–338, Sep. 2006.

[80] M. P. Raju and S. Khaitan. Domain Decomposition Based High Performance Parallel Computing. *M. P. Raju and S. Khaitan, "Domain Decomposition Based High Performance Parallel Computing", International Journal of Computer Science Issues,IJCSI, Volume 5, pp27-32*, 2009.

[81] C. J. Palansuriya, C.-H. Lai, C. S. Lerotheou, and K. A. Pericleous. A Domain Decomposition based Algorithm for Nonlinear 2D Inverse Heat Conduction Problems. 218:515–522, 1998.

[82] V. Horak and P. Gruber. Parallel Numerical Solution of 2D Heat Equation. In *Parallel Numerics '05*, pages 47–56, 2005.

[83] MPI Parallelization for numerically solving the 3D Heat equation. https://dournac.org/info/parallel_heat3d. Accessed: Apr. 2016.

[84] T. Hoefler and S. Gottlieb. Parallel Zero-Copy Algorithms for Fast Fourier Transform and Conjugate Gradient Using MPI Datatypes. In *Recent Advances in the Message Passing Interface (EuroMPI'10)*, volume LNCS 6305, pages 132–141, Sep. 2010.

[85] T. Hoefler, J. M. Squyres, W. Rehm, and A. Lumsdaine. A Case for Non-blocking Collective Operations. In *ISPA Workshops*, volume 4331 of *Lecture Notes in Computer Science*, pages 155–164. Springer, 2006. ISBN 3-540-49860-5.

[86] T. Hoefler, A. Lumsdaine, and W. Rehm. Implementation and Performance Analysis of Non-Blocking Collective Operations for MPI. In *Proceedings of the 2007 International Conference on High Performance Computing, Networking, Storage and Analysis, SC07*. IEEE Computer Society/ACM, Nov. 2007.

[87] R. Belli and T. Hoefler. Notified Access: Extending Remote Memory Access Programming Models for Producer-Consumer Synchronization. In *IPDPS*, pages 871–881. IEEE Computer Society, 2015. ISBN 978-1-4799-8649-1.

[88] Cray XC40. https://wickie.hlrs.de/platforms/index.php/CRAY_XC40_Hardware_and_Architecture. Accessed: Apr. 2016.

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