OPENSHMEM AS AN EFFECTIVE COMMUNICATION LAYER FOR PGAS MODELS

A Thesis

Presented to

the Faculty of the Department of Computer Science

University of Houston

In Partial Fulfillment

of the Requirements for the Degree

Master of Science

By

Naveen Namashivayam Ravichandrasekaran

December 2015

OPENSHMEM AS AN EFFECTIVE COMMUNICATION LAYER FOR PGAS MODELS

Naveen Namashivayam Ravichandrasekaran

APPROVED:

Barbara Chapman, Chairman
Dept. of Computer Science

Edgar Gabriel
Dept. of Computer Science

Mikhail Sekachev
TOTAL E&P Research and Technology USA, LLC

Dean, College of Natural Sciences and Mathematics

Acknowledgements

I would like to thank my advisor, Prof. Barbara Chapman, for her guidance, support and her faith in my capability without which this thesis wouldn't have materialized. I would also like to thank my other committee members, Prof. Edgar Gabriel and Mikhail Sekachev for taking time to review my work and providing valuable comments.

I would like to thank my mentor Deepak Eachempati, for his constant support through numerous stimulating discussions and general advice. I am very grateful to Dounia Khaldi, for providing her support in getting access to numerous supercomputing clusters, analyzing the problem at hand, organizing and presenting the results. I'm forever indebted to Tony Curtis, for helping me in my early stages of work in HPC Tools and his patience in introducing me to the whole new world of high performance computing, in particular to OpenSHMEM.

I would like to acknowledge the help of Sunita Chandrasekaran, and Yonghong Yan for introducing me to the HPC Tools group. I would like to thank Sid, for his valuable feedback and numerous engaging discussions over the past two years. In general, I would like to thank everyone in HPC Tools for their assistance with all types of technical problems & discussions.

I would like to thank Mark Pagel, David Knaak, Bob Cernohous, Krishna Kandalla, Steven Oyanagi, Dan Pou, and all other members of MPT team in Cray Inc., for their patience and support during the last few months of my thesis.

This work is supported through funding from TOTAL. I would like to thank Henri Calandra and others in TOTAL, for graciously allowing me to use their Cray system

for experimentation and performance evaluations. I would like to thank TOTAL, Texas Advanced Computing Center (TACC) and Oak Ridge Leadership Computing Facility (OLCF) for providing computing resources for the experiments.

Finally and most importantly, I would like to thank my parents for their support, encouragement and being the pillar of my life.

This work is dedicated to:

my parents Bhavani and Ravichandrasekaran

OPENSHMEM AS AN EFFECTIVE COMMUNICATION LAYER FOR PGAS MODELS

An Abstract of a Thesis

Presented to

the Faculty of the Department of Computer Science

University of Houston

In Partial Fulfillment

of the Requirements for the Degree

Master of Science

By

Naveen Namashivayam Ravichandrasekaran

December 2015

Abstract

Languages and libraries based on the Partitioned Global Address Space (PGAS) programming model have emerged in recent years with a focus on addressing the programming challenges for scalable parallel systems. Among these, Coarray Fortran (CAF) is unique in that as it has been incorporated into an existing standard (Fortran 2008), and therefore it is of particular importance that implementations supporting it are both portable and deliver sufficient levels of performance. OpenSHMEM is a library which is the culmination of a standardization effort among many implementers and users of SHMEM, and it provides a means to develop light-weight, portable, scalable applications based on the PGAS programming model. As such, we propose here that OpenSHMEM is well situated to serve as a runtime substrate for other PGAS programming models.

In this work, we demonstrate how OpenSHMEM can be exploited as a runtime layer upon which CAF may be implemented. Specifically, we re-targeted the CAF implementation provided in the OpenUH compiler to OpenSHMEM, and show how parallel language features provided by CAF may be directly mapped to OpenSHMEM, including allocation of remotely accessible objects, one-sided communication, and various types of synchronization. Moreover, we present and evaluate various algorithms we developed for implementing remote access of non-contiguous array sections, and acquisition and release of remote locks using the OpenSHMEM interface. Through this work, we argue for specific features like block-wise strided data transfer, multi-dimensional strided data transfer, and atomic memory operations which may be added to OpenSHMEM to better support idiomatic usage of CAF.

Contents

1 Introduction **1**

1.1 Motivation 3

1.2 Contributions 4

1.3 Thesis Organization 5

2 Background **6**

2.1 Defining Programming Models 6

2.2 Coarray Fortran 10

2.2.1 Representation and Execution 12

2.2.2 Types of Data Objects 12

2.2.3 Data Communication 12

2.2.4 Atomic Operations 13

2.2.5 Synchronization, Locks and Critical section 13

2.2.6 Survey on CAF Implementations 14

2.3 OpenSHMEM 17

2.3.1 Remote Memory Access 17

2.3.2 Synchronization 18

2.3.3 Collective Communication 18

2.3.4 Mutual Exclusion 19

2.3.5 Overview on OpenSHMEM Implementations 19

2.3.6 OpenSHMEM Reference Implementation 20
2.4 Architectural Overview 20

3 Relative-Performance Comparisons 23
3.1 OpenSHMEM for Inter-Node Communication 23
3.2 OpenSHMEM for Intra-Node Communication 26
3.2.1 Intel Xeon Phi Architecture 27
3.2.2 From Remote Memory Access to Local Load/Store Operations 29
3.2.3 Introducing Different Reduction Algorithms 31
3.2.4 Flat Tree (FT) 34
3.2.5 Recursive Doubling (RD) 35
3.2.6 Bruck (BR) 36
3.2.7 Rabenseifner (RSAG) 37
3.2.8 Performance Analysis of Different Reduction Algorithms . . . 39
3.2.9 Application of `shmem_ptr` on Reduction Algorithms 40
3.3 Why OpenSHMEM? 41

4 Implementation 44
4.1 CAF and OpenSHMEM Comparison 44
4.2 Symmetric Memory Allocation 45
4.3 Remote Memory Access (RMA) - Contiguous Data Communication . 46
4.4 Remote Memory Access (RMA) - Strided Data Communication . . . 47
4.5 Support for Locks 51
4.6 Unsupported Features 55

5 Experimental Results 56
5.1 Experimental Setup 56
5.2 PGAS Microbenchmarks 57

5.2.1 Contiguous Communications 58
5.2.2 Strided Communications 60
5.2.3 Locks 61
5.3 Distributed Hash Table (DHT) Benchmark 62
5.4 Himeno Parallel Benchmark 63

6 Related Work **65**

7 Conclusion **67**

8 Terms and Abbreviations **69**

Bibliography **72**

List of Figures

2.1 Types of Parallel Programming Models 7

2.2 Overview of PGAS Programming Models 9

2.3 An OpenSHMEM variant of a CAF code 11

2.4 Brief Overview of OpenSHMEM API with both C and Fortran Bindings 18

2.5 Different OpenSHMEM Implementations 19

2.6 Overview of OpenSHMEM and CAF Architectures 21

3.1 Put Latency comparison using two nodes (inter-node communication) for SHMEM, MPI-3.0 (RMA) and GASNet 25

3.2 Put Bandwidth comparison using two nodes (inter-node communication) for SHMEM, MPI-3.0 (RMA) and GASNet 26

3.3 Xeon Phi Microarchitecture 28

3.4 The use of `shmem_ptr` 30

3.5 Latency comparison put/get vs. `shmem_ptr` 31

3.6 Bandwidth comparison of communications between 1 and 2 pairs of PEs: load/store using shmem_ptr vs calls to shmem_put/shmem_get; note that green and red lines overlap 32

3.7 Bandwidth comparison of communications between 8 and 64 pairs of PEs: load/store using shmem_ptr vs calls to shmem_put/shmem_get; note that green and red lines overlap 33

3.8 Performance of different reduction algorithms implemented in OpenSHMEM vs. MPI using 64 PEs on Intel Xeon Phi (1 to 2^{10} and 2^{12} to 2^{20} number of integers) 38

3.9 SHMEM reduction algorithm performance using put/get vs. `shmem_ptr` using 64 PEs for small data sizes 39

3.10 SHMEM reduction algorithm performance using put/get vs. `shmem_ptr` using 64 PEs for large data sizes 40

3.11 Performance of different reduction algorithms using `shmem_ptr` vs. Intel MPI and MVAPICH using 64 PEs on Intel Xeon Phi 42

4.1 Example code on CAF RMA local and remote completion 47

4.2 Different Multi-dimensional Stride Orientation 48

4.3 Example for Multi-dimensional Strides 51

5.1 PGAS Microbenchmark Tests on Cray XC30: Put Bandwidth and 2-Dimensional Strided Put bandwidth 58

5.2 PGAS Microbenchmark Tests on Stampede: Put Bandwidth and 2-Dimensional Strided Put bandwidth 59

5.3 Microbenchmark test for locks on Titan. All images trying to attain and release lock on image 1. 61

5.4 Test for locks using Distributed Hash Table Benchmark in Titan . . . 62

5.5 Test for strided data communication using Himeno Benchmark in Stampede 64

List of Tables

2.1 Different CAF Implementations and their Underlying Communication Layers 15

3.1 Experimental Setup and Machine configuration details for Relative Performance Comparisons of OpenSHMEM, MPI-3.0(RMA) and GASNet 24

4.1 Summary of features for supporting parallel execution in Coarray Fortran (CAF) and OpenSHMEM 45

5.1 Experimental Setup and Machine configuration details for Performance Analysis of CAF over OpenSHMEM Implementation 56

Chapter 1

Introduction

In recent years, a class of languages and libraries referred to as *Partitioned Global Address Space* (PGAS) [8] has emerged as a highly scalable approach for programming large-scale parallel systems. The PGAS programming model is characterized by a logically partitioned global memory space, where partitions have affinity to the processes/threads executing the program. This property allows PGAS-based applications to specify an explicit data decomposition that reduces the number of remote accesses with longer latencies. This model attempts to combine both the advantages of the SPMD model for distributed systems with the data referencing semantics of shared memory systems.

Several languages and libraries provide a PGAS programming model, some with extended capabilities such as asynchronous spawning of computations at remote locations in addition to facilities for expressing data movement and synchronization. Unified Parallel C (UPC) [13], Coarray Fortran (CAF) [34], X10 [17], Chapel [14], and

CAF 2.0 [28] are examples of language-based PGAS models, while OpenSHMEM [15] and Global Arrays [32] are examples of library-based PGAS models. Library-based models assume the use of a base language (C/C++ and Fortran being typical) which does not require or generally benefit from any special compiler support. This essentially means the library-based PGAS implementations rely completely on the programmer to use the library calls to implement the correct semantics of its programming model.

Over the years there have been numerous system-level interfaces which have been used for both language- and library-based PGAS implementations, including GASNet [10], InfiniBand Verbs API [27], MVAPICH2-X [22], ARMCI [31], and Cray's uGNI [18], and DMAPP [39]. Several OpenSHMEM implementations are available over a broad range of parallel platforms, including implementations based on GASNet, ARMCI, MPI-3.0 [21], and MVAPICH2-X [5]. The Berkeley UPC project uses GASNet, and the University of Houston's CAF implementation [16] works with either GASNet or ARMCI. The SHMEM, CAF, and UPC implementations [41] from Cray use DMAPP.

OpenSHMEM, referred to above as a library-based PGAS model, is a library interface specification which is the culmination of a unification effort among many vendors and users in the SHMEM programming community. Some important SHMEM implementations, that follow the OpenSHMEM standards are discussed in Section 2.3. OpenSHMEM was designed to be implemented as a light-weight and portable library offering the facilities required by PGAS-based applications. In this work, we will examine the suitability of OpenSHMEM as a runtime layer for other PGAS languages.

In particular, we focus on how Coarray Fortran may be implemented on top of OpenSHMEM.

1.1 Motivation

The motivation for such a mapping is that OpenSHMEM is widely available, and it provides operations with very similar semantics to the parallel processing features in CAF – thus the mapping can be achieved with very little overhead which we show. Furthermore, such an implementation allows us to incorporate OpenSHMEM calls directly into CAF applications (i.e., Fortran 2008 applications using coarrays and related features) and explore the ramifications of such a hybrid model.

We found that not all operations in CAF have an efficient 1-to-1 mapping to OpenSHMEM. In particular, we developed in this work new translation strategies for handling remote access involving multi-dimensional strided data and acquisition and release of remote locks. Strided communications for multi-dimensional arrays are fundamental in many scientific applications and warrant careful consideration into how they may be efficiently handled in PGAS implementations. The OpenSHMEM API supports *indexed* remote memory access operations which work on only 1-dimensional strided data, and its interface for locks is unsuitable for CAF. Therefore, in this work we introduce a new algorithm for the general case of strided data communications in multi-dimensional arrays and also an adaptation of the MCS lock algorithm [29] for the case of coarray locks on a specified image.

1.2 Contributions

The contributions of this work include:

- analyzing various OpenSHMEM implementations by relative comparisons to other available libraries and system interfaces including MPI-3.0(RMA), MPI, and GASNet;
- analyzing the use of OpenSHMEM for both inter- and intra-node communications, we have made use of Intel Xeon Phi [3] for the purpose of intra-node communications. We specifically targeted Intel Xeon Phi for this performance study to understand the usage of OpenSHMEM on many core architectures;
- use of OpenSHMEM as a portable communication layer for CAF, more specifically retargeting CAF implementation in OpenUH compiler to OpenSHMEM;
- introduction of new algorithms for handling strided data communications for multi-dimensional arrays and the adaptation of the MCS lock algorithm to the case of non-collective CAF locks;
- a comprehensive evaluation of this CAF-OpenSHMEM implementation using our PGAS microbenchmarks [2], a coarray version of a distributed hash table code [26], and Himeno Benchmark [1].

1.3 Thesis Organization

This work is organized as follows: In Chapter 2 we discuss briefly on the various available PGAS models with special preference to CAF and OpenSHMEM in Section 2.2 and Section 2.3, respectively and their corresponding communication layers upon which these PGAS models are implemented. We motivate the use of OpenSHMEM as a communication layer for CAF in Chapter 3. We show the translation of the features of CAF to OpenSHMEM in Chapter 4. Experimental results using PGAS microbenchmarks, a distributed hash table benchmark, and the Himeno benchmark are presented in Chapter 5. We survey different related designs of PGAS models in Chapter 6. Finally, we discuss future work and conclude in Chapter 7.

Chapter 2

Background

In this Chapter, we will get to know about different PGAS programming models and in particular about CAF and OpenSHMEM. In order to understand about these different PGAS programming models, we need to briefly explain what we mean by PGAS programming model and what are the different classes of PGAS models.

2.1 Defining Programming Models

Programming models can be considered as an abstraction for programming interfaces and systems. They are typically used to define how the data objects are referenced and stored. They determine the design through which the computational tasks are defined and made to excecute. Typically, programming models are divided into sequential and parallel programming models based on the execution model.

Parallel-programming models provide the programmer with an abstraction of how the system will execute the parallel application and how its data will be laid out in memory. Based on the memory layout, we have three different parallel programming models as shown in Figure 2.1

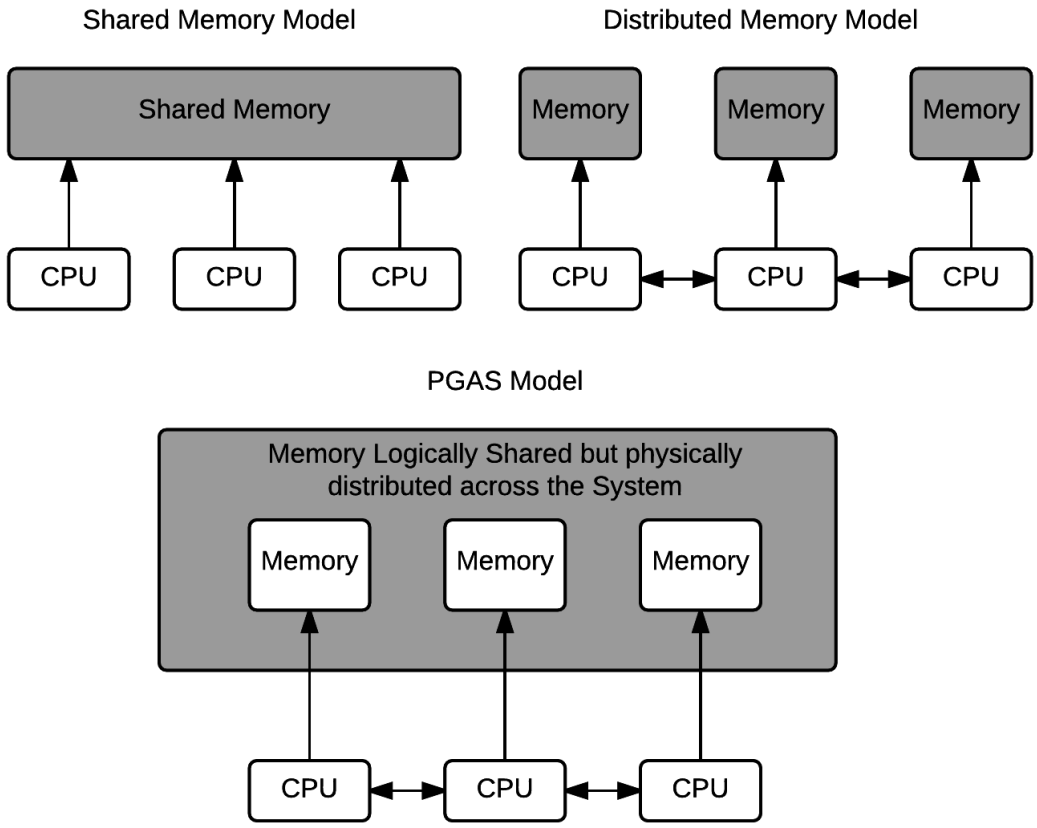

Figure 2.1: Types of Parallel Programming Models

The three different parallel programming model are:

- Shared-Memory Model - a model in which multiple threads execute their own instruction streams and share the same address space. The shared memory model does not burden the programmers with data partitioning responsibilities. The data de-referencing semantics of the shared memory model can be considered as its one of the important property.

- Distributed-Memory Model - a model in which each thread execute their own set of instruction streams and have access only to their local address space. Access to the remote address space is done through passing message to the remote CPU and then receiving the data from the remote address space. Usually this type of programming model follows the Single Program, Multiple Data (SPMD) style of programming.

- Partitioned-Global-Address-Space Model (PGAS) - PGAS programming model attempts to combine both the advantages of the SPMD style of programming from the distributed systems with the data referencing semantics of the shared memory systems. Here, each thread has access to their local address space as well as to a symmetric heap distributed across the distributed system.

In general, PGAS models are broadly classified as language- and library-based models. Figure 2.2 provides a brief overview on different layers of PGAS programming models and details about their communication substrate/system interface.

OpenSHMEM, RMA featureset in MPI-3.0, and Global Arrays belong to the library-based PGAS models. Library-based models do not assume the usage of a base language and they do not in general require any special compiler support. Hence,

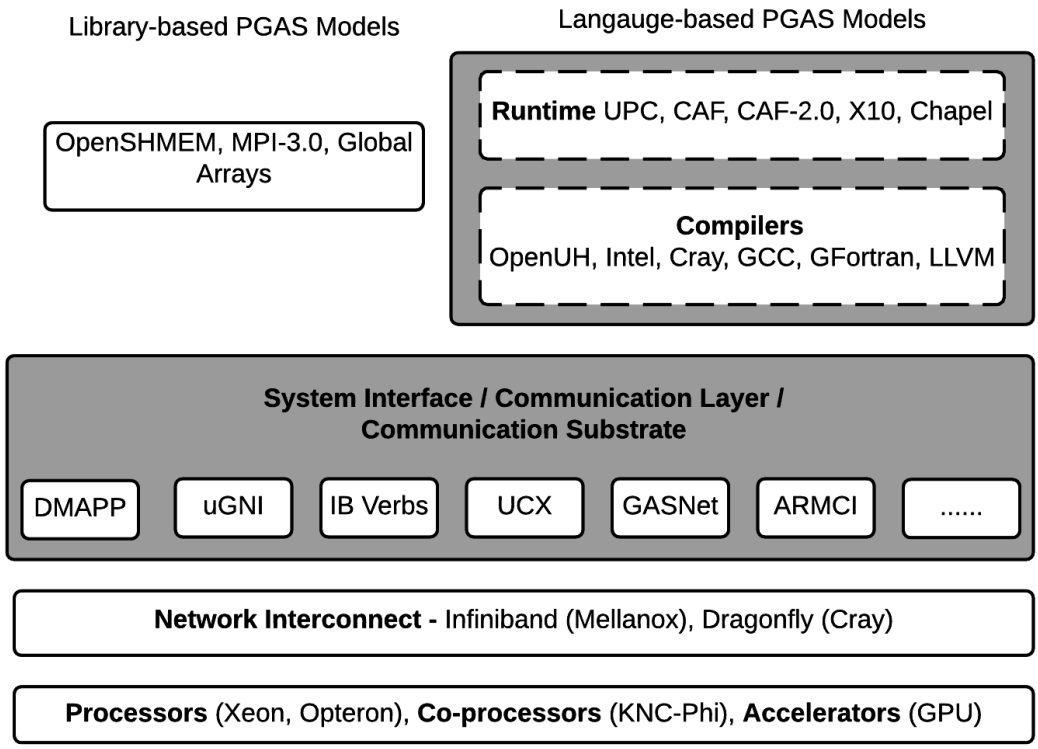

Figure 2.2: Overview of PGAS Programming Models

library-based models are highly portable. But, langauge-based models are completely on the other end of the spectrum, where they are dependent on the compilers and thus making it less portable.

All PGAS programming models are implemented over certain low-level communication layers, like PAMI, GASNet, UCCS, ARMCI, IB verbs, DMAPP, uGNI etc., which helps in communicating with the network. These communication layers are optimized towards some particular architecture. For example PAMI [23], is optimized

towards IBM Blue Gene/Q Supercomputers, while DMAPP [39] is in particular towards Cray XC machines. When language-based PGAS models are implemented over these communication layers they become highly optimized towards a particular architecture, but the optimizations in the runtime level in different available implementations are lost.

The most suitable solution for this problem is to make use of a common communication layer. But, creating a common communication layer towards different architectures will take a lot of effort and time. Projects like Libfabric [24], and UCX [37] tries to achieve this goal and they are still a work in-progress. The other feasible solution is to use the available library-based model as a common communication layer. The authors [42] show the usage of MPI-3.0 as a common communication layer for CAF. In this work, we analyze the usage of OpenSHMEM as a common communication layer for implementing CAF.

2.2 Coarray Fortran

CAF is a parallel subset for Fortran, which follows PGAS programming model. As SPMD model, all the execution units which are called as images are launched at the start of the program. The number of images remains unchanged throughout the execution and each image can access the data from other image without actually involving the source image through one-sided communication. CAF consists of various mechanisms for facilitating parallel processing, including: declaration of or dynamic allocation and deallocation of remotely-accessible data objects, one-sided

```
integer :: i
integer :: np
integer :: me
integer, allocatable :: &
         coarray_x(:)[:]
integer, allocatable :: &
         coarray_y(:)[:]

allocate ( coarray_x (4)[*] )
allocate ( coarray_y (4)[*] )



np = num_images()
me = this_image()


do i = 1, 4
  coarray_x = me
  coarray_y = 0
end do

coarray_y(2) = coarray_x(3)[4]


sync all
```

```
int i;
int np;
int me;
int *coarray_x;
int *coarray_y;


start_pes(0);
coarray_x = shmalloc(4 * \
       sizeof(int));
coarray_y = shmalloc(4 * \
       sizeof(int));

np = my_pe();
me = num_pes();


for (i = 0; i < 4; i++) {
  coarray_x[i] = me;
  coarray_y[i] = 0;
}

shmem_int_get(&coarray_y[1],
   &coarray_x[2], 1, 3);

shmem_barrier_all();
```

Figure 2.3: An OpenSHMEM variant of a CAF code

communication, remote atomic operations, barriers, point-to-point synchronization and image control mechanisms through locks and critical sections. In the following paragraphs, we will discuss the various features of CAF in detail. The left-hand side of Figure 2.3 depicts an example of a CAF program and it can be referred to understand the various features of CAF in detail.

2.2.1 Representation and Execution

Each image is represented by an unique image index and the total number of images can be obtained from `num_images()`, while the image index can be obtained from `this_image()` construct. The image index ranges from 1 to the number of images. Accessing other images or the remote memory access can be achieved through the use of co-indexed coarray references which are represented with square-brackets ([ ]).

2.2.2 Types of Data Objects

Data object belonging to an image may be remotely accessible or pure local objects (only accessible by the source/local image). Remotely accessible data objects in CAF may be further divided into two categories - symmetric and non-symmetric. Symmetric, remotely accessible objects include coarrays and subobjects of coarrays. Non-symmetric, remotely accessible objects include allocatable components of coarrays of a derived type, and objects with the target attribute which are associated with a pointer component of a coarray of derived type.

2.2.3 Data Communication

As discussed before, the remotely accessible objects in CAF can be accessed by a non-local image and this type of communication are done without the involvement of the local image in the form of one-sided communication. The data communication is facilitated by either remote get or put operations. A get operation is one where, the

local image receives data from the remote image and a put operation is one where the data from the local image is sent to the remote image object. The communication may be on a contiguous or a strided set of data. In Figure 2.3, the statement

```
coarray_y(2) = coarray_x(3)[4]
```

is an example for get operation, where the data from image 4 is being obtained by the local image. While, the operation like

```
coarray_x (1)[4] = coarray_y (2)
```

is an example for put operation, where the data is being sent from the local image to image 4.

2.2.4 Atomic Operations

Variables of kind integer or logical type which respectively have the atomic integer kind or atomic logical kind attribute can be referred to as Atomic variables. At present, Fortran 2008 defines two atomic subroutines: `atomic_define` and `atomic_ref`. As the name suggests, `atomic_define` is used to define and `atomic_ref` is used to refer to an atomic variable. The variable may not be accessed by another atomic subroutine while it is being defined or referenced.

2.2.5 Synchronization, Locks and Critical section

Synchronization and image control are done through the usage of `sync all`, `lock`, `unlock`, and `critical`, `end critical` operations. `sync all` provides a global barrier across all the images.

Locking operation makes sure when a coarray variable is being locked by one image, the access by other images are restricted and locks are obtained and released independently of corresponding locks of other images. The critical and end critical statements are used to define a critical section in the program which must be executed by only one image at any time.

2.2.6 Survey on CAF Implementations

In recent years, different Fortran compilers started to support their own CAF implementation. Several commercial and open-source projects started to support CAF in the compilers.

CAF implementation in the state-of-art Cray compiler is referred to as Cray-CAF in this thesis. Cray-CAF is one of the most advanced and highly optimized commercial implementation. It uses Cray's DMAPP [39] API, as its underlying communication layer and being optimized for both the Cray Aries and Gemini platform.

Intel implementation of CAF is referred to as Intel-CAF and it is one of the evolving implementations. It use MPI as its underlying communication layer and has a major advantage of usability in both Linux and Windows platforms. Intel-CAF is the only available commercial implementation which can be used in windows platforms.

UHCAF [16] is the implementation in OpenUH open-source compiler. UHCAF project has been evolved over the years and can be considered as a highly portable implementation with support across different platforms using their support to different

communication layers like GASNet [10], ARMCI [31] and SHMEM [6]. OpenSHMEM is a standard for different SHMEM implementations and based on the environment, we can configure the UHCAF to that particular OpenSHMEM implementation. For example, on InfiniBand Clusters we can make use of MVAPICH-2X OpenSHMEM implementation and on Cray Aries systems we can make use of Cray SHMEM implementation.

OpenCoarrays [19], is an open-source project which uses the GNU Fortran Compiler and can be considered as an evolving CAF implementation with rich platform support using their GASNet and MPI communication layers.

Implementation	Compiler	DMAPP	ARMCI	GASNet	MPI	MVAPICH-2X
Cray-CAF	Cray	✓	×	×	×	×
Intel-CAF	Intel	×	×	×	✓	×
UHCAF	OpenUH	×	✓	✓	×	×
OpenCoarrays	GNU	×	×	✓	✓	×
OSU-CAF	OpenUH	×	×	×	×	✓

Table 2.1: Different CAF Implementations and their Underlying Communication Layers

OSU-CAF [25], project makes use of OpenUH CAF implementation and retargeted the GASNet communication layer to their MVAPICH-2X implementation. With their optimized collectives and the effecient usage of IB-verbs, OSU-CAF is an effecient implementation for InfiniBand clusters.

Table 2.1, provides a brief view on the different CAF implementations as well as their support for different underlying communication layers.

2.2.6.1 CAF 2.0

Coarray Fortran (CAF) 2.0 [28] is a project from Rice University. Rice's new design for Coarray Fortran, is an expressive set of coarray-based extensions to Fortran. Compared to the emerging Fortran 2008 standards, Rice's new coarray-based language extensions include other additional features like:

- process subsets known as teams, which support coarrays, collective communication, and relative indexing of process images for pair-wise operations,
- topologies, which augment teams with a logical communication structure,
- dynamic allocation/deallocation of coarrays and other shared data,
- team-based coarray allocation and deallocation,
- global pointers in support of dynamic data structures, and
- enhanced support for synchronization for fine control over program execution,

To achieve portability with the Rice design, the compiler performs a source-to-source translation from CAF to Fortran 90 with calls to the CAF 2.0 runtime library primitives. Since, this design does not follow the regular CAF standard design, we did not include CAF 2.0 implementation for our analysis.

2.3 OpenSHMEM

OpenSHMEM [15] is a library interface specification, with bindings for C, C++, and Fortran, that unifies various specifications of the SHMEM programming API and adheres to the PGAS programming model. It is designed with a chief aim of performance, exploiting support for Remote Direct Memory Access (RDMA) available in modern network interconnects and thereby allowing for highly efficient data transfers without incurring the software overhead that comes with message passing communication.The right hand side of Figure 2.3 depicts an example of the SHMEM variant for the same program shown on the left.

OpenSHMEM programs follow an SPMD-like execution model, where all processing elements (PEs) are launched at the beginning of the program; each PE executes the same code and the number of processes remains unchanged during execution. OpenSHMEM PEs are initialized when the `start_pes` function is called. We present in the following paragraphs the main features of OpenSHMEM and Figure 2.4 provides a brief overview of different features available in OpenSHMEM specification.

2.3.1 Remote Memory Access

OpenSHMEM supports the PGAS memory model where it proceeds by one-sided communications to transfer data between PEs. `shmem_put` is the function that copies contiguous data from a local object to a remote object on the destination PE. `shmem_get` copies contiguous data from a remote object on the destination PE to a local object.

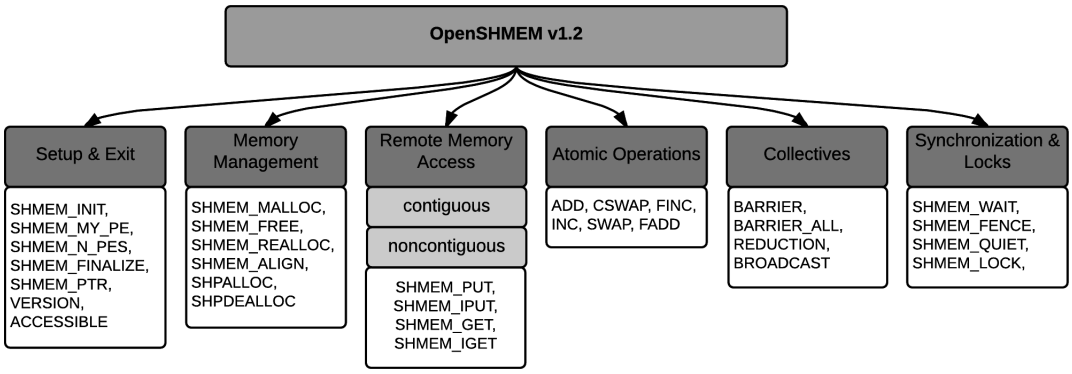

Figure 2.4: Brief Overview of OpenSHMEM API with both C and Fortran Bindings

2.3.2 Synchronization

We distinguish both types of synchronization: collective synchronization using for example the primitive `shmem_barrier_all`, that blocks until all other PEs issue a call to this particular statement, and point-to-point synchronization such as the `shmem_wait` primitive.

2.3.3 Collective Communication

One type of collective communications is reduction. `shmem_`*operator*`_to_all` is usually used to perform a reduction operation where *operator* can be `sum, and`, etc. on symmetric arrays over the active set of PEs. Symmetric arrays are remotely accessible data objects; an object is symmetric if it has a corresponding object with the same type, size and offset on all other PEs [6].

2.3.4 Mutual Exclusion

Mutual exclusion is implemented using locks of type integer. The `shmem_set_lock` function is used to test a lock and block if it is already acquired; the `shmem_set_unlock` function is used to release a lock.

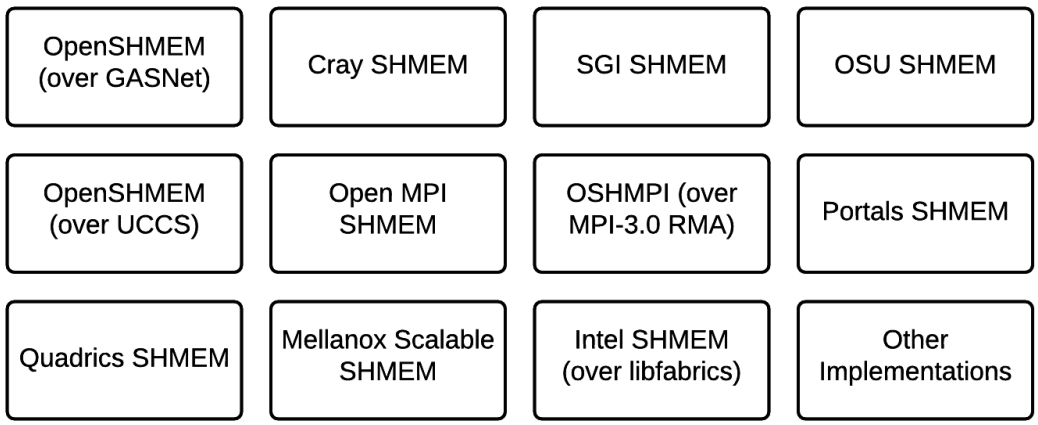

Figure 2.5: Different OpenSHMEM Implementations

2.3.5 Overview on OpenSHMEM Implementations

Historically, there have been multiple-vendor based implementations of SHMEM. OpenSHMEM [35] is an effort to create a standardized specification for the SHMEM interface. Figure 2.5, provides a brief image on the different available SHMEM implementations which follows the OpenSHMEM standards and [35] provides a detailed list of various available SHMEM libraries from different vendors including

Cray, SGI, Quadrics, and other open-source implementations based on GASNet and ARMCI. With various optimized vendor specific implementations, it has the potential to be used as an effective communication layer for other PGAS models.

2.3.6 OpenSHMEM Reference Implementation

OpenSHMEM implementation by University of Houston, which uses GASNet as a communication layer is considered as the reference implementation for the OpenSHMEM standards. OpenSHMEM reference implementation project is funded by OLCF and has been in development from 2012.

2.4 Architectural Overview

Figure 2.6, shows a general architectural overview of both OpenSHMEM and CAF runtime. The OpenSHMEM runtime calls are declared in the OpenSHMEM Library API layer, the API's declared in this layer are controlled by the OpenSHMEM specification. The declared API calls are defined in the OpenSHMEM runtime layer. The runtime layer is specific to implementations and system related optimizations like collective algorithms, effecient atomic operations, locking algorithms etc. The runtime layer can have one or more communication layers or system interfaces, to support the implementation over different architectures. The communication layers are specific to systems. Some common communication layers used are DMAPP, GASNet, UCCS, MV2X, and ARMCI.

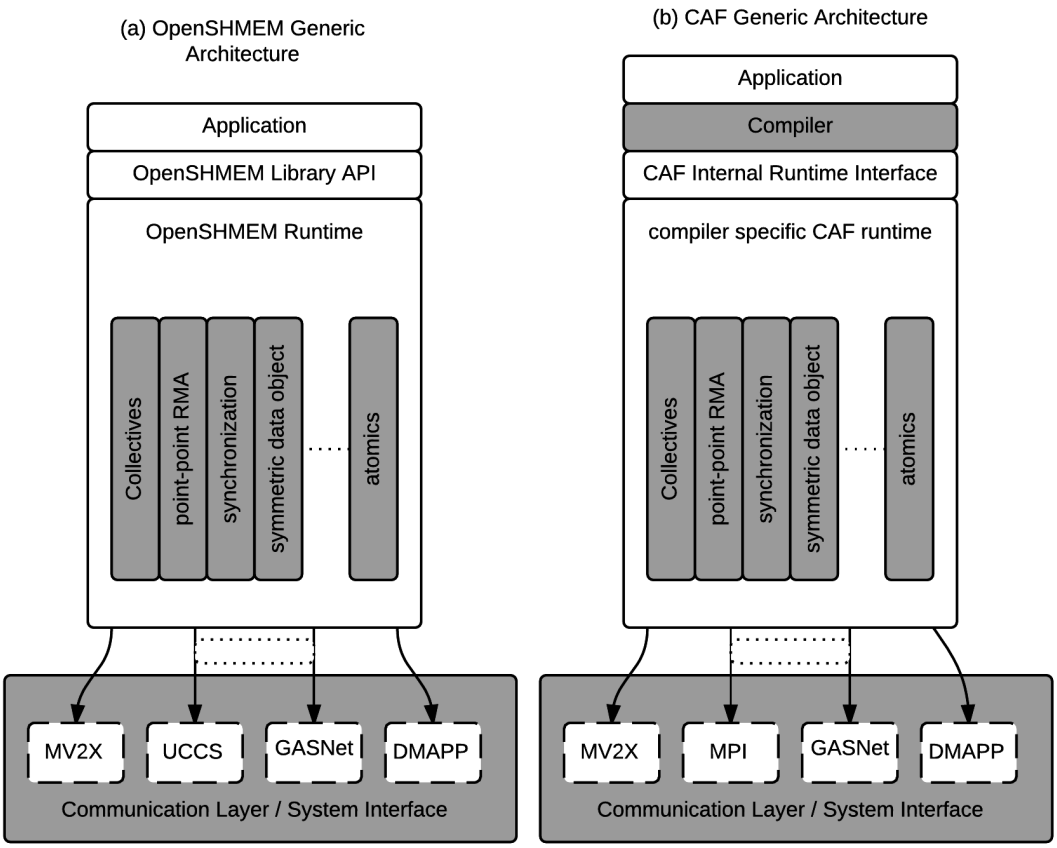

Figure 2.6: Overview of OpenSHMEM and CAF Architectures

Similar to the OpenSHMEM architecture, a brief overview on the CAF architecture shows the applications make use of the compiler semantics and the compiler in turn generate calls to the CAF internal runtime interface. The functionalities in the internal runtime interface are determined by the Fortran language standards. This in turn calls the compiler specific CAF runtime, which has all the compiler specific optimizations like collective algorithms, effecient atomic operations, locking algorithms etc., Similar, to OpenSHMEM this compiler specific CAF runtime makes use of one or more communication layers. As shown in Table 2.1, some common communication layers for CAF are GASNet, ARMCI, MPI-3.0(RMA) and DMAPP.

From Figure 2.6, we can determine that the architecture of both the OpenSHMEM and CAF runtime almost looks identical, with very few differences. With these similarities, we will further discuss on why OpenSHMEM should be used a communication layer for CAF.

Chapter 3

Relative-Performance Comparisons

In this chapter we will study the performance of OpenSHMEM in different environments for both inter- and intra-node communication against different available popular one-sided libraries.

3.1 OpenSHMEM for Inter-Node Communication

In this section, we motivate the use of OpenSHMEM as a portable communication layer for PGAS models for inter-node communication. The goal in this section is to identify the performance of OpenSHMEM implementations, namely Cray SHMEM and MVAPICH2-X SHMEM, against two other one-sided communication libraries which are candidates for supporting PGAS languages, GASNet and MPI-3.0. For these measures, we used two platforms, Stampede and Titan. The experimental setup and system configurations are shown in Table 3.1. For both systems, each

node allows execution with up to 16 cores per node.

Cluster	Nodes	Cores/Node	Processor Type	Interconnect
Stampede (TACC)	6,400	16	Intel Xeon E5 Sandy Bridge	InfiniBand Mellanox Switches/HCAs
Titan (OLCF)	18,688	16	AMD Opteron	Cray Gemini interconnect

Table 3.1: Experimental Setup and Machine configuration details for Relative Performance Comparisons of OpenSHMEM, MPI-3.0(RMA) and GASNet

We used microbenchmarks from the PGAS Microbenchmark Suite [2] to motivate our use of CAF over OpenSHMEM. We benchmarked point-to-point communication tests using `put` and `get` operations in OpenSHMEM, MPI-3.0(RMA) and GASNet. These tests are between pairs of PEs, where members of the same pair are always on two different nodes. The tests are performed using 1 and 16 pairs across two compute nodes, to assest communication cost with and without inter-node contention.

The latency performance results of `put` tests are shown in Figure 3.1. In Figure 3.1(a) and Figure 3.1(b), we have compared the OpenSHMEM implementation over MVAPICH2-X [5] against the GASNet and MPI-3.0 implementation over MVAPICH2-X on Stampede. In Figure 3.1(c) and Figure 3.1(d), we have compared the Cray SHMEM implementation against Cray MPICH and GASNet on the Cray XC30 platform.

In general, the latency of both GASNet and OpenSHMEM is less than the tested MPI-3.0 implementations when there is no contention (1 pair). For small data sizes until 2^{10} integers, both the performance of OpenSHMEM and GASNet are almost similar on Stampede, but Cray SHMEM performs better than GASNet on Titan. For

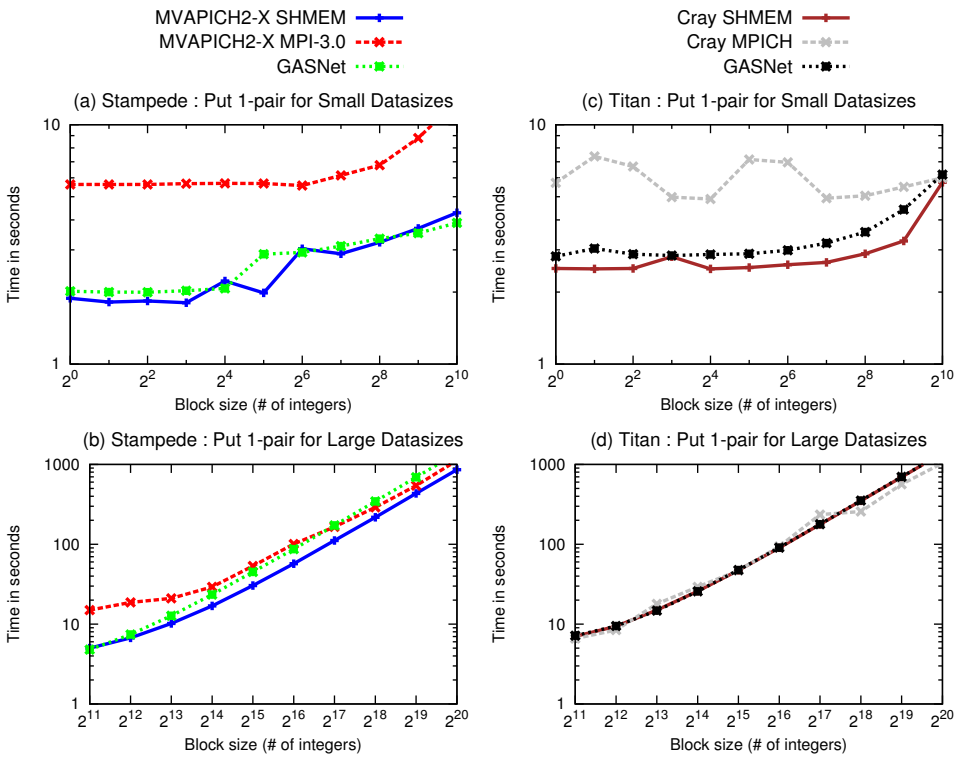

Figure 3.1: Put Latency comparison using two nodes (inter-node communication) for SHMEM, MPI-3.0 (RMA) and GASNet

large message sizes OpenSHMEM performs better than GASNet. With inter-node contention (16 pairs), SHMEM performs better than both GASNet and MPI-3.0 on Stampede, and has latency similar to GASNet on Titan.

The bandwidth performance results for `put` tests are shown in Figure 3.2. In Figure 3.2(a) and Figure 3.2(b), we have compared the MVAPICH2-X OpenSHMEM implementation [5] against the GASNet and MVAPICH2-X MPI-3.0 implementations on Stampede. In Figure 3.2(c) and Figure 3.2(d), we have compared the Cray SHMEM implementations against Cray MPICH and GASNet on Titan.

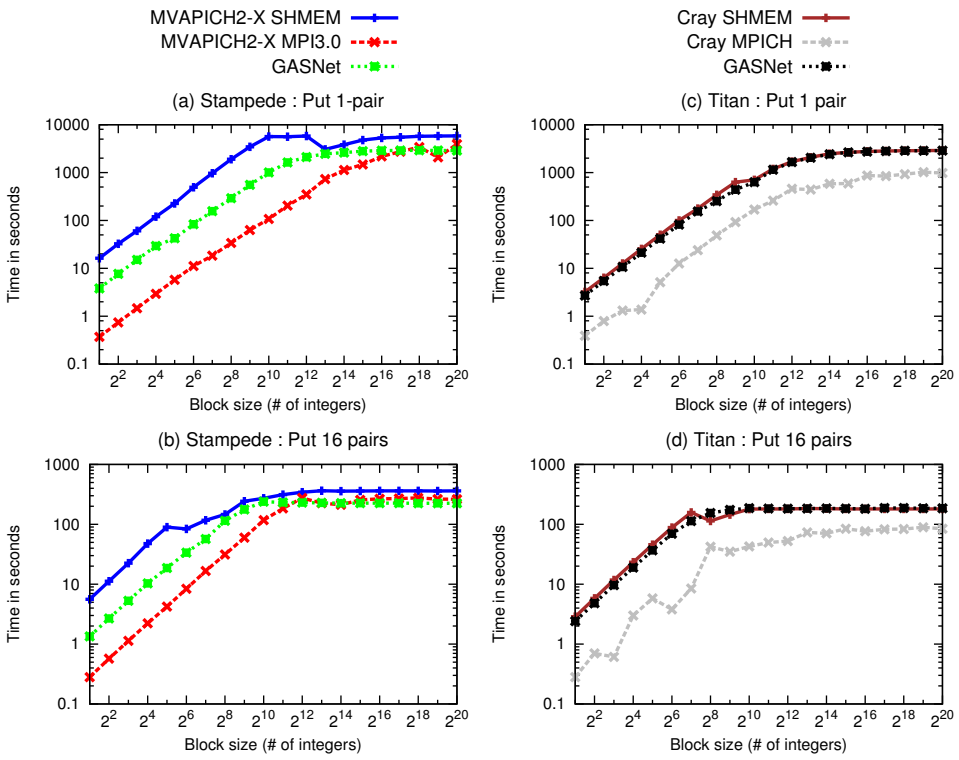

Figure 3.2: Put Bandwidth comparison using two nodes (inter-node communication) for SHMEM, MPI-3.0 (RMA) and GASNet

3.2 OpenSHMEM for Intra-Node Communication

In this section, we motivate the use of OpenSHMEM for intra-node communications. We made use of Intel Xeon Phi for analyzing these results. The many-core architectural features and the results of this analysis are discussed in the following paragraphs. One of the important contributions of our analysis includes the use of local load/store native operations using *shmem_ptr* to optimize OpenSHMEM performance for intra-node communications. We discuss about this optimizations in

Section 3.2.2

3.2.1 Intel Xeon Phi Architecture

Intel's popular Many Integrated Core (MIC) architectures are marketed under the name of Xeon Phi and they are the product of their KNC project. With the steady growth in requirements for the available number of computing cores in HPC, the popularity of Xeon Phi coprocessors among these computing-intensive applications has increased tremendously. GPUs are another example of accelerators that are able to accomplish such highly computing requirements. However, using GPUs, all the applications have to be ported to specific programming paradigms like CUDA, OpenCL, etc. Focusing on more abstract and generic approaches, we are using Xeon Phi which supports various popular programming models like MPI and OpenMP and which can also be targeted easily using OpenSHMEM library.

Xeon Phi provides x86 compatibility and runs on a Linux operating system. Intel Xeon Phi coprocessor consists of up to sixty-one (61) cores connected by a on-die bidirectional interconnect. In addition to the cores, there are 8 memory controllers supporting up to 16 GDDR5 (Graphics Double Data Rate, version 5) channels delivering up to 5.5 GT/s and special function devices such as the PCI Express system interface.

As shown in Fig. 3.3, each core includes a Core Ring Interface (CRI), interfacing the core and the ring interconnect. It comprises mainly the L2 cache and a distributed Tag Directory (TD) that ensures the coherence of this cache. Each core is

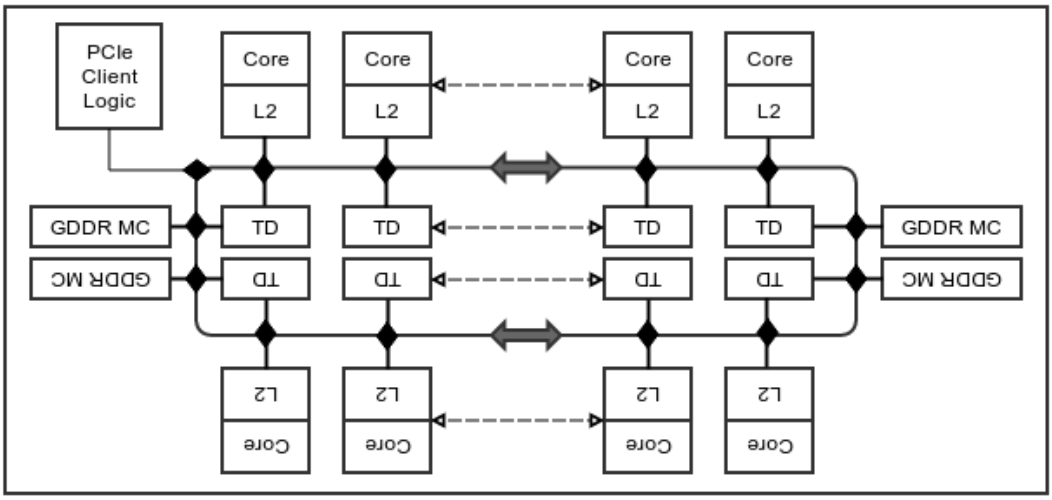

Figure 3.3: Xeon Phi Microarchitecture

connected to a bidirectional ring interconnect through the Core Ring Interface. The L1 and L2 cache have a data limit of 32KB and 512KB respectively. The three key aspects of Xeon Phi are its support of 512-bit vector instructions (each core includes a 512 bit-wide vector processor unit (VPU)), simultaneous multi-threading and in-order execution, which offers some ways to exploit Instruction Level Parallelism and introduces some caveats not otherwise encountered in other X86-based architectures.

For reasons mentioned before, we will only focus on the *native* mode of execution, wherein an application runs exclusively on the Xeon Phi co-processor. The native mode of Xeon Phi offers some benefits such as minimal code-porting overhead from existing architectures and not having to deal with host-to-co-processor data transfer latency (which is at least 15x slower than Xeon Phi intra-node latency). Although

one could achieve modest performance by porting existing CPU code on Xeon Phi, according to our observation it requires reasonable optimization effort to exploit its capabilities fully. Typically, the available memory varies, but it is within 8GB in the current architecture. As such, the number of active threads ($4 \times 61 = 244$) could saturate the memory, and being an in-order processor, this could result in excessive stalls unless memory is prefetched into the caches. Also, a Xeon Phi core could issue 1 or 2 instructions per cycle (the cores are typically clocked at 1 GHz), and only one of them could be a vector instruction such as division; prefetch instructions such as `VPREFETCH1` are not considered vector instructions. On the other hand, a thread can only issue vector instructions in every other cycle, which mandates the use of at least 2 threads/core to fully utilize a vector unit.

3.2.2 From Remote Memory Access to Local Load/Store Operations

This section describes one of our major contributions, namely how transforming remote memory accesses (`shmem_put/shmem_get`) into local load/store native operations using `shmem_ptr` can greatly improve OpenSHMEM performance.One of the most desirable features of the shared memory programming environment is accessing data via local load / store operations, which makes programming simpler and more efficient. This approach makes it possible for the compiler to analyze and perform essential optimizations for Xeon Phi. To enable such local load and store operations in OpenSHMEM, we suggest to employ the `shmem_ptr` builtin that returns the address

of a data object on a specific PE.

On a shared-memory machine, it is beneficial to use it as opposed to OpenSHMEM communication primitives such as `shmem_put` or `shmem_get`. Apart from saving the function call overhead which is nominal, `shmem_ptr` could be effectively employed to enable optimizations.

Our method consists of: (1) computing the memory address of a symmetric variable on a given PE only once at the beginning of the program and, (2) performing the translation of communication function calls to simple assignments. A simple example that illustrates the use of this methodology is provided in Figure 3.4.

```
shmem_int_put(target,
       source,m,pe);
```

```
int *ptr=(int *)shmem_ptr
           (target,pe);
for (i = 0; i < m; i+=1)
  ptr[i] = source[i];
```

Figure 3.4: The use of `shmem_ptr`

This yields significantly better performance in terms of latency as demonstrated in Figure 3.5. Moreover, Figure 3.6 and Figure 3.7 show the impact of this methodology on the memory bandwidth. When the message size is very large (block size $> 16K$), the difference in memory bandwidth between local load/store operations and remote access functions becomes small; this is due to the increasing number of cache misses. These results are obtained running the PGAS-Microbenchmarks [2] that are well suited for evaluating performance for both point-to-point operations and reductions in various PGAS implementations. For n pairs, the number of running PEs is $n \times 2$ on an Intel Xeon Phi in native mode. Note that the OpenSHMEM reference

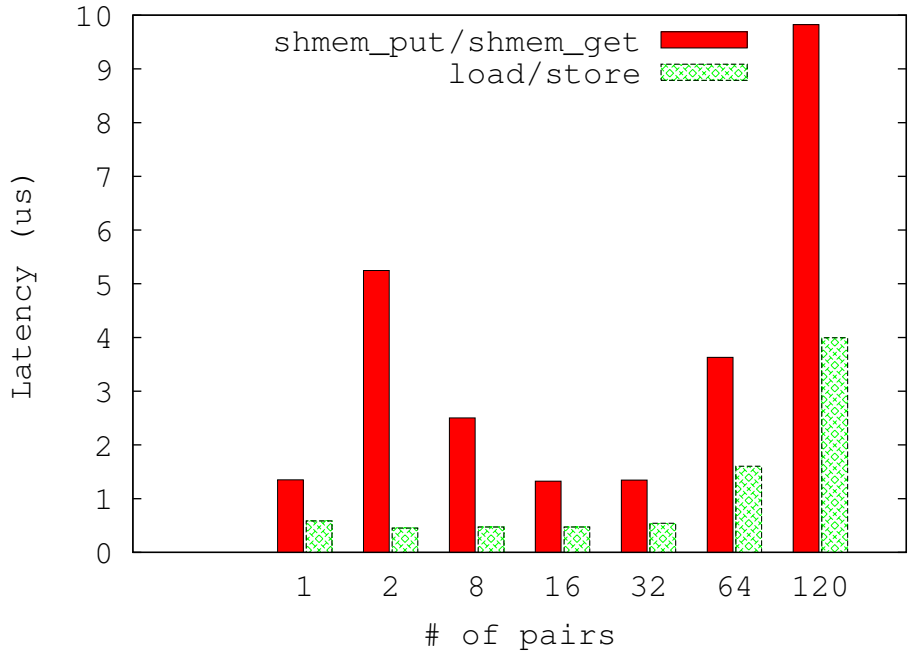

Figure 3.5: Latency comparison put/get vs. `shmem_ptr`

implementation was modified to support `shmem_ptr`.

3.2.3 Introducing Different Reduction Algorithms

We describe here four algorithms for parallel array reduction, and describe their performance characteristics using a performance model for the Xeon Phi architecture. We chose reduction as a representative collective operation because of its wide usage in many applications, and the fact that there is some computation associated with it (not only communication as in scatter/gather). It is possible for an architecture like Intel Xeon Phi to utilize vector pipelines to overlap computation and communication.

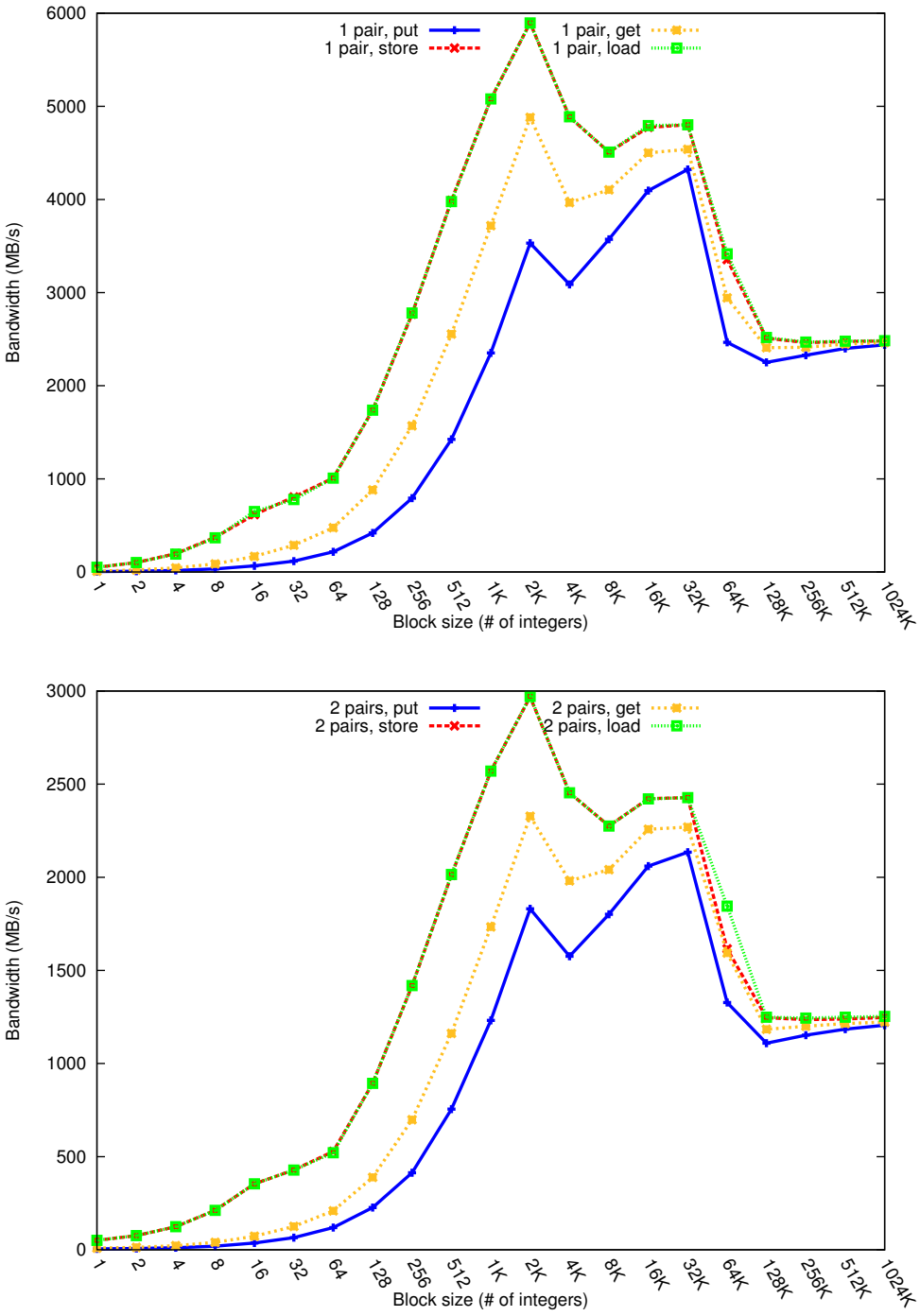

Figure 3.6: Bandwidth comparison of communications between 1 and 2 pairs of PEs: load/store using shmem_ptr vs calls to shmem_put/shmem_get; note that green and red lines overlap

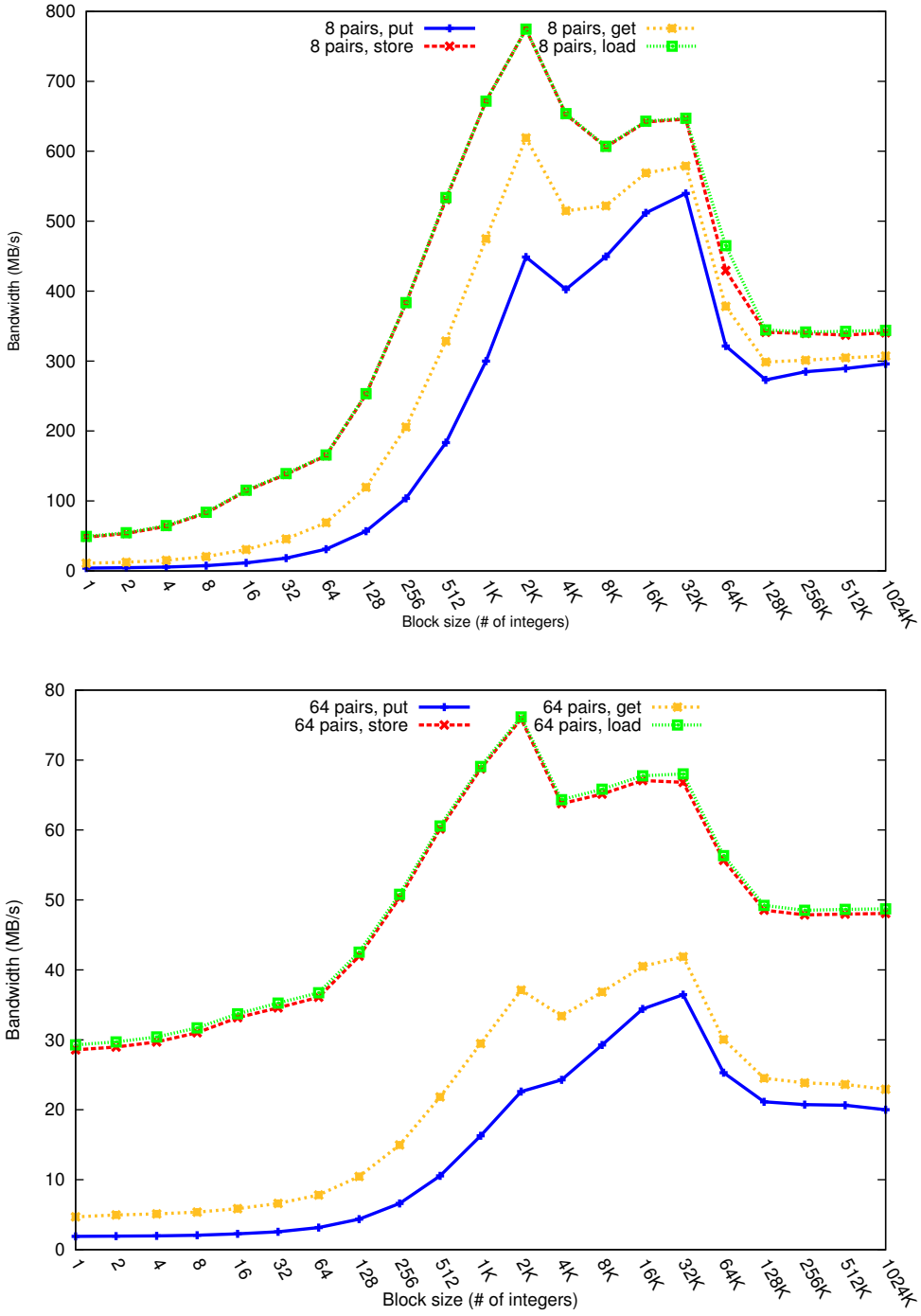

Figure 3.7: Bandwidth comparison of communications between 8 and 64 pairs of PEs: load/store using shmem_ptr vs calls to shmem_put/shmem_get; note that green and red lines overlap

In the following, we describe each of these algorithms and our reasoning for why we considered them for Xeon Phi. We have looked at both power-of-2 cases and non-power-of-2 cases for the number of PEs in our evaluations.

We use a simple cost model to estimate the total time taken by these algorithms based on three parameters: latency α, bandwidth β, and local computation cost per byte γ. We denote n the size of arrays and p the number of processes. This cost model assumes that all processes can send and receive one message at the same time.

3.2.4 Flat Tree (FT)

The flat-tree or linear algorithm uses a single root process that communicates with the remaining processes for carrying out the all-reduce operation, and it is considered to work well for arrays of short to medium size. This algorithm is carried out in two stages. In the first stage, the root process will gather the data from each of the other processes and apply the reduction operation on them. In the second stage, the results are broadcasted back to the other processes. Different strategies may be employed for gathering the data and then performing the subsequent broadcast, depending on factors such as array size and total number of processes involved in the reduction. We consider here the simplest strategy, where the root process will sequentially read the source buffers from each of the other processes and compute the reduced result, and then write the result back to each of the other processes sequentially.

The time taken in this algorithm is:

$$T_{FT} \quad = \quad p\alpha + 2n(p-1)\beta + n(p-1)\ \gamma.$$

3.2.5 Recursive Doubling (RD)

The recursive (distance) doubling algorithm [40] is a technique that can be applied for a variety of collective algorithms, including all-gather, barrier, and all-reduce. For cases where the number of processes performing the reduction is a power of 2, the algorithm completes in $\lg p$ steps. On the first step ($s = 0$), processes with even rank i will exchange their initial array values with the process of rank $i + 1$, and then all processes will perform the reduction operation using the received data. On each subsequent step s, each process i will exchange its updated results with either process $i + 2^s$ if $\text{mod}(i, 2^{s+1}) < 2^s$ or with process $i - 2^s$ if $\text{mod}(i, 2^{s+1}) \geq 2^s$, and then again perform the reduction operation with the received data.

When the number of processes is not a power of 2, each of the first r even ranked processes with rank i (where r is $p - 2^{\lfloor \lg p \rfloor}$) will send its data to the process with rank $i + 1$. These receiving processes will perform the reduction operation using the received data. We then have a set of $2^{\lfloor \lg p \rfloor}$ processes to carry out the core reduction algorithm that requires a power of 2 number of participants, namely the first r processes with odd rank and then the last $2^{\lfloor \lg p \rfloor} - r$ processes. After the core algorithm is completed among these processes, they will each contain the final reduction result. We complete the algorithm by having the first r odd ranked processes with rank i send their data to processes with rank $i - 1$.

The time taken in this algorithm is:

$$T_{RD} \;=\; \lg p\ \alpha + n \lg p\ \beta + n \lg p\ \gamma.$$

In order to optimize this algorithm for native mode execution on Xeon Phi, we

used the `shmem_ptr` routine which allows each PE to obtain a direct access to another PE's symmetric data. We also experimented with a "left computes" strategy, where for each pair of communicating processes the one with smaller rank (1) updates its local array by applying the reduction operation and using a direct reference to the array on its partner, (2) writes the resulting values back to the partner's array, and (3) notifies the partner that its array has been updated. This strategy allowed us to avoid the use of an additional receive buffer and `memcpy` operations for the data exchanges, and improve performance significantly for large array sizes.

3.2.6 Bruck (BR)

We have developed a variant of the Bruck all-gather algorithm [11] for reductions, which we will refer to here for simplicity as Bruck. As in the recursive doubling algorithm described above, the core algorithm described here works for a power of 2 number of processes, and we deal with the case where it is not a power of 2 as described above. For Bruck, on each step s a process with rank i will send its data to the process with rank $\mathrm{mod}(2^{\lfloor \lg p \rfloor} + i - 2^s, 2^{\lfloor \lg p \rfloor})$. Upon receipt of this data, the process will perform a reduction using it. After $\lg p$ steps where p is a power of 2, all p processes performing the reduction will contain the final result.

The time taken in this algorithm is:

$$T_{BR} = \lg p\ \alpha + n \lg p\ \beta + n \lg p\ \gamma.$$

3.2.7 Rabenseifner (RSAG)

We also implemented Rabenseifner's algorithms [40] in order to more efficiently carry out reductions on very large array sizes. As with recursive doubling, the core algorithm works for a power of 2 number of processes, and when the number of processes is not a power of 2 it can be handled as described in 3.2.5. The algorithm can be implemented in two phases. In the first phase, the PEs perform a Reduce-Scatter (RS) operation, using a distance doubling and vector halving procedure which completes in a $\lfloor \lg p \rfloor$ steps. In the second phase, the PEs perform an All-Gather (AG) operation, using a distance halving and vector doubling procedure which can again complete in $\lfloor \lg p \rfloor$ steps. While there is a greater number of messages being communicated in this algorithm among the PEs over the two phases, because on each step only a portion of the full array is being exchanged between the PEs this turns out to be more efficient for reductions on large array sizes.

The time taken in this algorithm is:

$$T_{RSAG} \;=\; 2 \lg p \; \alpha + 2\tfrac{p-1}{p} n \; \beta + \tfrac{p-1}{p} n \; \gamma.$$

For native-mode execution on Xeon Phi, we applied the following optimizations to improve the performance. We first combined the last step of the reduce-scatter phase with the first step of the all-gather phase. In effect, the processes that are paired with each other in the last step of reduce-scatter can exchange corresponding portions of their array data and perform the reduction operation using the received data. Thus, we can think of this algorithm as now divided into 3 stages: (1) the first $\lfloor \lg p \rfloor - 1$ steps of reduce-scatter, (2) an exchange and reduction of corresponding array blocks

between PEs which are a distance $2^{\lfloor \lg p \rfloor}/2$ apart, and (3) the final $\lfloor \lg p \rfloor - 1$ steps of all-gather. We also use `shmem_ptr` to avoid having to introduce an additional work buffer and perform `memcpy` operations. Instead, each PE can read directly from its partner's array to update its own array using the reduction operation. Finally, we also applied the "left computes" optimization described in 3.2.5 for the new middle stage of our algorithm.

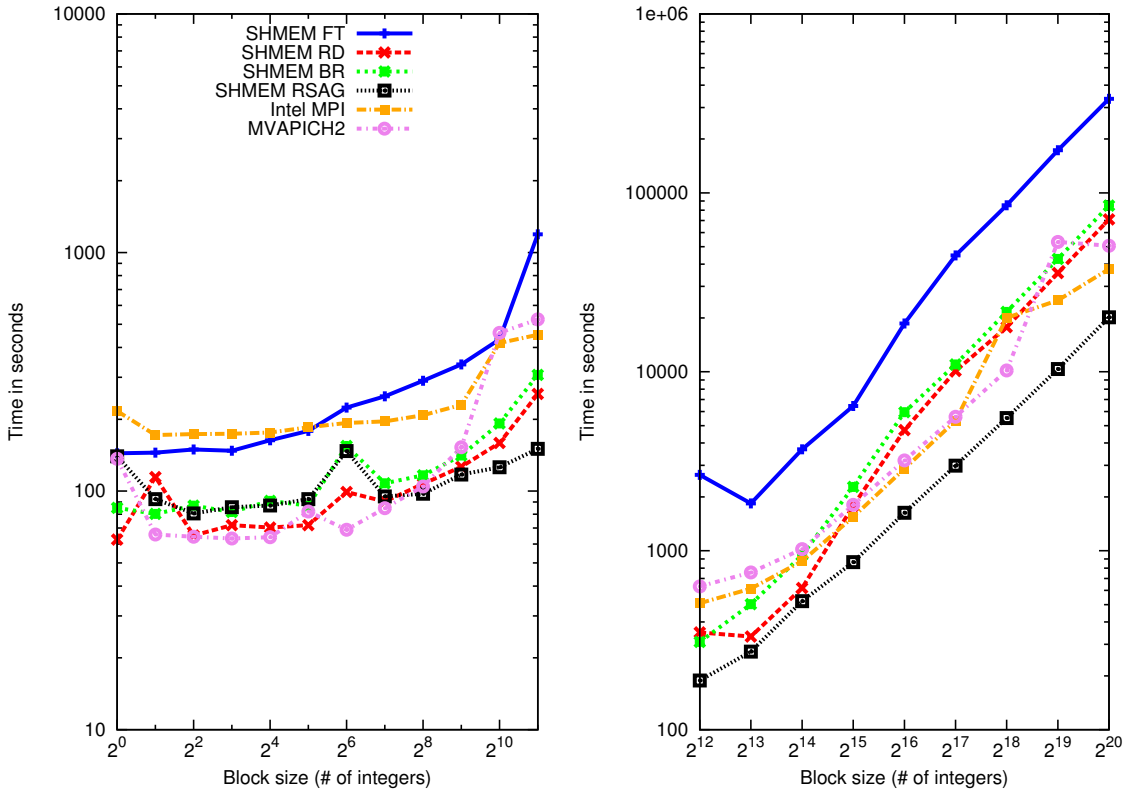

Figure 3.8: Performance of different reduction algorithms implemented in OpenSHMEM vs. MPI using 64 PEs on Intel Xeon Phi (1 to 2^{10} and 2^{12} to 2^{20} number of integers)

3.2.8 Performance Analysis of Different Reduction Algorithms

We implemented the aforementioned reduction algorithms in OpenSHMEM without using our `shmem_ptr` optimization (see next subsection), and tested the performance when running in native mode on Intel Xeon Phi using the reduction microbenchmarks in our PGAS-Microbenchmark suite. In Fig. 3.8, we found that the flat tree algorithm performed the worst, as predicted from our cost model.

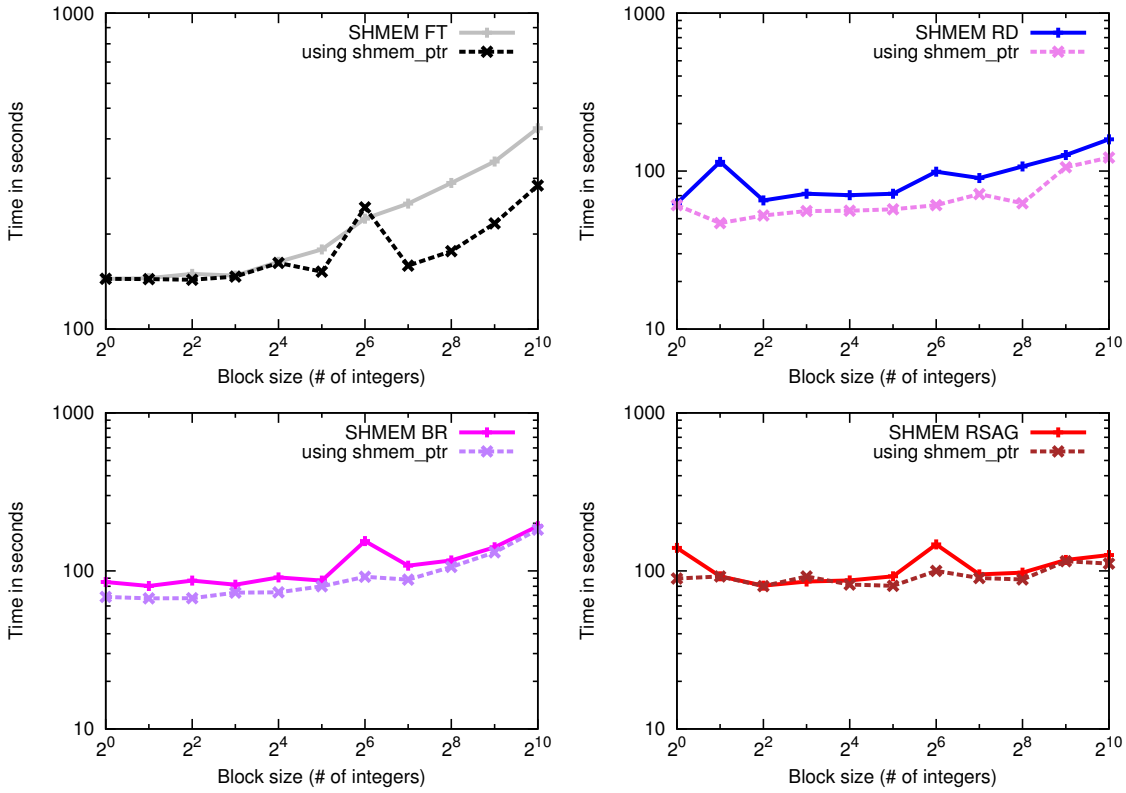

Figure 3.9: SHMEM reduction algorithm performance using put/get vs. `shmem_ptr` using 64 PEs for small data sizes

Moreover, we observe significant improvement when using recursive doubling for small message sizes (up to 4KB). Rabenseifner's algorithm performs the best for

large message sizes compared to the other existing algorithms. This is due to a lessened data transfer cost resulting from less congestion on the ring network of the Phi. However, for small message sizes, MVAPICH2 performs the best; this leads to the idea of using `shmem_ptr` in our SHMEM implementation of reduction algorithms, which we will be exploring in the next section.

3.2.9 Application of `shmem_ptr` on Reduction Algorithms

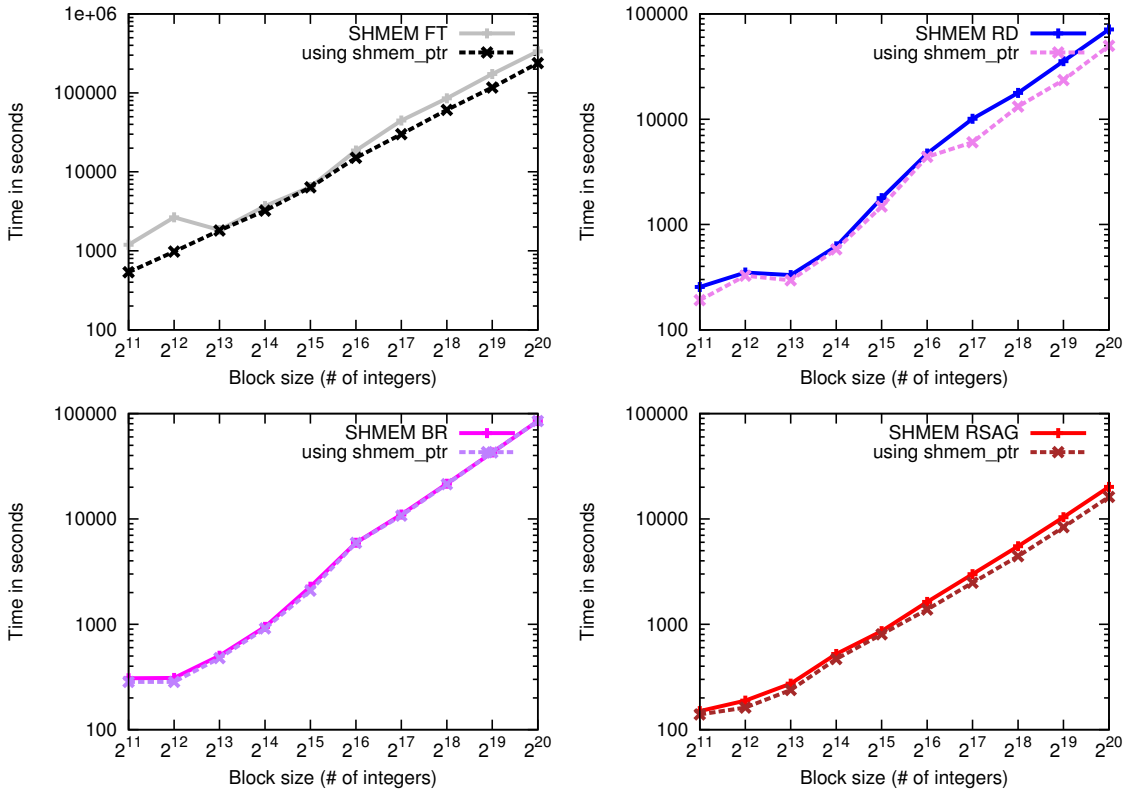

Figure 3.10: SHMEM reduction algorithm performance using put/get vs. `shmem_ptr` using 64 PEs for large data sizes

The use of `shmem_ptr` serves a dual purpose for running OpenSHMEM codes more

efficiently on Intel Xeon Phi. It allows us to eliminate the use of some intermediate buffers, which is important because there is a small amount of main memory of typically less than 8 GB. Furthermore, it exposes more instruction-level parallelism to the compiler, enabling vectorization opportunities which can take advantage of the Phi's 512-bit-wide vector units. Therefore, we have applied our method of using `shmem_ptr` in our implementation of the reduction algorithms presented in Section 3.2.3.

We ran the optimized version of these algorithms on Intel Xeon Phi using 64 PEs in native mode. The experimental results are shown for small message sizes in i Fig. 3.9 and for large message sizes in Fig. 3.10. We observe that the use of `shmem_ptr` improves the execution time of these reduction operations. Especially, in Fig. 3.11, the optimized versions of the recursive doubling and RSAG algorithms perform better than both Intel MPI's and MVAPICH2's default implementations for small and large message sizes respectively.

3.3 Why OpenSHMEM?

From Section 2.4, we can see that both OpenSHMEM and CAF has similar architectures. And, the performance results by relative comparisons with GASNet and MPI-3.0(RMA) shows that the OpenSHMEM has optimized performance for both inter- and intra-node communications on two different systems- Cray system and InfiniBand cluster for inter-node communication and Intel Xeon Phi multicore arechitecture for intra-node communication. Our work [30], discusses further about the usage of `shmem_ptr` for optimizing intra-node communications in OpenSHMEM.

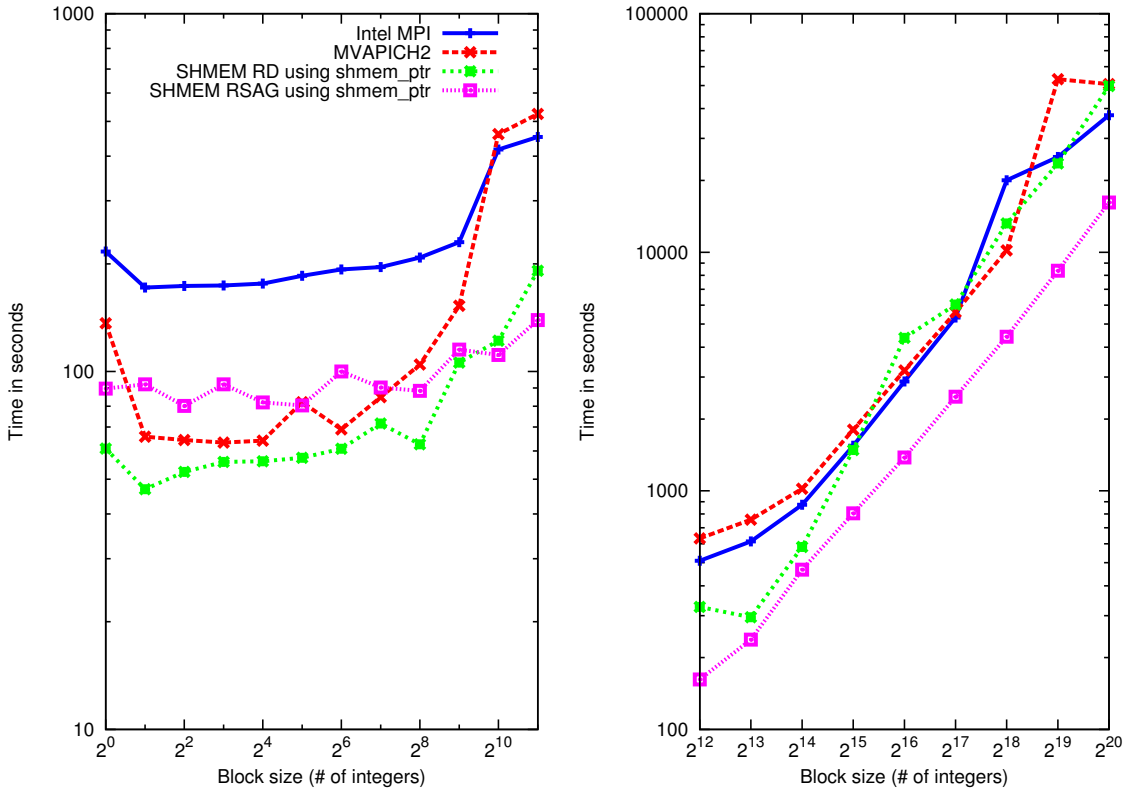

Figure 3.11: Performance of different reduction algorithms using `shmem_ptr` vs. Intel MPI and MVAPICH using 64 PEs on Intel Xeon Phi

Apart from the performance results using PGAS Microbenchmarks, [30] shows the performance gains in NAS Parallel Benchmarks [9] [4].

Based on the TOP-500 [7] list, we can see that Cray shares 14.2% of the total systems and around 40% in terms of performance. Similarly, InfiniBand shares around 51.8% of systems and 40% in terms of performance. This provides sufficient motivation, as OpenSHMEM performs better than the other available counterparts in these two systems. Based, on this analysis we have ported the CAF implementation in OpenUH compiler to OpenSHMEM. In further Chapters, we provide insights on the

implementation details and performance analysis on the CAF implementation over OpenSHMEM.

Chapter 4

Implementation

4.1 CAF and OpenSHMEM Comparison

Table 4.1 presents a comparative study of various available features in CAF and OpenSHMEM. Symmetric data allocations, remote memory data accesses, atomic operations, and collectives [1] are available in both models. So, these features available in OpenSHMEM may be used directly to implement the corresponding CAF operations. However, features which are unavailable in OpenSHMEM, such as multi-dimensional strided data communications and remote locks, have to be effectively implemented for CAF. In this section, we present the design and implementation of our translation.

[1] In UHCAF, we implement CAF reductions and broadcasts using 1-sided communication and remote atomics available in OpenSHMEM.

Properties	CAF	OpenSHMEM
Symmetric data allocation	`allocate`	`shmalloc`
Total image count	`num_images()`	`num_pes()`
Current image ID	`this_image()`	`my_pe()`
Collectives - reduction	`co_`*operator*	`shmem_`*operator*`_to_all`
Collectives - broadcast	`co_broadcast`	`shmem_broadcast`
Barrier Synchronization	`sync all`	`shmem_barrier_all`
Atomic swapping	`atomic_cas`	`shmem_swap`
Atomic addition	`atomic_fetch_add`	`shmem_add`
Atomic AND operation	`atomic_fetch_and`	`shmem_and`
Atomic OR operation	`atomic_or`	`shmem_or`
Atomic XOR operation	`atomic_xor`	`shmem_xor`
Remote memory put operation	[]	`shmem_put()`
Remote memory get operation	[]	`shmem_get()`
Single dimensional strided put	[] to strided data	`shmem_iput(,stride,)`
Single dimensional strided get	[] from strided data	`shmem_iget(,stride,)`
Multi dimensional strided put	[]	×
Multi dimensional strided get	[]	×
Critical Section	*critical*	×
Remote Locks	*lock*	×

Table 4.1: Summary of features for supporting parallel execution in Coarray Fortran (CAF) and OpenSHMEM

4.2 Symmetric Memory Allocation

Language features in CAF for supporting parallel execution are similar to the features provided by OpenSHMEM. Consider Figure 2.3 which shows the correspondence between CAF and OpenSHMEM constructs. Coarrays (which may be `save` or `allocatable`) are the mechanism for making data remotely accessible in a CAF program; the corresponding mechanism in OpenSHMEM is to declare the data either

as *static* or global or to explicitly allocate it using the *shmalloc* routine. Hence, remotely accessible data in CAF programs may be easily handled using OpenSHMEM. A `save` coarray will be automatically remotely accessible in OpenSHMEM, and we can implement the `allocate` and `deallocate` operations using `shmalloc` and `shfree` routines, respectively. Additionally, coarrays of derived type may be used to allow non-symmetric data to be remotely accessible. To support this with OpenSHMEM, we *shmalloc* a buffer of equal size on all PEs at the beginning of the program, and explicitly manage non-symmetric, but remotely accessible, data allocations out of this buffer. The CAF `this_image` and `num_images` intrinsics map directly to the `my_pe` and `num_pes` routines in OpenSHMEM.

4.3 Remote Memory Access (RMA) - Contiguous Data Communication

Remote memory accesses of contiguous data in OpenSHMEM can be performed using the `shmem_putmem` and `shmem_getmem` function calls, and they can be used to implement the remote memory accesses in CAF. Though the properties of RMA in CAF and OpenSHMEM are relatively similar, they are not exactly matching. CAF ensures that accesses to the same location and from the same image will be completed in order; OpenSHMEM has stronger semantics, allowing remote writes to complete out of order with respect to other remote accesses.

In the example shown in Figure 4.1, the initial values of *coarray_x* (which is 3) will

be written to *coarray_y* at image 2, achieved using a `shmem_putmem` call. The subsequent modification of *coarray_x* to the value 0 does not require additional synchronization, since `shmem_putmem` ensures local completion. However, the put operation from data from *coarray_b* to *coarray_a* needs an additional `shmem_quiet` function to ensure the remote completion of this transfer. Otherwise, the next transfer statement from *coarray_a* to *coarray_c* would become erroneous in OpenSHMEM.

```
coarray_x(:) = 3
coarray_y(:)[2] = coarray_x(:)
coarray_x(:) = 0

coarray_a(:)[2] = coarray_b(:)
coarray_c(:) = coarray_a(:)[2]
```

Figure 4.1: Example code on CAF RMA local and remote completion

Therefore, in our translation, we insert `shmem_quiet` after each call to `shmem_put` and before each call to `shmem_get` to handle remote memory transfers in CAF.

4.4 Remote Memory Access (RMA) - Strided Data Communication

Strided remote memory accesses (RMA) in OpenSHMEM are available; however, the API presumes accesses of 1-dimensional array buffers with a single specified stride. Hammond [20] proposes a solution for two-dimensional (matrix oriented) strided data communication as an extension to OpenSHMEM using the DMAPP interface; however, to the best of our knowledge, there are no efficient proposals for

multi-dimensional strided communications. In this section, we introduce an efficient algorithm for performing multi-dimensional strided remote accesses using the existing 1-dimensional strided routines (i.e., `shmem_iput` or `shmem_iget`).

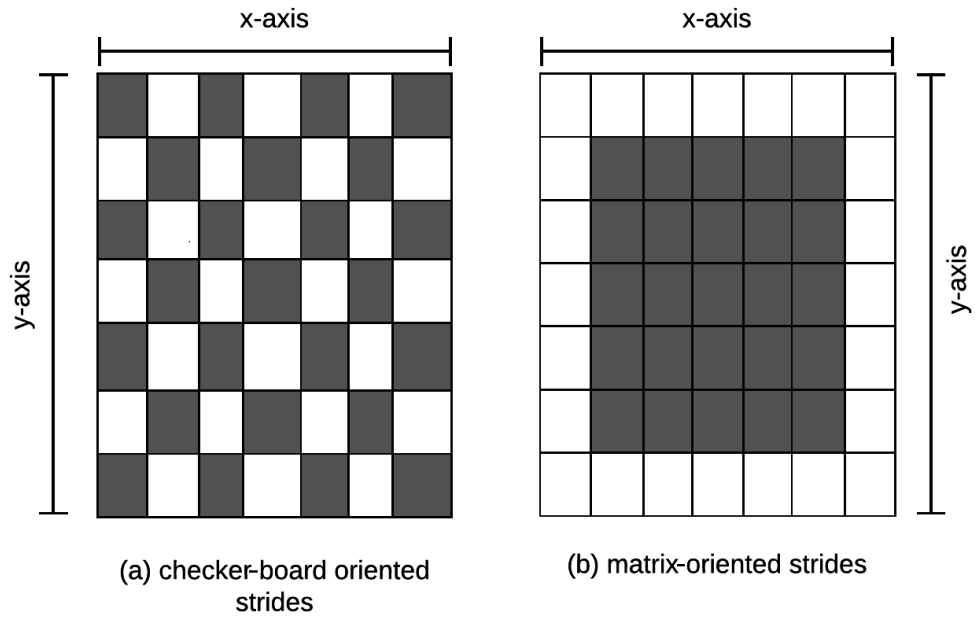

Figure 4.2: Different Multi-dimensional Stride Orientation

Figure 4.2 shows the two different orientations of strides in 2-dimensional array. All multi-dimensional strides can be represented using two-dimensional strides in general. Fortran as a language supports upto 7-dimensions. And, any muti-dimensional strides greater than 2-dimensions can be represented as replica of 2-dimensional arrays. Figure 4.2(a) shows the checker-board orientation and Figure 4.2(b) shows the matrix orientation. The checker board orientation cannot be represented using CAF.

We need to use at least two different strided calls to obtain checker-board orientation in CAF. The matrix orientation are the most common strided orientation used. As OpenSHMEM specification lack block-wise strided data transfer, the matrix oriented strides are impossible to be implemented using a single OpenSHMEM call. We have developed two different algorithms as shown in Algorithm 1 and Algorithm 2 to implemente multi-dimensional data transfer using OpenSHMEM.

The naïve way to implement multi-dimensional strides using OpenSHMEM is to use multiple `shmem_putmem` or `shmem_getmem` functions for each strided element on each dimension. For instance, consider a three dimensional array *coarray_X(100, 100, 100)*. When we perform a strided operation as *corray_X(1:100:2, 1:80:2, 1:100:4)*, we have 50, 40 and 25 strided elements in dimension 1, 2 and 3 respectively. Hence, we have a total of 50 * 40 * 25 `shmem_putmem` or `shmem_getmem` calls. Algorithm 1 shows the naïve way to implement multi-dimensional strides using multiple `shmem_putmem` or `shmem_getmem` function calls.

ALGORITHM 1: Baseline algorithm: use of `shmem_putmem`

```
procedure shmem_iputmem_using_putmem(void *dest_ptr, const void *src_ptr, ptrdiff_t
dest_stride, ptrdiff_t src_stride, size_t blksize, size_t nblks, int pe_id);
    char *temp_target = dest_ptr;
    char *temp_source = src_ptr;
    for (int i = 1; i ≤ nblks; i++) do
      shmem_putmem(temp_target, temp_source, blksize, pe_id);
      temp_target += dest_stride;
      temp_source += src_stride;
    end
end procedure
```

In our solution, we apply two optimizations: the first optimization is to identify the most suitable dimension for using the `shmem_iput` or `shmem_iget` strided calls and

use it as the base dimension `base_dim` in order to reduce the number of strided calls. Consider the same example as above where we have dimension 1 with 50 strided elements, dimension 2 with 40 and dimension 3 with 25 strided elements. Here, the base dimension would be dimension 1, because it has more strided elements than the other dimensions; the base dimension can be called through `shmem_iput` or `shmem_iget`. Hence, the number of calls will be reduced to 1 * 40 * 25 which is much better than the naïve solution.

In our solution, we apply two optimizations: the first optimization is to identify the most suitable dimension for using the `shmem_iput` or `shmem_iget` strided calls and use it as the base dimension `base_dim` in order to reduce the number of strided calls. Consider the same example as above where we have dimension 1 with 50 strided elements, dimension 2 with 40 and dimension 3 with 25 strided elements. Here, the base dimension would be dimension 1, because it has more strided elements than the other dimensions; the base dimension can be called through `shmem_iput` or `shmem_iget`. Hence, the number of calls will be reduced to 1 * 40 * 25 which is much better than the naïve solution.

Besides reducing the number of strided calls, we should consider the locality of data. If we consider the above example and we use the second dimension to be a base dimension, we have the strided length for each element to be of size 100 since the size of dimension 1 is 100.

In this case, we will obtain data from different cache levels. The tradeoff between reducing the number of calls and considering the data locality is considered in our approach by applying the `base_dim` calculation to only the two first dimensions. The

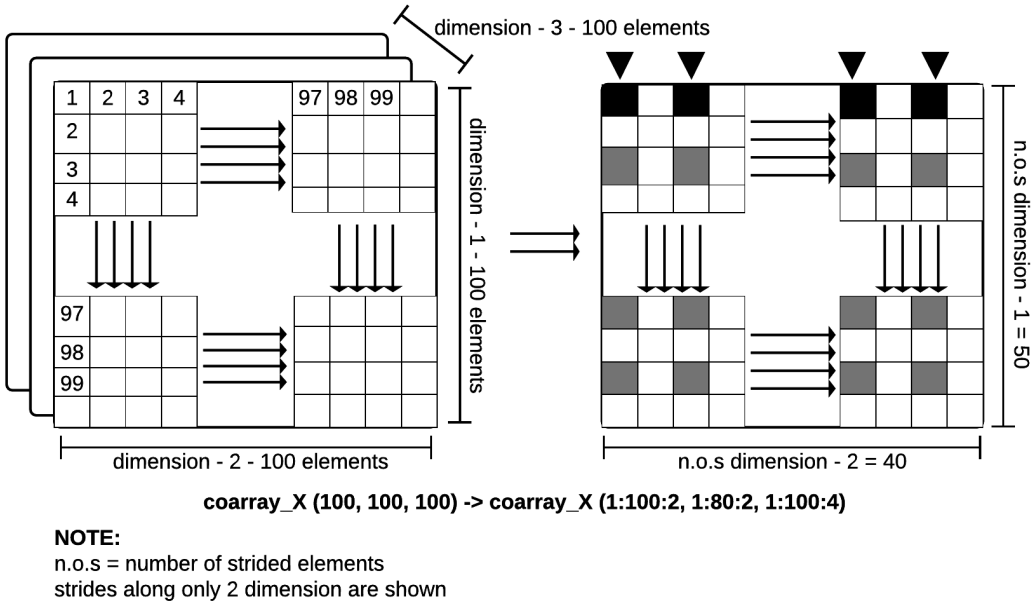

Figure 4.3: Example for Multi-dimensional Strides

dark spots on Figure 4.3 shows the locations where `shmem_iput` operations occur. We implemented both these two optimizations and call this algorithm `2dim_strided`. Algorithm 2, represents algorithm `2dim_strided`.

4.5 Support for Locks

CAF includes support for locks in order for an application to manage exclusive access to remotely-accessible data objects. Locks are required to be coarrays, and an image may acquire and release a lock at any other image. For example, for a lock

ALGORITHM 2: Baseline Algorithm: use of *iput* routines

```
procedure shmem_iputmem_using_iput(void *dest_ptr, const void *src_ptr, ptrdiff_t
dest_stride, ptrdiff_t src_stride, size_t blksize, size_t nblks, int pe_id)
    char *dest = dest_ptr, *src = src_ptr;
    ptrdiff_t sst = src_stride, dst = dest_stride;
    bool stride_16 = true, stride_8 = true, stride_4 = true;
    if (dst mod 4 ≠ 0 ∨ sst mod 4 ≠ 0) then
|     stride_4 = false; stride_8 = false; stride_16 = false;
    end
    else if (dst mod 8 ≠ 0 ∨ sst mod 8 ≠ 0) then
|     stride_8 = false; stride_16 = false;
    end
    else if (dst mod 16 ≠ 0 ∨ sst mod 16 ≠ 0) then
|     stride_16 = false;
    end
    if (blksize ≥ 16 ∧ stride_16) then
|     int size = blksize/16;
|     for (int i = 1; i ≤ size; i++) do
|   |     shmem_iput128(dest, src, dst/16, sst/16, nblks, pe_id);
|   |     dest += 16; src += 16;
|     end
|     blksize -= 16*size;
    end
    if (blksize ≥ 8 ∧ stride_8) then
|     int size = blksize/8;
|     for (int i = 1; i ≤ size; i++) do
|   |     shmem_iput64(dest, src, dst/8, sst/8, nblks, pe_id);
|   |     dest += 8; src += 8;
|     end
|     blksize -= 8*size;
    end
    if (blksize ≥ 4 ∧ stride_4) then
|     int size = blksize/4;
|     for (int i = 1; i ≤ size; i++) do
|   |     shmem_iput32(dest, src, dst/4, sst/4, nblks, pe_id);
|   |     dest += 4; src += 4;
|     end
    end
    else
|     shmem_iputmem_using_putmem(dest_ptr, src_ptr, dst, sst, blksize, nblks, pe_id);
    end
end procedure
```

that is declared with `type(lock_type) :: lck[*]`, an image may acquire the lock `lck` at image j using the statement `lock(lck[j])`, and it may release it using the statement `unlock(lock[j])`. Note that another image may simultaneously acquire the corresponding `lck` lock at another image. OpenSHMEM also provides support for locks, which must be symmetric variables. However, in OpenSHMEM symmetric locks are treated as a single, logically *global* entity, meaning that the library does not permit acquiring or releasing a lock at a specific PE. Consequentially, implementing CAF locks using OpenSHMEM's lock support would not be feasible. For a program running with N images, each image would need to allocate a symmetric array of size N for each declaration of a lock. A more space-efficient approach is necessary.

In our implementation, we adapted the well-known MCS lock algorithm [29] for scalable shared memory systems, designed to avoid spinning on non-local memory locations, to efficiently handle locks in CAF programs. In this algorithm, a distributed FIFO queue is used to keep track of images which are contending for lock `lck`, where each image holds a node within the queue called a *qnode* and `lck` contains an internal *tail* field which points to the last qnode in the queue. A *locked* field within the qnode indicates whether the lock is still being held by a predecessor image. A *next* field refers to a successor image which should be notified after the local image acquires and releases the lock, by resetting its qnode's *locked* field.

At any moment, an image may have up to M+1 allocated qnodes, corresponding to M currently held locks and potentially one additional lock it is currently waiting to acquire. We use a hash table lookup, with the hash key being the tuple (`lck`, `j`) for the lock `lck[j]`, to check if a lock is currently held during execution of the `lock`

statement or to obtain the local qnode for a held lock during execution of the `unlock` statement. When an image executes the `lock` statement, it will allocate a local qnode for the lock, and then it will use an atomic fetch-and-store operation to update the lock variable's tail pointer to point to its qnode. The fetched value will point to the prior tail, i.e. its predecessor's qnode. If this value is not $\varnothing$, it uses this to set its predecessor's *next* field. It will then locally spin on its qnode's *locked* field until it is reset to 0 by its predecessor. When an image executes the `unlock` statement, it uses an atomic compare-and-swap operation on the lock variable to conditionally set the *tail* field to $\varnothing$ if its qnode is still the tail (no successor yet). If there is a detected successor, then the image will reset the *locked* field of the successor's qnode. Finally, it will deallocate its qnode and remove it from the hash table.

Since the qnode must be remotely accessible to other images, it is allocated out of the pre-allocated buffer space for non-symmetric remote-accessible data. OpenSHMEM provides both atomic fetch-and-store and compare-and-swap for implementing the updates to the lock variable. The *tail* and *next* fields, functioning as pointers to qnodes belong to a remote image, are represented using 20 bits for the image index, 36 bits for the offset of the qnode within the remote-accessible buffer space, and the final 8 bits reserved for other flags. By packing this "remote pointer" within a 64-bit representation, we can utilize support for 8-byte remote atomics provided by OpenSHMEM.

4.6 Unsupported Features

Sections 4.2, 4.3, 4.4, and 5.2.3 show the similarities and difference between the properties of OpenSHMEM and CAF. It shows the analysis on implementing CAF over OpenSHMEM. Though most CAF properties can be implemented using OpenSHMEM, there are few properties in CAF which are not currently supported by the OpenSHMEM-1.2 specification. Those unsupported properties include atomic operations like atomic and, atomic xor and atomic or. 1-byte or *char* data type for strides where the element size is 1, is also not currently supported by OpenSHMEM.

Chapter 5

Experimental Results

5.1 Experimental Setup

Cluster	Nodes	Cores/Node	Processor Type	Interconnect
Stampede (TACC)	6,400	16	Intel Xeon E5 Sandy Bridge	InfiniBand Mellanox Switches/HCAs
Cray XC30	64	16	Intel Xeon E5 Sandy Bridge	Dragonfly interconnect with Aries
Titan (OLCF)	18,688	16	AMD Opteron	Cray Gemini interconnect

Table 5.1: Experimental Setup and Machine configuration details for Performance Analysis of CAF over OpenSHMEM Implementation

This section discusses experimental results for our implementation of UHCAF using OpenSHMEM as a communication layer on three sets of benchmarks: (1) the PGAS Microbenchmark suite [2], which contains code designed to test the performance and correctness for put/get operations and locks written in CAF, (2) a

distributed hash table benchmark which makes use of locks, and (3) a CAF version of the Himeno Benchmark.

We used three different machines for our experimental analysis. The configurations of these machines are given in Table 5.1. For all the experiments involving the UHCAF implementation, we used the OpenUH compiler version 3.40 with CAF runtime implemented over either GASNet or the native SHMEM implementations (Cray SHMEM or MVAPICH2-X SHMEM).

Stampede We used the Stampede supercomputing system at Texas Advanced Computing Center (TACC). We made use of the SHMEM and MPI-3.0 implementations available in MVAPICH2-X version 2.0b in Stampede for our analysis. We also used GASNet version 1.24.0 with IBV conduit for our relative comparisons.

Cray XC30 On the Cray XC30 machine, we used Cray SHMEM version 6.3.1, Cray Fortran compiler 8.2.6, and GASNet version 1.24.0 with Aries conduit for our relative analysis.

Titan Titan is a Cray XK7 supercomputing system at Oak Ridge Leadership Computing Facility (OLCF). We have used Cray SHMEM 6.3.1 and Cray Fortran compiler 8.2.6 and used GASNet version 1.24.0 with Gemini conduit for our analysis.

5.2 PGAS Microbenchmarks

We used the PGAS Microbenchmark suite [2] for analyzing the performance of CAF over OpenSHMEM in various environments described above to measure contiguous

and multi-dimensional strided `put` bandwidth as well as performance of locks.

5.2.1 Contiguous Communications

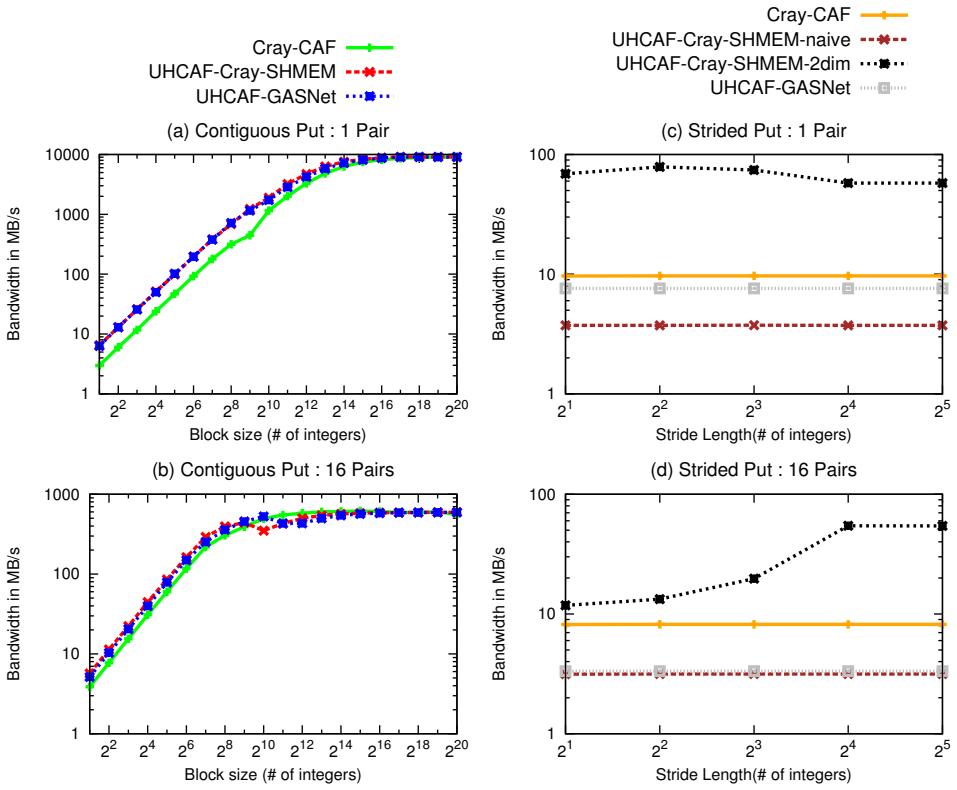

Figure 5.1: PGAS Microbenchmark Tests on Cray XC30: Put Bandwidth and 2-Dimensional Strided Put bandwidth

Based on the RMA design implementations described in Section 4.3, we analyzed the performance of CAF `put/get` tests using Cray Fortran (Cray CAF) vs OpenUH with CAF runtime (UHCAF) executing over Cray SHMEM on the Cray

XC30 machine. The same tests are also performed using UHCAF over MVAPICH2-X SHMEM vs UHCAF over GASNet on Stampede. Here, the bandwidth for the `put/get` operations performed between image pairs with each image on a different node are captured. We already showed in the previous performance tests described in Section 3.2 that the performance of OpenSHMEM is more suitable for short message sizes. These results are confirmed in plots (a) and (b) in Figure 5.1 Figure 5.2. We obtain an average of 18% improvement in UHCAF implementation over OpenSHMEM in both the Cray XC30 and Stampede environment.

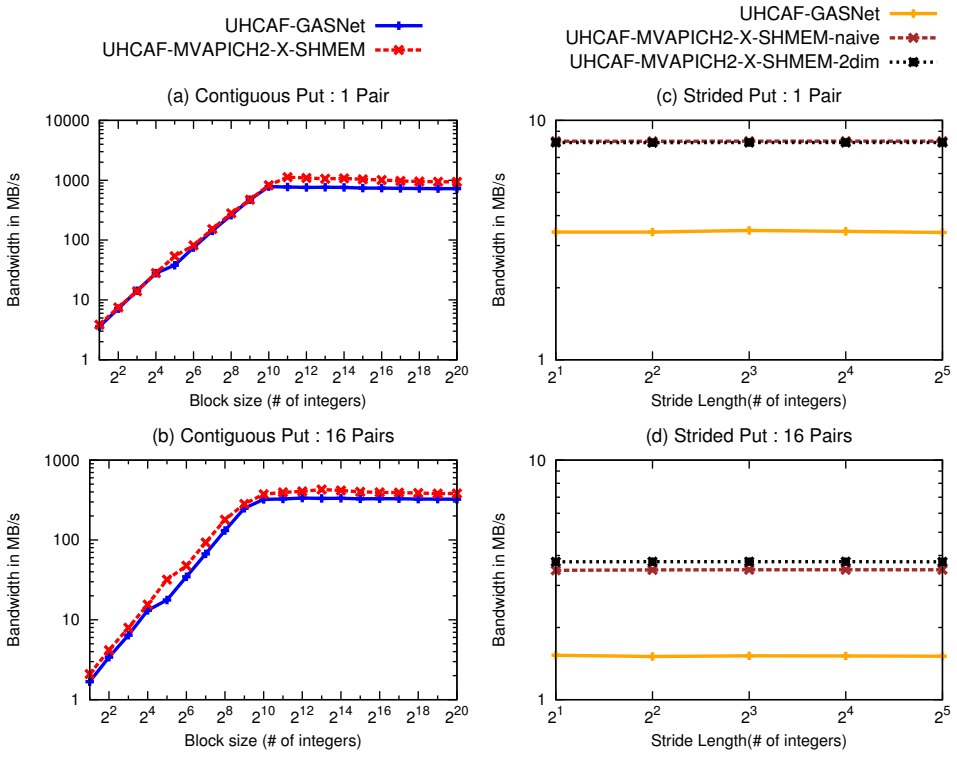

Figure 5.2: PGAS Microbenchmark Tests on Stampede: Put Bandwidth and 2-Dimensional Strided Put bandwidth

5.2.2 Strided Communications

Results from both Cray XC30 and Stampede to measure the bandwidth of strided `put` are presented in plots (c) and (d) in Figure 5.1 and Figure 5.2, respectively. The `2dim_strided` algorithm described in Section 4.4 is based on the assumption that the single dimensional strided routines `shmem_iput` or `shmem_iget` are optimized. In this microbenchmark test, we capture the bandwidth for 1-sided strided `put` on an image pair with each image of the pair on a separate node.

Results shown in plots (c) and (d) for Figure 5.2 show that the 1-dimensional strided routines `shmem_iput` or `shmem_iget` for the MVAPICH2-X SHMEM implementation is not optimized. They also show that UHCAF over MVAPICH2-X SHMEM for the naïve and the `2dim_strided` implementations are the same, because the `shmem_iput` or `shmem_iget` routines are performing multiple `shmem_putmem` or `shmem_getmem` calls underneath. However, OpenSHMEM performs better than the naïve GASNet implementation using multiple `shmem_putmem` or `shmem_getmem` routines.

Results shown in plots (c) and (d) for Figure 5.1 show the results on the Cray XC30 system. We observe that the `shmem_iput`/`shmem_iget` routines are optimized for Cray SHMEM using DMAPP and thus are more efficient than the naïve implementation using multiple `shmem_putmem` or `shmem_getmem` routines. Using the `2dim_strided` algorithm, we can get better results than with UHCAF using the naïve algorithm or with the Cray CAF implementation. The results show around 3x

improvement in bandwidth using UHCAF implementation over Cray SHMEM compared to Cray CAF, and 9x improvement compared to the naïve implementation.

5.2.3 Locks

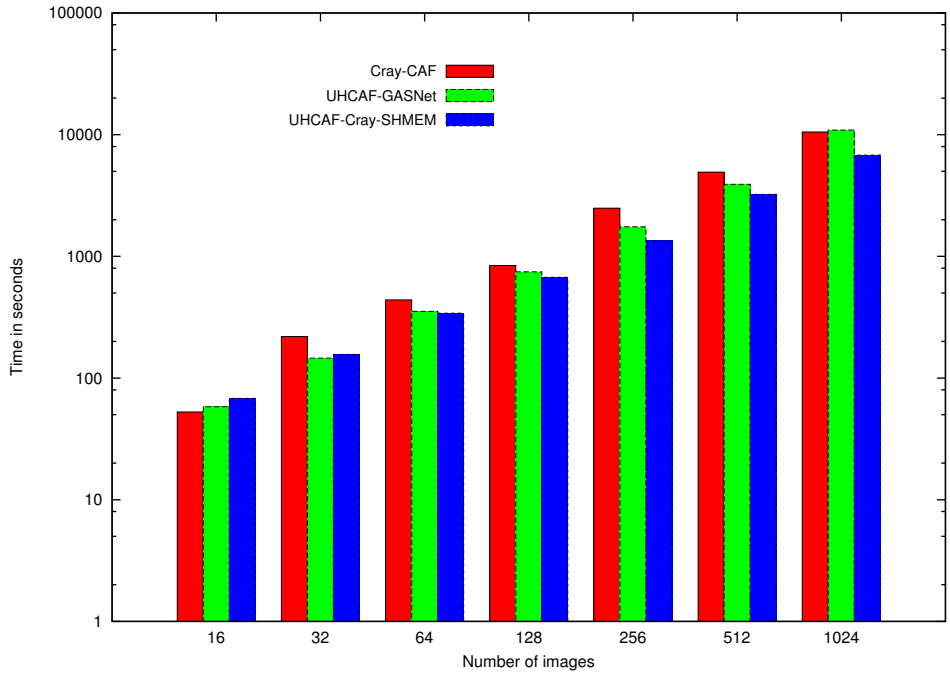

Figure 5.3: Microbenchmark test for locks on Titan. All images trying to attain and release lock on image 1.

We used a simple microbenchmark where all the images try to repeatedly acquire and then release a lock on image 1. Execution time results on Titan using up to 1024 images (over 64 nodes) are shown in Figure 5.3 where we compare Cray CAF, UHCAF over GASNet, and UHCAF over Cray SHMEM. Locking with the UHCAF over Cray SHMEM implementation performs better than with the UHCAF over

GASNet implementation, especially for number of images $\geq$ 128 images. Moreover, the implementation of locks using UHCAF over Cray SHMEM is more efficient than the one in Cray CAF. On average, using UHCAF over Cray SHMEM is 22% faster than using Cray CAF and 10% faster than using UHCAF over GASNet.

5.3 Distributed Hash Table (DHT) Benchmark

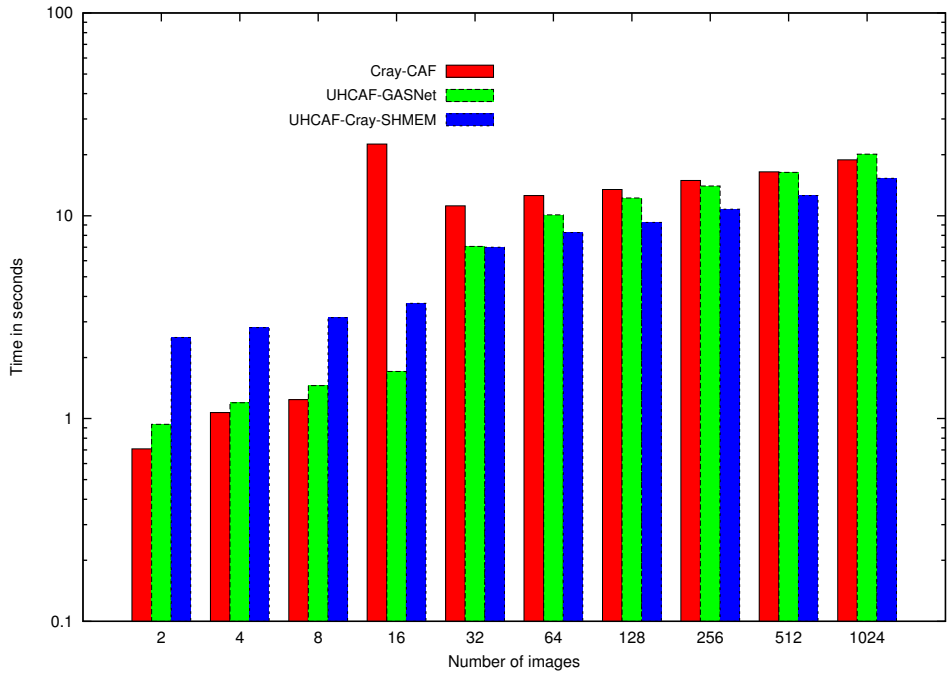

Figure 5.4: Test for locks using Distributed Hash Table Benchmark in Titan

In the Distributed Hash Table (DHT) benchmark [26], each image will randomly access and update a sequence of entries in a distributed hash table. In order to prevent simultaneous updates to the same entry, some form of atomicity must be

employed; this is achieved using coarray locks. Comparison of results on Titan until 1024 cores (over 64 nodes) are presented in Figure 5.4. We observe that the DHT benchmark using the UHCAF over Cray SHMEM implementation is 28% faster than the Cray CAF implementation and 18% faster than the UHCAF over GASNet implementation.

5.4 Himeno Parallel Benchmark

The Himeno parallel benchmark is used to evaluate the performance of an incompressible fluid analysis code, by using Jacobi iteration method to solve Poisson's equation. The CAF version of this benchmark is used to measure the performance of the CAF implementation over GASNet versus CAF over MVAPICH2-X SHMEM on Stampede. Figure 5.5 shows the results of this comparison for up to 2048 total cores (across 128 nodes). We see that the performance of UHCAF over MVAPICH2-X SHMEM is better than UHCAF over GASNet, when the number of images ≥ 16. This is because MVAPICH2-X SHMEM is highly optimized for inter-node communication.

The most important part of the Himeno benchmark is the usage of matrix oriented strided data for the halo communication. We found the best implementation for this benchmark is CAF over MVAPICH2-X SHMEM using the naïve algorithm. Note that there was no benefit in using our `2dim_strided` algorithm on the Stampede system, because MVAPICH2-X SHMEM implements the `shmem_iput/shmem_iget` routines as a series of contiguous put/get operations, and also the `2dim_strided`

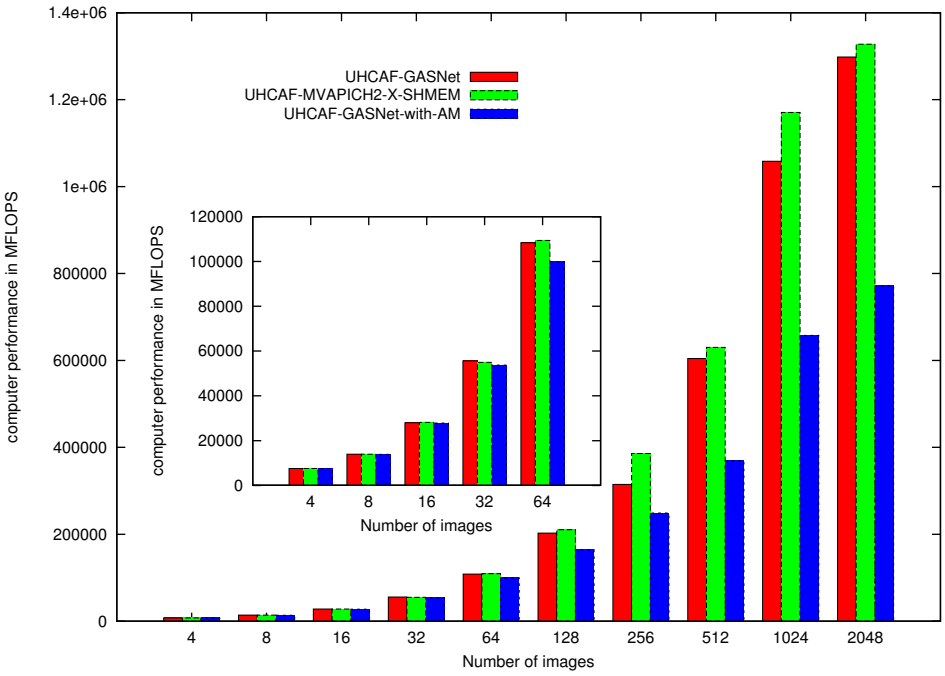

Figure 5.5: Test for strided data communication using Himeno Benchmark in Stampede

does not help for the matrix oriented multi-dimensional strides.

The major difference between the regular multi-dimensional strides and the matrix oriented multi-dimensional strides is the stride in the base dimension. For matrix-oriented strides, the base dimension is a contiguous block. It would be better to perform a single `shmem_putmem` than a single `shmem_iput` in the base dimension for this case. This is the reason why the Himeno benchmark was not performed on Titan or Cray XC30 machine. On an average, we obtain around 6% better performance and to the maximum we obtain 22% better performance with UHCAF over MVAPICH2-X SHMEM compared to the UHCAF over GASNet implementation.

Chapter 6

Related Work

Our optimizations on the multi-dimensional strided algorithm relies on the indexed, 1-dimensional strided put/get routines being optimized in OpenSHMEM. Though we cannot expect this optimization to be available in all the OpenSHMEM implementations, there are works [33] [12] related to these optimizations for non-contiguous derived data types which can be applied for OpenSHMEM 1-dimensional strided optimization as well.

[39] stresses on the importance of a generalized communication layer for PGAS programming models. There they emphasize the importance of using DMAPP as a unified communication layer. A common unified communication layer also eases the interoperability between different programming models.

In [36], PGAS models are combined with MPI implementations to obtain better

performance results for Gyrokinetic Fusion Applications. Various one-sided algorithms have been developed using CAF and integrated into the best performing algorithm using the preexisting MPI codes to obtain better performance. Hence, a hybrid application using PGAS and MPI is not uncommon. Yang et al. [42] have shown the importance of interoperability between CAF and MPI using their CAF 2.0 [28] implementation over MPI-3.0.

A comparative study on different parallel programming models has been presented in [38]. It is shown that PGAS models, in particular OpenSHMEM, has the potential to obtain better performance results than MPI implementations. Also, developing specific and highly optimized PGAS models for different architecture would be a critical task. The solution to this issue is to use a common base for all the different PGAS implementations which can handle all the low-level, architecture-specific optimizations. For example, Cray has implemented all of their PGAS models (i.e., UPC, CAF and OpenSHMEM) with DMAPP [41]. Libraries like DMAPP are vendor specific; creating a standard among different vendor libraries would be highly improbable. OpenSHMEM may be considered as a potential candidate for this purpose. This will make it possible to increase the potential for developing many interoperable hybrid applications with OpenSHMEM and other PGAS models.

Chapter 7

Conclusion

Through this work, we demonstrated how OpenSHMEM can function as a communication layer for CAF. The addition of OpenSHMEM to the runtime implementation provided in the OpenUH CAF compiler (UHCAF) was done via two steps: (1) direct mapping of CAF language features to OpenSHMEM, including allocation of remotely accessible objects, one-sided communication, and various types of synchronization, and (2) evaluation of various algorithms we developed for implementing non-contiguous communications and acquisition and release of remote locks using the OpenSHMEM interface. We obtain an average of 18% of improvement of bandwidth for the put/get operations performed between image pairs using UHCAF implemented over OpenSHMEM in both the Cray XC30 and Stampede environments. For strided communications, we achieved around 3x improvement of bandwidth using UHCAF implementation over Cray SHMEM compared to performance with the Cray Fortran implementation. The result of our lock microbenchmark was that UHCAF

over Cray SHMEM executed 22% faster than Cray Fortran implementation and 10% faster than the UHCAF over GASNet implementation. Performance improvement relative to the original UHCAF over GASNet implementation was also observed for the Himeno CAF benchmark.

We plan as future work to further improve the multi-dimensional strided communications algorithm. We intend to account for more parameters to negotiate the trade-off between locality and minimizing the number of single calls, such as cache line size and number of elements in each dimension. We plan also to utilize the `shmem_ptr` operation to convert intra-node accesses into direct load/store instructions and to develop hybrid codes with CAF and OpenSHMEM.

Chapter 8

Terms and Abbreviations

Terms	Meaning
AMO	Atomic Memory Operations, like atomic AND, OR, SWAP, INC, CSWAP etc.,
API	Application Programming Interface
ARMCI	Aggregate Remote Memory Copy Interface - a communication layer
BR	Bruck - a collective reduction algorithm described in this work
CAF	Coarray Fortran
CAF 2.0	Coarray Fortran implementation in Rose compiler
ComEx	Communications Runtime for Extreme Scale
Cray	An American supercomputer manufacturer previously known as Cray Research, Inc.
Cray CAF	Coarray Fortran Implementation in Cray Fortran Compiler
Cray MPICH	MPI implementation in Cray Message Passing Toolkit
Cray SHMEM	OpenSHMEM implementation in Cray Message Passing Toolkit

Terms	Meaning
CUDA	Compute Unified Device Architecture
DHT	Distributed Hash Table
DMAPP	Distributed Memory Application
FFT	Fast Fourier Transform
FIFO	First In First Out
FT	Flat Tree - a collective reduction algorithm described in this work
GA	Global Arrays Toolkit from PNNL
GASNet	GASNet is a language-independent, low-level networking layer
GCC	GNU Compiler Collection
GDDR	Graphics Double Data Rate
Get	One-sided read operation
Gfortran	Fortran compiler in GNU Compiler Collection
GPU	Graphics Processor Unit
IB	InfiniBand
IBV	InfiniBand Verbs
Image	A Process in an SPMD program. Also called Processing Element(PE)
KNC	Knights Corner
KNF	Knights Ferry
KNH	Knights Hill
KNL	Knights Landing
libfabrics	OpenFabrics Interface
MIC	Many Integrated Core
MPI	Message Passing Interface
MV2X	MPI implementation from NOWLAB, Ohio State University
MVAPICH MPI	implementation from NOWLAB, Ohio State University
MVAPICH2-X MPI-3.0	MPI-3.0 RMA features implemented in MV2X
MVAPICH2-X SHMEM	OpenSHMEM implementation in MV2X
NIC	Network Interface Card
NUMA	Non-uniform Memory Architecture
OFIWG	OpenFabrics Interface Working Group

Terms	Meaning
OLCF	Oak Ridge Leadership Computing Facility
OpenSHMEM	name of standards for different SHMEM implementations
OpenUH	Open Source compiler for C, C++ Fortran with support for OpenMP, CAF and OpenACC
OS	Operating System
PAMI	Parallel Active Message Interface
PE	A Process in an SPMD program. Described by OpenSHMEM standards
PGAS	Partitioned Global Address Space
PNNL	Pacific Northwest National Laboratory
Put	One-sided write operation
RD	Recursive Doubling - a collective reduction algorithm described in this work
RDMA/RMA	Remote Direct Memory Access
RSAG	Reduced Scatter All Gather - a collective reduction algorithm described in this work
SGI	An American manufacturer of high-performance computing solutions, previously Silicon Graphics, Inc
SMP	Shared Memory Programming
SPMD	Single Program Multiple Data
TACC	Texas Advanced Computing Center
UCCS	Universal Common Communication Substrate
UCR	Unified Common Runtime
UPC	Unified Parallel C
VPU	Vector processing Unit

Bibliography

[1] Himeno Parallel Benchmark. http://accc.riken.jp/2444.htm.

[2] HPCTools PGAS-Microbenchmarks. https://github.com/uhhpctools/pgas-microbench.

[3] Intel Xeon Phi Co-Processors code name Knights Corner. https://software.intel.com/en-us/articles/intel-xeon-phi-coprocessor-codename-knights-corner.

[4] OpenSHMEM NAS Parallel Benchmarks, Version 1.0a. http://www.openshmem.org/site/Downloads/Examples.

[5] OSU SHMEM implementation over MVAPICH2-X. http://mvapich.cse.ohio-state.edu/overview/.

[6] OpenSHMEM Application Programming Interface (version 1.0). http://upc.gwu.edu/documentation.html, 2012.

[7] T. 500. TOP 500 System statistis and Performance Update. Technical report.

[8] G. Almasi. In D. A. Padua, editor, *Encyclopedia of Parallel Computing*. 2011.

[9] D. H. Bailey. *NAS Parallel Benchmarks*. Springer, 2011.

[10] D. Bonachea. GASNet Specification, V1.1. Technical report, 2002.

[11] J. Bruck, C.-T. Ho, S. Kipnis, E. Upfal, and D. Weathersby. Efficient Algorithms for All-to-All Communications in Multiport Message-Passing Systems. *Parallel and Distributed Systems, IEEE Transactions on*, 8, Nov 1997.

[12] S. Byna, X.-H. Sun, R. Thakur, and W. Gropp. Automatic Memory Optimizations for Improving MPI Derived Datatype Performance. In *Proceedings of the 13th European PVM/MPI User's Group Conference on Recent Advances*

in Parallel Virtual Machine and Message Passing Interface, EuroPVM/MPI'06, 2006.

[13] W. W. Carlson, J. M. Draper, and D. E. Culler. Introduction to UPC and Language Specification.

[14] B. Chamberlain, D. Callahan, and H. Zima. Parallel Programmability and the Chapel Language. *Int. J. High Perform. Comput. Appl.*, 21(3), Aug. 2007.

[15] B. Chapman, T. Curtis, S. Pophale, S. Poole, J. Kuehn, C. Koelbel, and L. Smith. Introducing OpenSHMEM: SHMEM for the PGAS Community. In *Proceedings of the Fourth Conference on Partitioned Global Address Space Programming Model*, PGAS '10, 2010.

[16] B. Chapman, D. Eachempati, and O. Hernandez. Experiences Developing the OpenUH Compiler and Runtime Infrastructure. *Int. J. Parallel Program.*

[17] P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. von Praun, and V. Sarkar. X10: An Object-oriented Approach to Non-uniform Cluster Computing. In *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA'05, 2005.

[18] Cray. Using the GNI and DMAPP APIs. Technical report, February 2014.

[19] A. Fanfarillo, T. Burnus, V. Cardellini, S. Filippone, D. Nagle, and D. Rouson. OpenCoarrays: Open-source Transport Layers Supporting Coarray Fortran Compilers. In *Proceedings of the 8th International Conference on Partitioned Global Address Space Programming Models*, PGAS '14, 2014.

[20] J. R. Hammond. Towards a Matrix-oriented Strided Interface in OpenSHMEM. In *Proceedings of the 8th International Conference on Partitioned Global Address Space Programming Models*, PGAS '14, 2014.

[21] J. R. Hammond, S. Ghosh, and B. M. Chapman. Implementing OpenSHMEM Using MPI-3 One-Sided Communication. In *Proceedings of the First Workshop on OpenSHMEM and Related Technologies. Experiences, Implementations, and Tools - Volume 8356*, OpenSHMEM 2014, 2014.

[22] W. Huang, G. Santhanaraman, H. Jin, Q. Gao, and D. Panda. Design and Implementation of High Performance MVAPICH2: MPI2 over InfiniBand. 2007.

[23] S. Kumar, A. Mamidala, D. Faraj, B. Smith, M. Blocksome, B. Cernohous, D. Miller, J. Parker, J. Ratterman, P. Heidelberger, D. Chen, and B. Steinmacher-Burrow. PAMI: A Parallel Active Message Interface for the Blue Gene/Q Supercomputer. In *Parallel Distributed Processing Symposium (IPDPS), 2012 IEEE 26th International*, pages 763–773, May 2012.

[24] Libfabric. Specification and Project Details: OpenFabric. Technical report.

[25] J. Lin, K. Hamidouche, K. Lu, M. Li, and D. Panda. High-Performance Coarray Fortran Support with MVAPICH2-X: Initial Experience and Evaluation. May 2015.

[26] C. Maynard. Comparing One-Sided Communication With MPI, UPC and SHMEM. Technical report, Cray Users Group (CUG), 2012.

[27] Mellanox. Specification: Mellanox InfiniBand Verbs API. Technical report.

[28] J. Mellor-Crummey, L. Adhianto, W. N. Scherer, III, and G. Jin. A New Vision for Coarray Fortran. In *Proceedings of the Third Conference on Partitioned Global Address Space Programing Models*, PGAS '09, 2009.

[29] J. M. Mellor-Crummey and M. L. Scott. Algorithms for Scalable Synchronization on Shared-memory Multiprocessors. *ACM Trans. Comput. Syst.*, 9(1), Feb. 1991.

[30] N. Namashivayam, S. Ghosh, D. Khaldi, D. Eachempati, and B. Chapman. Native Mode-Based Optimizations of Remote Memory Accesses in OpenSHMEM for Intel Xeon Phi. In *Proceedings of the 8th International Conference on Partitioned Global Address Space Programming Models*, PGAS '14. ACM, 2014.

[31] J. Nieplocha and B. Carpenter. ARMCI: A Portable Remote Memory Copy Libray for Ditributed Array Libraries and Compiler Run-Time Systems. In *Proceedings of the 11 IPPS/SPDP'99 Workshops Held in Conjunction with the 13th International Parallel Processing Symposium and 10th Symposium on Parallel and Distributed Processing*, 1999.

[32] J. Nieplocha, R. Harrison, and R. Littlefield. Global Arrays: a portable ldquo;shared-memory rdquo; programming model for distributed memory computers. In *Supercomputing '94., Proceedings*, Nov 1994.

[33] J. Nieplocha, V. Tipparaju, and M. Krishnan. Optimizing Strided Remote Memory Access Operations on the Quadrics QsNetII Network Interconnect. In *Proceedings of the Eighth International Conference on High-Performance Computing in Asia-Pacific Region*, HPCASIA '05, 2005.

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

[35] S. Poole, O. Hernandez, J. Kuehn, G. Shipman, A. Curtis, and K. Feind. OpenSHMEM - Toward a Unified RMA Model. In D. Padua, editor, *Encyclopedia of Parallel Computing*. 2011.

[36] R. Preissl, N. Wichmann, B. Long, J. Shalf, S. Ethier, and A. Koniges. Multithreaded Global Address Space Communication Techniques for Gyrokinetic Fusion Applications on Ultra-Scale Platforms. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '11, 2011.

[37] P. Shamis, M. G. Venkata, M. B. Lopez, O. Baker, Y. Hernandez, M. Itigin, G. Dubman, R. Shainer, L. Graham, Y. Liss, , S. Shahar, D. Potluri, D. Rossetti, D. Becker, C. Poole, S. Lamb, C. Kumar, G. Stunkel, G. Bosilca, and A. Bouteiller. UCX: An Open Source Framework for HPC Network APIs and Beyond. In *HOT-Interconnects: 23rd Annual Symposium on High-Performance Interconnects*, August 2015.

[38] H. Shan and J. P. Singh. A Comparison of MPI, SHMEM and Cache-Coherent Shared Address Space Programming Models on a Tightly-Coupled Multiprocessors. *Int. J. Parallel Program.*, 29.

[39] M. ten Bruggencate and D. Roweth. DMAPP: An API for One-Sided Programming Model on Baker Systems. Technical report, Cray Users Group (CUG), August 2010.

[40] R. Thakur, R. Rabenseifner, and W. Gropp. Optimization of Collective Communication Operations in MPICH. *International Journal of High Performance Computing Applications*, 19(1):49–66, 2005.

[41] A. Vishnu, M. ten Bruggencate, and R. Olson. Evaluating the Potential of Cray Gemini Interconnect for PGAS Communication Runtime Systems. In *Proceedings of the 2011 IEEE 19th Annual Symposium on High Performance Interconnects*, HOTI '11, 2011.

[42] C. Yang, W. Bland, J. Mellor-Crummey, and P. Balaji. Portable, MPI-interoperable Coarray Fortran. *SIGPLAN Not.*