COARRAY FORTRAN RUNTIME IMPLEMENTATION
IN OPENUH

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
Debjyoti Majumder
December 2011
COARRAY FORTRAN RUNTIME IMPLEMENTATION
IN OPENUH

Debjyoti Majumder

APPROVED:

Dr. Barbara Chapman, Chairman
Dept. of Computer Science

Dr. Jaspal Subhlok
Dept. of Computer Science

Dr. Terrence Liao
TOTAL E&P Research & Technology USA, LLC

Dean, College of Natural Sciences and Mathematics
Acknowledgements

I would like to express my gratitude to Dr. Barbara Chapman for providing financial support and for creating an excellent environment for research. I would like to thank my mentor Deepak Eachempati. Without his guidance, this work would not have been possible. I sincerely thank TOTAL S.A. for funding this project and Dr. Terrence Liao for his valuable inputs and help in performing experiments on TOTAL’s hardware. I also thank the interns at TOTAL, Asma Farjallah, and France Boillod-Cerneux for the applications that I used for performance evaluation. I thank Hyoungjoon Jun for his earlier work on the CAF runtime. I thank Tony Curtis for his suggestions and excellent work on maintaining the infrastructure on which I ran my experiments. I thank Dr.Edgar Gabriel for answering all my questions and his help with using the shark cluster. I also thank the Berkeley Lab and PNNL’s ARMCI developers for their guidance in using GASNet and ARMCI. Lastly, I would like to thank the Department of Computer Science for giving me the opportunity to be in this great country of USA and all my lab-mates and friends.
COARRAY FORTRAN RUNTIME IMPLEMENTATION

IN OPENUH

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
Debjyoti Majumder
December 2011
Abstract

Coarray Fortran (CAF) is an extension to the Fortran programming language which is included in the Fortran 2008 standard. It enables parallel programming in Fortran with minimum change in the language syntax. Like UPC and Global Arrays, CAF is a Partitioned Global Address Space (PGAS) language, and is suitable for both shared memory and distributed memory systems. CAF follows the SPMD programming model where multiple copies of the same program are executed independently. Providing features for expressing parallelism in the language is beneficial, in contrast to an external library, as the compiler has more scope for optimizations.

Our effort has created the first efficient open-source compiler for CAF. This thesis presents the runtime library used by the compiler, which is essential for implementing the PGAS memory model. The thesis describes how one-sided remote memory access is achieved, and how other runtime features are implemented. It also presents the optimizations implemented in the runtime library and the mechanisms used to preserve program correctness. Experimental results using micro-benchmarks and real applications show that the implementation performs as well as MPI and other PGAS programming models, and in some cases outperforms MPI.

We also describe the different timing and tracing options built into the runtime library to provide useful debugging information and insights into performance bottlenecks.
# Contents

1 Introduction ..................................................... 1
  1.1 Coarray Fortran ............................................. 1
  1.2 Motivation ................................................... 5
  1.3 OpenUH Compiler Overview .................................. 5
  1.4 New Contribution ............................................ 6
  1.5 Joint Contribution ........................................... 7
  1.6 Thesis Overview ............................................. 7
  1.7 Terms and Abbreviations .................................... 7

2 Related Work ..................................................... 9
  2.1 Global Arrays Toolkit ....................................... 9
  2.2 Unified Parallel C (UPC) ................................... 10
    2.2.1 Berkeley UPC .......................................... 11
  2.3 Remote Memory Access in MPI-2 ............................ 12

3 Enabling Remote Direct Memory Access (RDMA) .................. 14
  3.1 GASNet ...................................................... 17
  3.2 ARMCI ....................................................... 21
    3.2.1 ARMCI over One-sided MPI ............................ 25
  3.3 GASNet & ARMCI Comparison ................................ 27
## List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1</td>
<td>Logical view of memory in CAF</td>
<td>3</td>
</tr>
<tr>
<td>1.2</td>
<td>The OpenUH compiler infrastructure</td>
<td>6</td>
</tr>
<tr>
<td>3.1</td>
<td>GASNet/ARMCI library provides network-independent PGAS abstraction</td>
<td>17</td>
</tr>
<tr>
<td>4.1</td>
<td>Dope and codope vectors (cited from Jun’s thesis)</td>
<td>31</td>
</tr>
<tr>
<td>4.2</td>
<td>Code structure</td>
<td>32</td>
</tr>
<tr>
<td>4.3</td>
<td>Shared memory heap</td>
<td>34</td>
</tr>
<tr>
<td>4.4</td>
<td>Creation and management of a 500bytes shared memory segment</td>
<td>35</td>
</tr>
<tr>
<td>4.5</td>
<td>Deallocation of a shared memory segment</td>
<td>36</td>
</tr>
<tr>
<td>4.6</td>
<td>Screen shot of time-trace output</td>
<td>44</td>
</tr>
<tr>
<td>4.7</td>
<td>Screen shot of time summary trace output</td>
<td>45</td>
</tr>
<tr>
<td>4.8</td>
<td>Screen shot of memory-trace output</td>
<td>45</td>
</tr>
<tr>
<td>4.9</td>
<td>Screen shot of debug output</td>
<td>46</td>
</tr>
<tr>
<td>5.1</td>
<td>Get cache data structure</td>
<td>48</td>
</tr>
<tr>
<td>5.2</td>
<td>Execution time difference with get-cache optimization</td>
<td>50</td>
</tr>
<tr>
<td>5.3</td>
<td>Pre-fetched data storage</td>
<td>52</td>
</tr>
<tr>
<td>5.4</td>
<td>Data structure to store the address-list to be prefetched</td>
<td>53</td>
</tr>
<tr>
<td>5.5</td>
<td>Execution time difference with pre-fetching</td>
<td>54</td>
</tr>
<tr>
<td>5.6</td>
<td>Non-blocking data structure</td>
<td>56</td>
</tr>
</tbody>
</table>
5.7 Execution time difference with non-blocking optimization ........ 58

6.1 Topology of each node on cluster ........................................ 60
6.2 Topology of a socket on the SMP ........................................ 61
6.3 Latency on cluster with Infiniband ...................................... 62
6.4 Latency on SMP .............................................................. 63
6.5 Bandwidth on cluster with Infiniband ................................... 64
6.6 Ratio of write over read on Infiniband cluster ....................... 65
6.7 Bandwidth of put on SMP ................................................. 66
6.8 Ratio of bidirectional over unidirectional write on Infiniband cluster 67
6.9 Bandwidth of non-contiguous put on cluster with Infiniband ........ 68
6.10 Bandwidth of broadcast on cluster (NP8) with Infiniband .......... 69
6.11 Bandwidth of broadcast on SMP (NP8) ................................. 69
6.12 Bandwidth of contiguous data communication ....................... 72
6.13 Bandwidth of non-contiguous data communication .................. 73
6.14 Performance gain from GASNET_VIS_AMPIPE .......................... 74
6.15 Edges: Performance improvement of UHCAF over IntelMPI with 8 processes ................................................................. 76
6.16 Faces: Performance improvement of UHCAF over IntelMPI with 8 processes ................................................................. 77
6.17 Kernel execution time (includes communication) ..................... 79
6.18 Communication time (seconds) with 8 processes ..................... 79
6.19 Communication time ratio (MPI/UHCAF) with 8 processes ........ 80
6.20 Speedup over 4 processes using fixed domain size of 16GB .......... 81
6.21 Communication time (seconds) for TTI ................................ 83
6.22 Total time (seconds) for TTI .............................................. 84
6.23 Speedup over 16 processes .............................................. 85
List of Tables

1.1 Terms and abbreviations used in this thesis . . . . . . . . . . . . . . . 8

3.1 ARMCI functions mapped to MPI one-sided functions . . . . . . . . . 26

4.1 GASNet & ARMCI functions invoked for reads and writes . . . . . . . 38

4.2 Image inquiry function results for A(10)[2:2,3:4,0:*] on 6 images . . 43

6.1 Communication time (sec) with and without GASNET_VIS_AMPIPE 74

6.2 Communication time (seconds) for edges (12) with 8 processes . . . . 75

6.3 Communication time (seconds) for faces (6) with 8 processes . . . . 76

6.4 Communication time (seconds) for corners (6) with 8 processes . . . 77

6.5 Isotropic forward wave equation solver timings (sec) with 8 processes 78

6.6 Timing (seconds) for domain size 16GB . . . . . . . . . . . . . . . . . 81

6.7 Communication time (seconds) for TTI . . . . . . . . . . . . . . . . . 82

6.8 Memory usage comparison between TTI and isometric . . . . . . . . 83

6.9 Total execution time (seconds) for 16 GB domain size . . . . . . . . 84
Chapter 1

Introduction

This thesis presents the implementation of the Coarray Fortran runtime library in the OpenUH compiler. This chapter provides an overview of the Coarray Fortran language and the motivation of this work. It lists the new contributions in the runtime and outlines the structure of this thesis. There is a glossary of terms and abbreviations that are used in the thesis at the end of this chapter.

1.1 Coarray Fortran

Coarray Fortran (CAF) is an extension to the Fortran programming language which is part of the Fortran 2008 standard [33]. It adds new features to the Fortran language to make Fortran programs execute in parallel asynchronously. It follows the SPMD (Single Program Multiple Data) model, where copies of the same program are executed on multiple processing elements (called ‘images’) which may or may not
reside on the same physical node. CAF is suitable for both shared memory systems and distributed memory systems (clusters).

The array syntax of Fortran is extended with additional subscript notation in square brackets called co-subscript. Co-subscripts can be used to refer to the object residing in a different image. New intrinsic functions which return number of images, index of the current image, etc. are added to the language. Synchronization functions are also added.

CAF follows a PGAS (Partitioned Global Address Space) memory model, which means that the CAF programmer can access memory of any remote image without the help of that image (1-sided), as if it is accessing shared memory. This is demonstrated in a simple hello world program below:

```fortran
PROGRAM HELLOWORLD

    ! create coarray with 1 dimension & 1 codimension
    integer :: A_coarray(10)[*]

    ! Initialize - Img1: 10*1 , Img2: 10*2
    A_coarray(:) = 10*this_image()

    ! Wait for all images to execute this statement
    sync all

    if (this_image() == 1) then
      ! Overwrite value on img1 with value of img2
      A_coarray(:) = A_coarray(:)[2]
    end if

    if (this_image() == 1) then
```
! print the local value on Img1

print *, "Resulting array (on image 1):", A_coarray(:)

END if

END PROGRAM

Output: Resulting array (on image 1): 20 20 20 20 20 20 20 20 20 20

In the above program, image 1 reads data from image 2 without any involvement of image 2. Figure 1.1 shows the logical shared memory view. Note however, that image 1 is aware of which data is local and which is remote. Specifying A_coarray(n) without the co-subscript accesses only the local data. This differentiates PGAS from GAS memory model that does not distinguish between local and remote memory. Since the remote memory access is explicit, it forces the user to think about optimal data distribution and optimizations like overlapping communication with computation. CAF provides programmability of shared memory systems and control of data like message passing systems.

Figure 1.1: Logical view of memory in CAF

Only objects declared as coarrays can be accessed remotely. Coarrays can be
global/static and allocatable. Like array dimension, co-dimension of a coarray can be more than 1. For multiple co-dimensions, the number of images are logically divided into a multi-dimensional matrix. This provides better visualization of the topology. For example,

real :: x(2,3)[2,3:4,*]

divides the number of images into a 2 x 2 x n matrix.

Coarrays can be remotely read and written into. For example,

b(5:6)[2] = a(3:4)

writes to the 5th and 6th element of coarray b on image 2.

a(1:2) = b(5:6)[2]

reads from the 5th and 6th element of coarray b in image 2.

CAF provides both all-to-all barrier (sync all) and point-to-point barrier (sync images). Sync images can be used to synchronize with one image or with a list of images. CAF also provides many intrinsic functions for image inquiry like lcobound, ucobound which returns the lower and upper bounds of the codimension of a coarray. However, CAF lacks many of the parallelism features provided by MPI and some other parallel programming models. For example, there is no support for teams, parallel file input-output, and non-blocking communications. This is because, the goal of CAF is to introduce minimum changes to Fortran and focus on productivity.
1.2 Motivation

There is a lack of portable open-source CAF compiler. Having an open-source compiler is important for an emerging language as it promotes sharing of ideas and encourages people to freely experiment with it. The only CAF compilers which are not hardware specific are Intel-CAF and G95. At present the Intel CAF compiler performs very poorly and the free version of G95 only allows up to 5 images to run on a cluster. Rice University has a CAF 2.0 [26] compiler which does not adhere to the Fortran 2008 standard. In the University of Houston, the CAF compiler is implemented as part of the OpenUH compiler, which is an open-source compiler for C, C++, and Fortran, and supports OpenMP.

1.3 OpenUH Compiler Overview

OpenUH is based on the Open64 compiler. It compiles C, C++, Fortran, CAF, and OpenMP on platforms like x86_64, IA64, IA32 and Opteron. Figure 1.2 shows the main modules of the OpenUH compiler on the left and describes how they work together to produce an executable. Five levels of tree-based intermediate representation (IR) called WHIRL exist to support different analysis and optimization phases. The compiler can be used for source-to-source translator using IR-to-source conversion tools.
The CAF implementation in OpenUH is called UHCAF. The Cray Fortran 95 front-end was modified to support coarray syntax and CAF intrinsics. In order to utilize the optimization capabilities of the OpenUH back-end, the coarray semantics are preserved into the back-end, where the coarray operations are converted into runtime library calls.

1.4 New Contribution

The CAF runtime was previously implemented by Hyoungjoon Jun as described in [20]. However, the implementation was redone due to change of design described in section 4.1.
1.5 Joint Contribution

Deepak Eachempati implemented the CAF support in the OpenUH compiler. He designed the runtime API structure and also implemented some portion of the runtime library.

1.6 Thesis Overview

This thesis is structured as follows. Chapter 2 describes some of the similar programming models which use one-sided remote memory access. Chapter 3 explains how the remote memory access is achieved in the CAF runtime. Chapter 4 presents the details of the runtime implementation. Chapter 5 discusses the optimizations implemented in the runtime. Chapter 6 presents the experimental results and performance evaluation. Chapter 7 summarizes the work and outlines future research directions.

1.7 Terms and Abbreviations

Table 1.1 lists the terms and abbreviations used in this thesis.
<table>
<thead>
<tr>
<th>Term</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>CAF</td>
<td>Coarray Fortran</td>
</tr>
<tr>
<td>OpenUH</td>
<td>Open Source compiler for C, C++ &amp; Fortran with support for OpenMP, CAF and UPC.</td>
</tr>
<tr>
<td>UHCAF</td>
<td>Coarray Fortran implementation in the OpenUH compiler</td>
</tr>
<tr>
<td>PGAS</td>
<td>Partitioned Global Address Space</td>
</tr>
<tr>
<td>RDMA/RMA</td>
<td>Remote Direct Memory Access</td>
</tr>
<tr>
<td>Image</td>
<td>A Process in an SPMD program. Also called Processing Element(PE)</td>
</tr>
<tr>
<td>Put</td>
<td>One-sided write operation</td>
</tr>
<tr>
<td>Get</td>
<td>One-sided read operation</td>
</tr>
<tr>
<td>ARMCI</td>
<td>Aggregate Remote Memory Copy Interface</td>
</tr>
<tr>
<td>GASNet</td>
<td>Global Address Space Networking</td>
</tr>
<tr>
<td>IB, IBV</td>
<td>Infiniband</td>
</tr>
<tr>
<td>GM</td>
<td>Communication protocol that runs over Myrinet interconnect</td>
</tr>
<tr>
<td>NIC</td>
<td>Network Interface Card</td>
</tr>
<tr>
<td>OS</td>
<td>Operating System</td>
</tr>
<tr>
<td>SMP</td>
<td>Shared Memory Processor</td>
</tr>
<tr>
<td>NUMA</td>
<td>Non Uniform Memory Access</td>
</tr>
<tr>
<td>NP</td>
<td>Number of processes / images / processing elements</td>
</tr>
<tr>
<td>MPI</td>
<td>Message Passing Interface</td>
</tr>
<tr>
<td>UPC</td>
<td>Unified Parallel C</td>
</tr>
<tr>
<td>GA</td>
<td>Global Arrays Toolkit</td>
</tr>
</tbody>
</table>

Table 1.1: Terms and abbreviations used in this thesis
Chapter 2

Related Work

This chapter describes some of the programming models which use one-sided remote memory access.

2.1 Global Arrays Toolkit

Global Arrays (GA) Toolkit [23, 28] is a PGAS library for C and Fortran. GA has SPMD model of execution and provides one-sided access to shared data structures. It uses ARMCI runtime library for allocating global memory and for accessing remote memory locations. It has a Memory Allocator interface for allocating local memory, which provide memory availability and utilization information and statistics. It also has a Distributed Arrays (DA) layer which detects the location of the data that an operation is accessing. It checks if the memory location is in the physical shared memory or remote memory and invokes the corresponding ARMCI functions.
Global arrays can be created using the function `nga_create` and read/write operations are done through `nga_get` and `nga_put`. GA also allows creation of global arrays with irregular distribution, i.e. the one process may have n elements of the array while another have m elements. Apart from standard remote read/write, synchronization and inquiry functions, GA provides a plethora of useful functions like duplicating arrays, setting data, non-contiguous data transfer, collective operations, atomic operations, etc. It supports locality (`nga_distribution`). `GA_scale` multiplies all the elements in the array by a specified factor. `GA_copy` copies the content of one global array into another. GA provides many functions to perform linear algebra operations like `nga_matmut` (matrix multiplication). All these operations can also be performed on teams or on a patch of the array. User can create ghost cells which gets automatically updated when their corresponding values are changed in the neighboring process. GA also supports parallel IO. These extensive features make GA one of the most mature and usable PGAS libraries. GA is popularly used in NWChem computational chemistry suite [38].

### 2.2 Unified Parallel C (UPC)

UPC is the parallel extension of the C programming language. It also follows the PGAS programming model. Apart from normal C variables which can be accessed only by the local process, UPC has shared variables which can be accessed by all processes. Shared variables are created in the partitioned global address space. The data distribution pattern can be specified by the programmer. There can be private
or shared pointers pointing to shared data. In UPC the programmer can choose the memory consistency model to be either strict or relaxed. Strict consistency model is same as sequential consistency model, wherein all the statements must be executed in program order. In the relaxed consistency model, the execution order can be changed. The relaxed model gives the UPC compilers to do more computation and communication overlap as described in [13].

### 2.2.1 Berkeley UPC

The Berkeley UPC compiler is divided into three main components:

1. UPC-to-C translator
2. UPC runtime system
3. GASNet communication system

In the first compilation phase the UPC code is converted into platform-independent C code with the remote communications replaced with runtime calls. This code is then compiled again with the target system’s C compiler and the runtime library is linked. The runtime system performs initiation tasks like thread creation and shared data allocation. The runtime functions invokes the GASNet library functions to do the low level network communication.
2.3 Remote Memory Access in MPI-2

The Remote Memory Access (RMA) API extends MPI communication to enable one process to specify all the communication parameters, both for the sending side and for the receiving side. This obviates matching operations and explicit polling. MPI RMA has 2 parts:

1. communication: MPI_Put, MPI_Get, and MPI_Accumulate;
2. synchronization: many different types of calls.

The correct order of memory access has to be imposed by the user using synchronization calls.

The RMA operation can be done only on a portion of memory which needs to be registered as a window. Every process in a communication group creates the window collectively using MPI_Win_create function to register portion of their memory before an RMA can be done. The window object contains the starting address and size of the registered memory on all processes. If the size is zero on a process, no memory is registered. The user can also specify a displacement unit greater than 1 to enable efficient byte displacement. An MPI window is logically divided into public and private copy. This is essential for systems where the network and host processor are not coherent. Remote accesses are performed on public copy, while local accesses are performed on the private copy. The copies are updated during synchronization operation, and either cannot be accessed when the other is locked.
All RMA communication calls must take place within an ‘epoch’, i.e., a start and end point defined by synchronization calls. The following are the 2 modes of RMA:

1. Passive target: This is truly one sided as the target does not have to invoke any function to receive the data. MPI one-sided communication is portable to non-cache coherent architectures, which increases its complexity. On a system where core-core cache coherency is weak, more than one passive memory access to the same cache line can cause incorrect result or memory corruption. Hence, even the one-sided passive \emph{puts} and \emph{gets} are invoked within a lock-unlock section (epoch). The lock marks the beginning of the epoch, and the unlock section ensures that the data transfer is complete. The lock can be exclusive or shared. A shared lock (as opposed to exclusive lock) can still cause data corruption if two processes write the same data simultaneously.

2. Active target: The target process also participates in the synchronization operation. It must invoke MPI\_Win\_post and MPI\_Win\_wait to mark the start and end of epoch corresponding to the MPI\_Win\_Start and MPI\_Win\_complete on the source process. A collective MPI\_Fence operation can also be used instead. This makes the active target process not entirely one sided, even though only one process passes all the communication arguments and no receive buffer is required.
Chapter 3

Enabling Remote Direct Memory Access (RDMA)

This chapter explains how direct access to remote memory is achieved, and how the PGAS memory is created.

Remote Direct Memory Access (RDMA) or one-sided memory access is to read or write memory on a remote processor without the participation of the remote processor. The problem with accessing memory on a remote processor without the help of its operating system is that the memory might not be physically available. Memory is swapped in and out due to virtual memory management and the network hardware must ensure that the memory is accessible to a remote process in order to enable RDMA. When the network hardware does not have such capability, CAF runtime implementation has to use two-sided message passing under the hood to provide the CAF programmer a semblance of one-sided remote memory operation.
On more sophisticated network interconnects, some form of RDMA is supported by hardware. These networks can be broadly divided into 2 categories [6]:

1. Automatic hardware-assisted: The NIC (Network Interface Card) acts like a paging-based virtual memory system. It initiates page faults and tracks changes in the page table by accessing the virtual memory subsystem of the operating system. It maintains a hardware TLB (Translation Look-aside Buffer) for quick virtual address translation. Even though the hardware complexity is increased manifold, the entire memory space becomes remotely accessible and RDMA operations are very fast. Example, Quadrics [31].

2. Passive pinning-based: The NIC has the ability to prevent the OS from swapping certain portions of the physical memory. This memory is called pinned memory. The OS does not swap out pinned-memory until the application exits or it is unpinned. In theory, the maximum memory that can be pinned is restricted to the total amount of physical memory, but in practice it is much more limited. For example, Infiniband has a limit on the number of contiguous memory pages that can be simultaneously pinned. The following techniques are commonly used to overcome this limitation:

(a) Rendezvous protocol: A request to pin a remote region is sent to the target processor. Once the initiator receives the reply confirming that the remote region is pinned, the one-sided operation can be initiated. The data transfer is asynchronous. The memory region is unpinned after the data transfer is completed. This method is also called on-demand dynamic
memory registration protocol [29].

Advantages: The resulting one-sided operation has good large message bandwidth and doesn’t interrupt the remote host processor.

Disadvantages: The rendezvous request does interrupt the remote host processor and each memory access effectively requires two network roundtrips (3 if the remote region must be unpinned). This is bad for a large number of small transfers.

(b) Bounce Buffers: One or more buffers are created during program initiation and their memory locations are pinned. The RDMA is done via these buffers.

Advantages: There is no registration cost at runtime and full memory space is accessible.

Disadvantages: It has local copy costs.

Every network interface has its own network programming API. It is cumbersome to develop different runtime libraries for different hardware. To simplify this work, ARMCI and GASNet libraries are used, which provide an abstraction over various network-specific APIs. Figure 3.1 shows the layered structure of UHCAF runtime.
Figure 3.1: GASNet/ARMCI library provides network-independent PGAS abstraction

3.1 GASNet

GASNet (Global Address Space Networking) [2, 10] provides a network independent abstraction for PGAS languages. It offers 2 levels of API:

1. Core API: The GASNet Core API provides initialization, termination and memory registration operations. The memory registration allows the user to attach a chunk of memory to be used for RDMA. Note that memory registration can only be done at the beginning of the program. For communication between different images it provides active message (AM) [39]. Active message is like a remote procedure call which must not block and must execute very quickly. A handler function is defined and attached on image A. In order for image B to send data to A, it needs to send an active message which contains the address of this handler (function pointer) along with other parameters and data. For receiving the AM, the following techniques are used depending on
the hardware support available:

(a) An extra thread (helper thread) polls the network.
(b) Hardware interrupts are used to interrupt the computation thread.
(c) Dedicated NIC processors run arbitrary handler code.

The handler function may send a reply message back. There are 3 types of active messages:

(a) Short AM: Short active messages contain integer arguments.
(b) Medium AM: Medium active messages contain a payload of data to be transferred to a temporary buffer.
(c) Long AM: Long active messages carry DMA transfer to a location specified by sender (requires hardware support).

2. Extended API: The GASNet extended API provides blocking and non-blocking memory-to-memory data transfer functions, notify-wait synchronization and barrier functions. The data transfer functions are one-sided *puts* and *gets*. On networks with automatic hardware assisted RDMA these functions directly map to the underlying network API, like portals in Cray Seastar2+ network. In a socket-based Ethernet network where there is no hardware support for RDMA, these extended API functions are implemented using active messages from the core API. On passive pinning-based networks like Infiniband, GASNet enhances the performance of the rendezvous protocol (described earlier) by using the firehose algorithm [6]. Firehose is similar to rendezvous, except that the memory is not unpinned after the data transfer is over. If the next access
falls in the same pinned region (likely due to spacial locality) the additional handshake is avoided. Additionally, the cost of synchronization and pinning is amortized over multiple remote memory operations. The handshake is done using Active messages. It has a flexible API and has low synchronization overhead. In the common case it is one-sided and zero-copy.

There are 3 type of GASNet configuration:

1. FAST: The amount of pin-able memory is smallest in this case and the performance is optimal. On networks with hardware support for DMA (like Quadrics), this is the amount of memory that the firmware on NIC(Network Interface Card) can directly register. On networks without hardware support, a reasonable portion of physical memory is pinned from the start of the program, and the user has to handle memory allocation.

2. LARGE: This configuration is same as fast, except that larger memory segment might be available for DMA. This configuration may be slower than FAST configuration.

3. EVERYTHING: The entire virtual memory space is available for DMA and all global variables are automatically accessible from other images. The user need to explicitly manage the heap. It is the slowest.

Note, however that the size of the memory segment available for DMA on FAST and LARGE can be changed by setting the environment variable GAS-NET_MAX_SEGSIZE.
Network specific implementation of GASNet is a conduit. GASNet (version 16.2) has optimized implementation for the following networks:

MPI – Message Passing Interface
DCMF – IBM’s Deep Computing Message Framework over BlueGene/P
ELAN – targets the Quadrics libelan software, a low level interface which is implemented on top of the hardware-level elan3/4 library
GM – Myrinet
IBV – Infiniband IBVerbs/OpenIB API from the OpenFabrics Alliance
VAPI – Infiniband “Verbs API” from Mellanox
LAPI – IBM LAPI communication protocol
PORTALS – Cray XT3/XT4
SHMEM – SGI and CrayX1
SMP – single node shared memory processor

GASNet API is rapidly growing; its API source code has many other functions like non-contiguous data transfers, broadcasts and reduce which are not included in the specification. For efficient non-contiguous data communication, the environment variable GASNET_VIS_AMPIPE must be set. The impact of not setting this flag is demonstrated in table 6.1 in chapter 6.

GASNet is widely used in runtime implementation of PGAS languages and libraries like Berkeley UPC [12] and OpenSHMEM (at University of Houston and University of Florida).
3.2 ARMCI

Like GASNet, ARMCI (Aggregate Remote Memory Copy Interface) [29, 1] is a portable library which enables remote memory access operations. It provides blocking and non-blocking memory-to-memory data transfer functions for both contiguous and non-contiguous (vector and strided) data. It provides vectored, strided and contiguous accumulate operations, synchronization and barrier functions. ARMCI also provides atomic read-modify-write, broadcast, and reduce operations. ARMCI implicitly aggregates small messages into large messages. Unlike GASNet, it has memory management functions to register memory any time in the program. However, ARMCI does not support active messages and support for Global Procedure Call (GPC) is rudimentary. ARMCI has optimized implementation for the following networks:

1. Portals (SeaStar2+ interconnect on Cray XT5)
2. Myrinet (GM)
3. Quadrics
4. Infiniband (using OPENIB and Mellanox verbs API)
5. Ethernet

ARMCI acts as a thin wrapper around the network API to implement functions that are supported by the hardware. For example, on Cray ARMCI acts as a wrapper for contiguous RDMA reads and writes, which are directly mapped to Portal API calls [36]. For other calls like non-contiguous write for which there is no direct support from Portals, ARMCI has to do some extra work. For some networks like
Ethernet, there is no hardware support for RDMA. In such case, ARMCI has to implement one-sided remote memory access on its own. On other networks, like Infiniband, only remote write is supported by the hardware but remote read is not. In order to provide uniform capability across all network interconnects, ARMCI uses a Communication Helper Thread (CHT), also called server thread. There is one helper thread associated with every shared memory. These threads communicate with each other to provide PGAS functionalities. The threads are spawned using fork(), clone(), or pthreads library on an idle core if one is available. If none of the cores are idle, glibc sched_setaffinity call is used to schedule the thread appropriately. The threads use blocking wait instead of active polling to reduce CPU usage. Active polling can be enabled by setting ARMCI\_POLLING\_RECV while building ARMCI on GM or VIA. The following techniques are used to enable various PGAS functionalities with the helper thread [29]:

1. Host-based/buffered baseline and pipelined protocols:

   This is based on the bounce-buffers protocol described earlier. There is at least one pre-allocated buffer for each image. The helper thread moves data to and from these buffers. For example, a put request without network RDMA support is implemented as follows:

   (a) The helper thread on the source image copies the data along with address information into its buffer.

   (b) The helper thread on the source image transmits the packet to the destination image (e.g. MPI\_send) .

   (c) The helper thread on the destination image receives the message and
copies the packet into its buffer.

(d) The helper thread on the destination image sends an acknowledgement to the source helper thread.

(e) The helper thread on the destination image copies the data to the specified memory location.

This procedure can be pipelined if there is more than one buffer per image. A similar technique is used for non-contiguous transfers, etc. If the data size is larger than buffer size, it is broken into multiple chunks. Hence, it is preferable to have a large buffer size and many buffers available. However, to prevent the total buffer size from getting very large on a many-core system, the buffer size is restricted to less than 5% of the node memory. To avoid compromising performance, flow intimation is used where a spare buffer is used for trailing blocks of data. These blocks are sent without additional header information as the first block was already sent and the receiver knows the number of remaining chunks that are expected to arrive.

2. Rendezvous protocols (on-demand memory registration):

On networks like Infiniband, GM, and VIA, the cost of registering memory is high and the amount of memory that can be registered is limited. For message sizes where registration is cheaper than copying the data, ARMCI switches to a protocol where the memory is registered/unregistered dynamically. The following steps describe a put of contiguous block of data:

(a) The helper thread on the source image registers its local memory and
sends a message to the helper thread on the remote image.

(b) The helper thread on the remote image attempts to register memory.

(c) If registration is successful, RDMA occurs.

3. Zero-copy host-assisted protocols [37]:

   This is another way to avoid using the buffer. In many networks, RDMA write is supported but read is not. To implement get operation on such networks, the following protocol is used:

   (a) The image doing the get sends a request to the helper thread on the remote image.

   (b) The helper thread on the remote image does a RDMA put.

   Thus the get is converted into a put call. Similar non-contiguous RDMA calls are converted to scatter-gather calls on networks without RDMA support for non-contiguous memory operations.

   ARMCI automatically detects whether remote images reside on the same node or not. If on the same node, it reads from the shared memory instead of initiating a RDMA call. ARMCI uses a location consistency model for data transfers [21]. In this model, a process observes its own operations with respect to a given target process in the order in which they are completed locally. Another process accessing the same data may observe a different ordering of these operations.
3.2.1 ARMCI over One-sided MPI

ARMCI uses MPI 2-sided send-receive calls to implement RDMA on Ethernet network and on networks for which it has no specialized support. This is implemented using helper threads which use an extra core. Instead of using send-receive, [15] demonstrates using MPI one-sided put-get which is available in MPI-2 and supported by most network hardware. This version of ARMCI is available at [4]. One-sided MPI makes ARMCI much more portable, and also removes the overhead of helper threads. However, one-sided MPI is slower than using network specific API calls. MPI-3 proposes several extensions to improve performance of these communications.

As described in section 2.3, MPI one-sided communication needs to set up a memory window, and the RDMA address must reside in that window. ARMCI uses a Global Memory Region (GMR) translation layer which converts absolute ranks to ranks within MPI window groups and absolute addresses to offsets from the start of the window. MPI assumes a non-cache coherent system, and adds extra overhead to prevent memory corruption. This includes locking and unlocking before RDMA when using passive puts and gets. GMR does the lock management using MPI exclusive locks. This ensures that operations issued within the same epoch are non-conflicting and operations issued by different sources using a shared lock are also non-conflicting. The local source for a put and the local destination for a get can be part of an MPI_Window. Therefore, their access must also be protected by locks. This can result in deadlock when both the source and destination are part of the same window. To avoid such deadlocks, ARMCI-MPI uses a separate temporary local buffer. In MPI, communication operations are not left in flight as it must
complete before the window is unlocked. Hence ARMCI_Fence operation is a no-op. As of MPI 2.2 there is no support for atomic read-modify-write operations which are required to implement ARMCI_Rmw. Doing a consecutive read and write to the same location within an epoch is also forbidden. To solve this problem, ARMCI-MPI uses two different epochs and ensure atomicity by using mutexes. The Table 3.1 demonstrates the basic mapping between ARMCI and MPI one-sided API.

<table>
<thead>
<tr>
<th>ARMCI Functionality</th>
<th>ARMCI-MPI Implementation</th>
</tr>
</thead>
<tbody>
<tr>
<td>ARMCI_Malloc</td>
<td>MPI_Win_create</td>
</tr>
<tr>
<td>ARMCI_Free</td>
<td>MPI_Win_Destroy</td>
</tr>
<tr>
<td>ARMCI_Get, ARMCI_Put</td>
<td>MPI_Win_lock, MPI_Get/ Put, MPI_Win_unlock</td>
</tr>
<tr>
<td>ARMCI_Fence</td>
<td>n/a as MPI_Win_unlock ensures completion</td>
</tr>
<tr>
<td>ARMCI Mutexes API</td>
<td>MPI Mutexes API</td>
</tr>
<tr>
<td>ARMCI_Rmw</td>
<td>MPI_Mutex_lock, get, modify, put, MPI_Mutex_unlock</td>
</tr>
</tbody>
</table>

Table 3.1: ARMCI functions mapped to MPI one-sided functions

ARMCI strided operations are directly mapped to MPI subarray derived datatypes. ARMCI-MPI uses three methods to implement noncontiguous data transfer using generalized IO vector (IOV):

1. Conservative method: uses one RMA communication operation per segment, each in a separate RMA epoch.
2. Batched method: uses $n$ operations per epoch where $n$ is configurable. It requires that all segments be in the same GMR and do not overlap.
3. Derived datatype method: generates two MPI indexed datatypes to represent the data layout at the source and the destination. A single communication operation is issued using these datatypes. This method also requires that all segments be in the same GMR and do not overlap.
3.3 GASNet & ARMCI Comparison

The overall performance of ARMCI and newer versions of GASNet are quite similar for the Infiniband interconnect and SMP. Performance evaluation on other interconnects could not be performed as they are not available. Even though the performance of ARMCI and GASNet is similar for most cases, there are differences for certain type of communication pattern. For example, GASNet can achieve better performance for noncontiguous data when the flag GASNET_VIS_AMPIPE is enabled. Remote reads performs better in ARMCI as it is implemented using the helper thread which converts the get operation into a put operation which is hardware assisted on Infiniband.

On hardwares without RDMA capabilities, GASNet uses active messages to provide one-sided memory access abstraction to the user. ARMCI uses helper threads to do the same. However, GASNet exposes the use of active messages to the user which can be useful to implement many things which are not supported by GASNet. In ARMCI, there is no support for active messages.

While using GASNet, shared memory must be created during program initialization, whereas ARMCI can create it whenever requested. The maximum memory that can be safely ARMCIMalloced on Infiniband is limited to around 2GB. For many applications (as described in section 6.2) 2GB is not enough. GASNet also restricts the maximum pin-able memory to a conservative value, but it can be changed using the environment variable GASNET_MAX_SEGSIZE. Therefore, the entire physical memory space can be pinned down when using GASNet.
In ARMCI, non-blocking communication is different from GASNet. ARMCI_Wait only ensures local completion. Only ARMCI_Fence guarantees remote completion. However, ARMCI_Fence can only be invoked for an image and not for a specific non-blocking handle. In GASNet, there are 2 sets of non-blocking functions: one which does not distinguish between local and remote completion, while the other guarantee local completion as soon as they return. The second feature is not available for strided/vectored/indexed operations yet.

### 3.4 Problem with using MPI as the Communication Layer

MPI is the most widely available and portable software interface for programming on distributed memory machines. However, both MPI 1.1 and 2.0 are inadequate for implementing GAS languages [8]. When using two-sided communication, latency for short messages is much higher than native APIs and additional communication is required to set up a receive call. Active target RMA operation is not truly one-sided as the target process must invoke synchronization operations as described in section 2.3 on chapter 2. This destroys the possibility of implementing RDMA without any helper thread or remote procedure call. Passive target communication has the following issues:

1. The memory of only 1 process can be accessed during an epoch. This will cause
serialization or creation of separate windows for each process in case of a collective operation like broadcast. Moreover, the local process also cannot make changes to any memory location in the window within the epoch. This necessitates some kind of synchronization between the source and target process, and makes the transfer two sided. The other alternative is to use temporary buffers as described in subsection 3.2.1.

2. Since the RMA operation must finish by the end of the epoch, it is impossible to implement non-blocking communication.

3. A separate window can be created for each allocatable coarray, but to implement save coarrays a large window is required to coalesce them together. However, MPI-RMA restricts concurrent update of the same window within the same epoch, even though the access is on a different memory location. This will cause severe limitations.

4. The MPI spec [3] says that implementors may restrict RMA operations only on memory allocated using MPI_Alloc_mem. A large amount of memory allocation may fail as implementors might restrict the size to the maximum pinable page size provided by network hardware.
Chapter 4

CAF Runtime Implementation

This chapter describes how the PGAS memory is managed and how the CAF features are implemented in the runtime library.

4.1 Overview of Previous CAF Runtime

The CAF runtime was previously implemented by Hyoungjoon Jun as described in [20]. However, the implementation was redone due to change of design. Earlier, every coarray was treated as a dope/codope pair as shown in the Figure 4.1. This added unnecessary complication and carried a lot of information that was not really needed. In the new implementation, coarrays are represented just like a normal array using only a dope vector. There is an extra flag in the dope vector indicating whether it represents an array or a coarray. Another change in design was the allocation of symmetric shared memory. Earlier, it was done per coarray basis and
their communication handles were tracked individually. In the new implementation, a large chunk of memory is allocated at the start of the program and managed by the runtime as described in section 4.3. This is necessary for using GASNet more efficiently and for using the optimizations described in section 5.1.

![Figure 4.1: Dope and codope vectors (cited from Jun’s thesis)](image)

### 4.2 CAF Runtime Structure

CAF runtime library is composed of 2 layers:

1. Compiler interface layer: This layer contains functions which are invoked by a CAF executable. The compiler inserts functions like `remote_read`, `remote_write`, coarray allocation, synchronization and image inquiry functions in the executable file. These functions are defined in `caf_rtl.c` in the folder `osprey/libcaf` in the source code.

2. Communication layer: Our runtime uses ARMCI/GASNet library to enable
remote memory access. In order to implement many functions in the compiler-interface layer like remote_read/remote_write, ARMCI/GASNet library functions are required to be invoked. These invocations are defined in this layer. This layer has separate code for ARMCI and GASNet. One of these is linked depending on the layer that is being used. If the compiler has both ARMCI and GASNet enabled, the user can choose which one to use by passing layer=armci/gasnet parameter while compiling the CAF program. This layer is implemented in armci/armci_layer.c and gasnet/gasnet_layer.c in the osprey/libcaf folder of the source code.

The Figure 4.2 describes how the compiler invokes the runtime library functions.

![Diagram](image)

**Figure 4.2**: Code structure
4.3 Shared Memory Management

In this section, the term ‘shared memory’ is used in the PGAS sense. That is, the memory may not be physically shared, but it must be registered/pinned-down for direct memory access and the operating system must not swap out the memory.

During program initialization, the communication layer creates a big chunk of shared memory. Static coarrays are allocated memory from this chunk (except for GASNet ‘everything’ configuration as static variables have same address). The remaining memory is left for:

1. allocable coarrays
2. pointers in coarrays of derived datatype (henceforth referred as asymmetric data).

The structure of the memory heap is shown in Figure 4.3.
Normal Fortran allocation calls are intercepted to check whether they are for coarrays or asymmetric data. If they are for coarrays, the coarray allocation function is invoked. Since allocatable coarrays must have symmetric address, a separate heap must be created for asymmetric data. To avoid wasting memory by statically reserving separate memory for asymmetric data, a single memory block is used to store both allocatable coarrays and asymmetric data. The top of the heap is used for allocatable coarrays (which grows downward) and the bottom of heap for asymmetric data (which grows up) as shown in Figure 4.3. A linked-list of struct shared_memory_slot (shown below) is used to manage allocation and deallocation:

```c
struct shared_memory_slot{
    void *addr;
}
```
unsigned long size;
unsigned short feb; //full empty bit. 1=full
struct shared_memory_slot *next;
struct shared_memory_slot *prev;
};

The common_slot is an empty slot which always lies in between allocatable heap and the asymmetric heap, and used by both to reserve memory. Figure 4.4 demonstrates how a coarray and asymmetric data are allocated from the common-slot.

![Diagram](image)

**Figure 4.4:** Creation and management of a 500bytes shared memory segment

The allocatable heap consumes the common_slot from the top, while asymmetric heap consumes it from the bottom. Each allocation address and size is stored in a separate slot (node in the list). Each slot has a full-empty bit(feb). During deallocation the full-empty-bit is set to 0 (empty). If any neighboring slot is empty, they are merged. This makes the common-slot grow when a slot bordering the common-slot is deallocated. If there is no more space left in the common slot, empty slots are used
from above for allocable coarrays or from below for asymmetric data. This prevents the 2 heaps from colliding/mixing. Figure 4.5 demonstrates how the deallocation is done. During exit, the function `coarray_free_all_shared_memory_slots()` is used to free all nodes in the shared memory list.

The size of the symmetric memory heap is 30MB by default. It can be changed during execution using the cafrun parameter ‘shared-memory-size’. It can also be changed by setting environment variable `UHCAF_SHARED_MEMORY_SIZE`. All values must be in bytes.

e.g (for 2GB shared memory):

export `UHCAF_SHARED_MEMORY_SIZE`=2000000000.

There is a problem with this approach which is demonstrated in the following case study:

Consider the case when, there are many asymmetric data allocation. Let’s say it eats up 90% of the shared memory space. Subsequently these are deallocated except for the slot immediately below the common-slot. So the common-slot remains on top,
reserving 90% space for asymmetric data, even though most of the slots are empty. Subsequent coarray allocation may fail as it cannot use memory from below the common slot. To solve this problem, we might have to de-fragment slots. However, this is very unlikely as the slot bordering common-slot will be the latest slot and will get implicitly deallocated when the subroutine returns.

The following are the differences when using “GASNET EVERYTHING” configuration:

1. The entire memory is registered, so the runtime can directly invoke GASNet RDMA functions for any address in another image.
2. Global/save coarrays have same address on all images. They are not allocated in shared memory heap. Therefore, the entire shared memory size is used only for allocatable coarrays and asymmetric data.

### 4.4 Remote Reads and Writes

For remote read/write, our runtime uses ARMCI/GASNet `get` and `put` calls. The remote address is calculated as below:

\[
\text{remote}\_\text{addr} = \text{remote}\_\text{start}\_\text{address} + \text{offset}.
\]

where \( \text{offset} = \text{local}\_\text{address} - \text{local}\_\text{start}\_\text{address} \).

All remote start addresses are stored on all images during init (except when using GASNet everything config, discussed later). Table 4.1 shows the mapping of CAF read/write to the underlying function calls in ARMCI and GASNet.
Table 4.1: GASNet & ARMCI functions invoked for reads and writes

<table>
<thead>
<tr>
<th>Operation</th>
<th>GASNet Function</th>
<th>ARMCI Function</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>Contiguous Read</td>
<td>gasnet_get</td>
<td>armci_get</td>
<td>a=b[2]</td>
</tr>
<tr>
<td>Contiguous Write</td>
<td>gasnet_nb_put</td>
<td>armci_put</td>
<td>b[2]=a</td>
</tr>
<tr>
<td>Strided Read</td>
<td>gasnet_gets_bulk</td>
<td>armci_gets</td>
<td>a(:,:,)=b(7:8,3:4)[2]</td>
</tr>
<tr>
<td>Strided Write</td>
<td>gasnet_puts_bulk</td>
<td>armci_puts</td>
<td>b(7:8,3:4)[2]=a(:,:,)</td>
</tr>
</tbody>
</table>

The following are the differences when using “GASNET EVERYTHING” configuration:

1. If the remote read/write is to a static coarray, the address will be the same on remote image. Hence the address is checked to see if it is in the shared memory segment. Otherwise, no address conversion is required.

2. During the program initialization, the start address of remote images are not collected. It is fetched from a remote image during the first access to that image. Therefore, a remote start address may not get fetched at all if there is no communication between them. This can benefit performance. This is possible only on ‘everything’ configuration because static variables have same address on all images. The local start address is stored in a static variable which is directly accessed by other image.

4.5 Synchronization

There are 2 kinds of synchronization operation in CAF: ‘sync all’ and ‘sync images’. ‘Sync all’ is a global barrier which can be directly mapped to global barrier functions provided by GASNet and ARMCI.
In GASNet, ‘sync all’ is implemented by invoking the following functions:

```c
gasnet_wait_synenbi_all()
gasnet_barrier_notify(barrier-count, barrier-flag)
gasnet_barrier_wait(barrier-count, barrier-flag).
```

In ARMCI, it is implemented as:

```c
ARMCI_WaitAll()
ARMCI_BARRIER().
```

‘Sync images’ is a barrier for a subset of images. It can be 1-to-1 or many-to-1 or many-to-many. Since GASNet and ARMCI does not have any function corresponding to ‘sync images’, it is implemented in the UHCAF runtime as described below:

1. GASNet: During program initialization, an array of flags (indexed by images) is created and initialized to 0. These flags are incremented by a handler function which can be invoked by an active message from a remote image. When a sync image statement is encountered, active messages are sent to the image-list in the argument. Corresponding active messages are expected to arrive from other syncing images which increment their corresponding flag on this image. This image blocks until the flag is greater than zero. Then it decrements the flag in a thread-safe way. It is necessary to increment/decrement the flag instead of using a full-empty bit because imageX can receive more than 1 active message from imageY. The pseudo code for the implementation is shown below:

```c
Function comm_sync_images ( image_list[] )

    /* Loop to increment my_image flag on remote image */
```
LOOP each remote_image in image_list
    /*Send Active message to remote image*/
    gasnet_AMRequestShort1(my_image, ...);
END LOOP

/* Loop to wait */
LOOP each remote_image in image_list
    /* Wait till remote image flag is greater than 0 */
    GASNET_BLOCKUNTIL(remote_image_flag);
    /*Handler safe lock*/
    gasnet_hsl_lock( &sync_lock );
    remote_image_flag--; /* Decrement sync images flag */
    gasnet_hsl_unlock( &sync_lock );
END LOOP
END FUNCTION

Handler function invoked by ACTIVE MESSAGE( int image )
    gasnet_hsl_lock; /*handler safe lock*/
    invoking_image_flag++;
    gasnet_hsl_unlock;
END Active message handler

2. ARMCI: Unlike GASNET, ARMCI does not support active messages. In order
to increment sync-flags on a remote image, atomic read-modify-write function (ARMCI_Rmw) is used. The pseudocode for the implementation is shown below:

Function comm_sync_images (image_list[])

/* Loop to increment my_image flag on remote image */
Loop each remote_image in image_list

/* complete all communications to remote image first */
ARMCL_Fence(remote_img);
/* Lock mutex_id ‘my_img’ on remote_img */
ARMCL_Lock(my_img, remote_img);
/* Atomically increment my_img flag on remote_img */
ARMCI_Rmw (ARMCI_FETCH_AND_ADD...);
/* Unlock mutex_id ‘my_img’ on remote_img */
ARMCL_Unlock(my_img, remote_img);
End Loop

/* Loop to wait */
Loop each remote_image in image_list

/* user usleep to wait at least 1 OS time slice before checking flag again */
Loop until remote_img flag on this image >0
usleep(50);
End Loop

/* Lock mutex_id ‘remote_img’ on my_img. This ensures that my_img...
is not decrementing the flag while remote_img is incrementing it*/

    ARMCI_Lock(remote_img, my_img);

    /* decrement remote_img flag on this image, dont just make it 0,
    maybe more than 1 sync_images are present back to back*/

    flag--;

    /* UnLock mutex_id ‘remote_img’ on my_img */

    ARMCI_Unlock(remote_img, my_img);

End Loop

END FUNCTION

4.6 Image Inquiry

Image inquiry functions are used to get information about the logical location of an image on the topology. Table 4.2 shows the result of different image inquiry functions on a coarray A(10)[2:2,3:4,0:*] on 6 images. Since these functions do not require any remote communication, they are implemented in the CAF runtime layer without any GASNet/ARMCI calls.
Table 4.2: Image inquiry function results for A(10)[2:2:3:4:0:*] on 6 images

<table>
<thead>
<tr>
<th>Function</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>image_index(a,(/2,4,0/))</td>
<td>2</td>
</tr>
<tr>
<td>this_image()</td>
<td>image id</td>
</tr>
<tr>
<td>this_image(a) on img6</td>
<td>2 4 2</td>
</tr>
<tr>
<td>this_image(a,2) on img6</td>
<td>4</td>
</tr>
<tr>
<td>lcbound(a)</td>
<td>2 3 0</td>
</tr>
<tr>
<td>ucbound(a)</td>
<td>2 4 2</td>
</tr>
<tr>
<td>lcbound(a,3)</td>
<td>0</td>
</tr>
<tr>
<td>ucbound(a,3)</td>
<td>2</td>
</tr>
</tbody>
</table>

4.7 Wrappers to Compile and Execute CAF Programs

CAF programs are compiled using the ‘uhcaf’ wrapper program. For example, ‘uhcaf HelloWorld.f90’ will produce an a.out executable. The user can use the flag ‘–layer’ to choose either ARMCI or GASNet library. GASNet is used by default when the flag is not used. All the uhf90 options can be used on this wrapper. The command ‘uhcaf -h’ shows all the available options.

CAF programs are executed using ‘cafrun’ wrapper program. For example, ‘cafrun -np 4 ./a.out’ runs a.out using 4 images. When using GASNet SMP conduit, the program should be invoked using ‘cafrun –use-smp –num-pshm-nodes=n ./a.out’. The maximum size of coarrays per image is 30MB by default. This can be changed by using the flag ‘–shared-memory-size’. Other options like enabling traces, etc. are also available. All mpirun options can be used as well, when not using GASNet SMP conduit.
4.8 Trace

Tracing can be enabled by using the cafrun parameter ‘log-levels=<levels ’. The following levels are available:

1. TIME: Both CPU time and wall-clock time can be obtained for every runtime function call. The function clock() (from time.h) is used to measure CPU time and gettimeofday(from sys/time.h) is used to measure wall-clock time. The TIME trace output for a small program with a strided write operation and a barrier is shown in Figure 4.6. This information can be useful to identify performance bottlenecks in a program. When non-blocking writes are enabled, the time taken by a write operation is not accurate as it only provides the time for local completion.

![Figure 4.6: Screen shot of time-trace output](image)

2. TIME_SUMMARY: Time-summary prints the aggregate time taken for program initialization, remote reads, remote writes and synchronization. Figure 4.7 shows a screen-shot of time-summary trace output.
3. MEMORY: The memory trace prints the memory addresses after allocation and deallocation. It also prints details about the memory management data-structure described in section 4.3. Figure 4.8 shows a screen-shot of memory trace for image 1 allocating and deallocating a coarray.

4. DEBUG: The debug option prints information about every runtime function invocation. It is useful to detect race conditions and other programming errors. Figure 4.9 shows a screen-shot of debug output for image 1 of a small program with just a remote read. The first line prints IMG0 instead of IMG1, because
that line is printed even before the image identification is done.

![Screen shot of debug output](image)

**Figure 4.9:** Screen shot of debug output

When tracing is not enabled, the functions are defined as empty functions.

### 4.9 Correctness Test Suite

A test suite is created which comprises of many small programs which verify the different syntaxes of CAF. It can be compiled and executed using a shell script. The output of the programs are checked against the expected output to ensure correctness. This can be useful to verify proper installation of UHCAF.
Chapter 5

CAF Runtime Optimizations

This chapter describes the optimizations implemented in the runtime library. These optimizations are implemented with no compiler support and can be enabled during the program execution.

5.1 Get Cache Optimization

As described in section 4.3, a large chunk of pinned memory is created during program initiation, and coarrays are allocated memory from this memory space. This results in coarrays having contiguous addresses suitable for exploiting spacial locality. In the get-cache optimization, a cache is created during program initialization. The data structure of the cache is shown in Figure 5.1. On the first read operation, instead of fetching the required size the whole cache line is fetched. On the subsequent read operations the cache is checked to see if the data are already present. If the data
are found in the cache, the remote read operation gets converted to a local read from the cache. This can save a lot of time on interconnects with significant network latency. Moreover, for strided operations, local strided copy is many times faster than strided remote read. If the data are not in the cache, the corresponding cache line is re-fetched. There are some exceptions to this process. If the required data size is more than the cache line size, or the data are from the end of the pinned memory segment, the cache is left unchanged.

![Figure 5.1: Get cache data structure](image)

When a remote image writes to an address which is in the local cache, the data in the cache becomes stale. However, as per the language specification, the programmer is guaranteed to see any remote changes only after a synchronization. Therefore, data consistency will be maintained if the data in the cache is deleted at synchronization points. To increase the chances of a cache hit and to overlap communication
with computation, the cache is populated again immediately after the synchroniza-
tion using non-blocking get. This is somewhat similar to the pre-fetch optimization
described in the next section.

When the local image writes to a remote address which is present in its cache,
the cache must be updated immediately. Hence, any remote write has the overhead
of checking the cache and if conflicting address is present, a cache update.

This optimization can be enabled during the runtime. The default cache line size
is 64 kilobytes. It can be modified by the user at runtime.

Figure 5.2 shows that the optimization provides good performance benefit when
the cache is well utilized. The application used to evaluate the performance gain has
the following structure:

Loop 200 times

26 Gets (from different coarrays)

Barrier

Computation

Loop.

The experiment was performed on an Infiniband cluster with each node having dual
core AMD Opteron processors, and each core operating at a frequency of 2.4 GHz.

The computation phase takes around 0.1 seconds. Initially the communication
time in the figure is less than 0.05 seconds when the cache is used. Since there are
26 coarrays, the total cache size required for storing all the coarrays together has to
be more than 64KB if the coarray size exceeds 2.5KB. When the cache size is more
than 512KB, the communication time increases suddenly as the buffer size becomes more than the maximum available bandwidth. When the cache size exceeds 1 MB, it is faster to do 26 gets of 32KB than 1 get of 1MB and 25 local copies.

As a future optimization, the number of cache line can be increased and a hash map can be used to map memory addresses to cache lines.

5.2 Get Pre-fetch Optimization

Unlike the previous optimization, the get pre-fetch does not deal with data locality. Instead it targets a special kind of communication pattern and tries to overlap communication and computation using non-blocking gets. The target communication
patterns for this optimization are iterative algorithms which fetch the same data over and over again. For example, a picture smoothing program modifies each pixel based on the value of its neighboring pixels iteratively. When the bitmap is divided for processing in parallel, the halo cells are exchanged after every iteration. The pattern is usually similar to the following:

```
Loop
  Synchronization
  Computation
  Remote read
End Loop
```

This kind of application has a very predictable communication pattern which can be exploited to overlap communication with computation.

As described in [13], this optimization does pre-fetching. All remote reads are recorded in a data structure shown in Figure 5.4. This information is used to pre-fetch data into a get-cache at synchronization points. This does not effect program correctness because any write operation is guaranteed to be seen by other images only after a synchronization point. Synchronization points are used to divide the application into phases. At the end of a phase (at a synchronization point), the cache is cleared and non-blocking gets are invoked for the data in the pre-fetch list. The handles of these gets are populated in the cache. These handles are waited upon when a get finds its match in the cache. After applying the optimization, the program pattern described earlier will be as below:
Loop

Synchronization

Pre-fetch (non-blocking) into cache

Computation

Local Copy (if cache hit)

End Loop

The data structure of the get-cache is shown in Figure 5.3.

**Figure 5.3:** Pre-fetched data storage
Remote reads in CAF are blocking. Unlike writes, they cannot be directly converted to non-blocking as the correctness of the program will be affected. This optimization will improve performance of applications with iterative computations because the blocking read operations are converted into non-blocking reads. The cache clean-up is done within the notify and wait [2] of the synchronization operation to prevent overhead.

On the Infiniband cluster, this optimization failed to provide any performance benefit as shown in the Figure 5.5. However, on networks with high network latency, this optimization can be beneficial (refer [13]). As a future enhancement, compiler analysis can be used to identify applications with iterative computation and this runtime optimization can be automatically enabled only for those kind of applications.
5.3 Non-blocking Write Optimization

The CAF compiler can make the remote write operations nonblocking to achieve communication computation overlap. However, the compiler does not have remote address information as it is determined only after the program starts execution. This restricts the ability of the compiler to perform accurate blocking-to-nonblocking conversion. It is much more convenient to do this conversion in the runtime, as described in [13]. Conversion of the remote writes to non-blocking operations is permissible only when the writes complete before the following program points:
1. **Local variable overwrite**: The local variable which was the source of the remote write gets modified, e.g., \( A[2] = b; b = 2 \). Checking for local variable is not feasible in the runtime. ARMCI provides ARMCI_Wait functions which wait for local completion. GASNet provides a non-blocking write function `gasnet_put_nb` which blocks until the operation is completed locally (as opposed to `gasnet_put_nb_bulk`). For strided or vectored writes, GASNet has no such equivalent function. A temporary local buffer is used as the source of the remote write in case of strided writes.

2. **Read conflict**: A remote read operation accesses an address that was previously written by the same image; e.g., \( A[2] = 1; b = A[2] \). The remote completion is ensured by waiting on handles (GASNet) or using a fence operation (ARMCI).

3. **Write conflict**: A remote write operation targets an address that was previously written by the same image; e.g., \( A[2] = 2; A[2] = 4 \). This is a problem because non-blocking writes do not guarantee order.

4. **Synchronization event**: All the pending writes must be completed during synchronization events like ‘sync all’ and ‘sync images’.

Invoking a non-blocking write in GASNet returns a handle which can be used to wait for the write operation to complete. The handles for the non-blocking writes are stored in an array of linked-lists as shown in Figure 5.6. The tuple (handle, address, size) are indexed by destination image so that the conflict check will be performed only against writes to the same image. The list stores the operations in FIFO order as an older write is likely to complete first.
Figure 5.6: Non-blocking data structure

The minimum and maximum address on which there are pending *puts* are also stored in a separate array. This array is checked on every remote read or write to see if the remote address is within the range of the pending writes. This step reduces the overhead of searching through the entire linked list for every remote operation. For programs that access remote memory monotonically, the overhead is reduced to $O(1)$ complexity. Only if the address is within the range, the linked-list (in Figure 5.6) is searched to see if the address has a pending *put* to it. Once found, the search is stopped and the pending write is waited upon. While searching the list, the pending *put* status is checked (using *gasnet_try_syncnb*) to see if it has completed. The node is removed from the linked list if the *put* has completed. This helps in keeping the list short to reduce the traversal time.
ARMCI does not provide any function to wait on remote completion for individual writes handles. ARMCI_Fence is used to wait on all writes to an image. Hence, the data structure of Figure 5.6 is not used for ARMCI, and there is less scope for effective overlap of computation with communication.

This technique might not benefit programs which have much less scope for overlapping computation and communication. This can be eliminated by compiler analysis to determine whether a particular write should be made non-blocking.

Figure 5.7 shows that the optimization provides some performance benefit on an Infiniband cluster with each node having dual core AMD Opteron processors, and each core operating at a frequency of 2.4 GHz. The application used to evaluate the performance gain has the following structure:

Loop 200 times

26 Puts (each with buffer size in fig)

Computation

Barrier

Loop.

The computation phase takes around 0.1 seconds and hence there is scope for overlapping the communication with computation. For a buffer of 4 bytes, the communication time is too small to see any significant performance gain. For 256 bytes and 1 KB, the improvement is less than 0.1 seconds. As the message size increases, the performance gain also increases as the network bandwidth is better utilized. However, for a buffer size of more than 64 KB, the performance gain reduces, and at 256 KB there is a performance penalty. This is because Infiniband has a limit on the
number of contiguous memory pages that can be simultaneously pinned, and buffers large message transfers (see Chapter 3).

![Figure 5.7: Execution time difference with non-blocking optimization](image)
Chapter 6

Performance Evaluation

This chapter presents the experimental results and performance evaluation of UH-CAF using microbenchmarks and real applications.

6.1 Microbenchmarks

The microbenchmark suite has the following tests:

1. Put & Get Latency
2. Put & Get Bandwidth
3. Bidirectional Bandwidth
4. Noncontiguous Bandwidth
5. Broadcast Bandwidth
It has equivalent codes for CAF, ARMCI, GASNet, UPC, Global Arrays, and MPI (1-sided). It is an extended version of the microbenchmarks created by Asma Farjallah while working as an intern at Total. The original version had latency and bandwidth test for CAF and MPI. The CAF tests in current version is same, but the MPI tests are changed from two-sided to one-sided.

The experiments are performed on a cluster of dual core AMD Opteron systems, with each core operating at a frequency of 2.4 GHz. The nodes have 2011MB memory and runs openSUSE 11.3 Linux operating system. The nodes are connected with Infiniband interconnect. Each core is equipped with separate L1 instruction and data cache of size 64KB and a unified L2 cache of 1MB size. It has 64bit registers. The topology is described in Figure 6.1.

![Figure 6.1: Topology of each node on cluster](image)
Note that only 1 process per node is used in the experiments, which leaves the other core free. This free core can be useful in some cases. For example, in ARMCI the helper thread is automatically scheduled on the free core.

The experiments are also run on an SMP. It is a NUMA AMD Opteron system with 48 cores. It has 64GB memory and runs openSUSE 11.3 Linux operating system. Each core operate at a frequency of 800MHz. Each node has 4 sockets. The architecture of each socket is described in Figure 6.2.

![Figure 6.2: Topology of a socket on the SMP](image)

**Time Measurement**: Assembly code is used to read the value of the register which stores the number of ticks since reset. The clock resolution is measured by calculating the number of ticks for 1 second sleep. Separate version of the code is available for X86, X86_64, IA64, and PPC64 architectures, which is chosen by setting an environment variable TIMER_ARCH. The user can also choose gettimeofday by
setting TIMER_ARCH to NONE.

The compiler/library versions used for our experiments are OpenMPI 1.4.3, Berkeley UPC 2.12.2, ARMCI 1.4, GASNet 16.2, Global Arrays 5.0.2, gcc 4.5.0, and Intel cluster studio 2011. UHCAF-GASNet uses the ‘fast’ configuration for all tests. All tests are compiled with -O3 optimization flag.

6.1.1 Latency

In this experiment, the time taken to transfer an integer from one image to another is measured. An average of 1000 iterations is recorded. The latency measured on a cluster is shown in the Figure 6.3. The experiment is run on 2 nodes, with 1 process on each.

Figure 6.3: Latency on cluster with Infiniband
Generally, *put* latency is less than *get* latency because there is no hardware supported RDMA read on most network including Infiniband. Others like SCI have lower performance for RDMA read.

ARMCI *get* performance is closer to its *put* because the *get* request is sent to the helper thread (refer section 3.2) on the remote process. The helper thread writes the required data to the requesting image. Thus the *get* is converted into a *put* [29]. MPI *put* and *get* has more latency as these operations are not truly one-sided and uses a rendezvous protocol which requires a hand-shake communication before the data transfer takes place. UHCAF has some overhead over bare ARMCI and GASNet because it allocates a temporary buffer before the communication.

The latency measured on SMP is shown in Figure 6.4.
MPI latency on SMP is not shown in the figure as it is very high compared to others. The latency is 20.45 and 16.43 microseconds for put and get respectively. This is because MPI is not optimized for shared memory systems.

6.1.2 Contiguous Put & Get Bandwidth

In this experiment, the time taken to write on a contiguous chunk of memory on a remote image is measured. The average of 100 iterations is recorded. Figure 6.5 shows the results obtained on a cluster with 2 nodes, each running 1 process.

![Figure 6.5: Bandwidth on cluster with Infiniband](image)

In the above graph it can be seen that UHCAF, UPC, GASNet, and ARMCi achieves very similar bandwidth. These PGAS languages attain better bandwidth than MPI for smaller messages because latency of MPI is high. GA (Global Arrays) preforms better when the message size is less than 128KB, but it does worse for larger

64
message size. This is unexpected because GA uses ARMCI to implement its runtime communication layer. However, as the number of iterations are reduced from 100 to 10, the performance gain reduces. This shows that GA probably optimizes the iteration loop during compilation. Performance of Intel CAF is very poor. Results of Intel CAF are not included in later figures.

Figure 6.6 shows the ratio of the bandwidth of write (put) operation over the bandwidth of read (get) operation. The reference line of 1 represents equal bandwidth for read and write. More than 1 means faster writes while less than 1 means faster reads. Except for MPI, bandwidth of write is generally better than read. For message sizes larger than 64KB, the bandwidth of read and write is almost equal.

![Figure 6.6: Ratio of write over read on Infiniband cluster](image)

Figure 6.7 compares the bandwidth of put on an SMP. GASNet performs better on SMP than ARMCI. Note that GASNET 'everything' configuration is very slow.
on SMP. Significant difference is not found when using the GASNet SMP-conduit as compared to the IBV-conduit.

6.1.3 Bidirectional Bandwidth

In this experiment, 2 processes writes to each other simultaneously. The average bandwidth of 100 iterations is recorded. Figure 6.8 shows the ratio of bidirectional bandwidth over unidirectional bandwidth. The reference point in the figure is fixed at 2 to compare the bandwidth lag from the ideal case. For big message size, the bandwidth gets almost doubled except for Global Arrays.

![Figure 6.7: Bandwidth of put on SMP](image)
6.1.4 Non-contiguous Put

In this experiment, non-contiguous blocks of 4 bytes with stride of 64 bytes are written from one process to another. Figure 6.9 compares the non-contiguous bandwidth on a cluster with 2 nodes, each running 1 process. Results of UPC is not included as it is very slow. The performance varies proportionally with the size of stride. The GASNET_VIS_AMPIPE flag is set, which optimizes non-contiguous writes for GASNet. Without this flag, GASNet performance is poor.
6.1.5 Broadcast

In this experiment, blocks of contiguous data are broadcast to all processes. The average of 100 iterations are recorded. Since there is no broadcast functions in CAF, data is written in a loop to each image. Figure 6.10 compares the broadcast bandwidth when run on a cluster with 8 processes. As expected, CAF broadcast is very slow because it is done in a naive way.
Figure 6.10: Bandwidth of broadcast on cluster (NP8) with Infiniband

Figure 6.11 compares the broadcast bandwidth when run on SMP with 8 processes.

Figure 6.11: Bandwidth of broadcast on SMP (NP8)
6.2 Experiments at TOTAL E&P Research & Technology USA, LLC

The programs used for the experiments were created by Asma Farjallah, France Boillod-Cerneux, and Terrence Liao at TOTAL.

TOTAL performs seismic exploration to find oil both on land and beneath the sea. Sound energy waves are created on the surface using dynamites. Sound waves travel at different velocity in different kind of materials. The timings of the reflected waves are recorded using geophones and hydrophones. These timing information are processed to create seismic profiles using different mathematical models. The programs that are used to evaluate UHCAF performance are part of this process. These programs store the timing data in 3-D matrices and the computation is highly iterative. This makes parallelizing harder, as all the boundary elements (ghost cells) needs to be communicated to other threads/processes after the end of each iteration.

The experiments are performed on a cluster of 330 compute nodes (2640 cores) which have a peak performance of 29.5 TFLOPS. Each node has 2 Intel Nehalem quad-core CPUs, with each core operating at a frequency of 2.8 GHz. The nodes are diskless and have 24GB memory. The interconnect is QDR Infiniband on 8X PCIe 2.0 in a fat tree topology. The upload and download bandwidth of the interconnect is 40Gbps. It uses a shared parallel file system.

The MPI version of the program are executed using Intel MPI version 12. MPI uses 2-sided non-blocking send and receive calls, MPI_ISEND and MPI_Irecv. The
compiler flag ‘-fp-model precise’ is used to ensure that floating point operations conform to IEEE standard. Compiler optimization level -O3 is used for both UHCAF and MPI. The experiments do not evaluate performance on SMP, as only 1 process is run on 1 node. Evaluating SMP performance is out of the scope of these experiments as several other factors (like location of data in memory) have to be considered. Time is measured using the C function gettimeofday. All measurements are an average of 100 iterations. Only the GASNet version of UHCAF is used because ARMCI has some limitations (about 2GB) on the amount of memory that can be registered when using the Infiniband native API. Since the domain size of the programs are mostly greater than 2GB, ARMCI cannot be used.

Before running the experiments, some bandwidth measurements were collected using the microbenchmarks discussed earlier. Figure 6.12 compares the effective bandwidth of UHCAF, OpenMPI and IntelMPI. Note that MPI uses one-sided passive puts in the microbenchmarks. Figure 6.13 compares bandwidth of non-contiguous data. Data blocks of 4 bytes with strides of 64 bytes are used for this experiment. MPI uses one-sided passive puts. Intel MPI does not perform well for strided transfers.
Figure 6.12: Bandwidth of contiguous data communication
GASNet uses environment variable GASNET_VIS_AMPIPE to enable packing of most non-contiguous put/gets. This support is currently experimental, and thus disabled by default. For all the experiments GASNET_VIS_AMPIPE is set. It is noticed that not setting this variable can cause huge performance degradation for noncontiguous transfers. The table 6.1 compares the communication time difference due to this environment variable using the finite difference benchmark for edges (described later) on 8 processes. Figure 6.14 shows that setting this variable provides 16 times performance gain even for small buffer size.
Table 6.1: Communication time (sec) with and without GASNET_VIS_AMPIPE

<table>
<thead>
<tr>
<th>Buffer Size</th>
<th>Domain Size</th>
<th>With</th>
<th>Without</th>
</tr>
</thead>
<tbody>
<tr>
<td>24K</td>
<td>64M</td>
<td>0.0091</td>
<td>0.1140</td>
</tr>
<tr>
<td>32K</td>
<td>128M</td>
<td>0.0115</td>
<td>0.1675</td>
</tr>
<tr>
<td>40K</td>
<td>256M</td>
<td>0.0135</td>
<td>0.2207</td>
</tr>
<tr>
<td>48K</td>
<td>512M</td>
<td>0.0144</td>
<td>0.2215</td>
</tr>
<tr>
<td>64K</td>
<td>1G</td>
<td>0.0174</td>
<td>0.2227</td>
</tr>
<tr>
<td>80K</td>
<td>2G</td>
<td>0.0241</td>
<td>0.3335</td>
</tr>
</tbody>
</table>

Figure 6.14: Performance gain from GASNET_VIS_AMPIPE

6.2.1 Finite Difference Communication Benchmark

In this benchmark, finite difference method is used to solve wave equations. The input data is stored in a 3-D matrix and different programs work on the face, edge, and corner of the cube. In all the tables and figures, the domain size refers to the size of the 3-D matrix, and the buffer size is the actual amount of data that is being
communicated. These experiments are executed on 8 images with 1 image per node.

Table 6.2 shows communication time for ‘edges’. There are 12 edges, but the amount of data is less compared to the face. However, the data is very non-contiguous. Figure 6.15 demonstrates that UHCAF can reduce communication costs up to 3 times with respect to the MPI 2-sided non-blocking send/receive implementation.

Table 6.2: Communication time (seconds) for edges (12) with 8 processes

<table>
<thead>
<tr>
<th>Buffer Size</th>
<th>Domain Size</th>
<th>Intel MPI</th>
<th>UHCAF</th>
</tr>
</thead>
<tbody>
<tr>
<td>24K</td>
<td>64M</td>
<td>0.0212</td>
<td>0.0091</td>
</tr>
<tr>
<td>32K</td>
<td>128M</td>
<td>0.0278</td>
<td>0.0115</td>
</tr>
<tr>
<td>40K</td>
<td>256M</td>
<td>0.0316</td>
<td>0.0135</td>
</tr>
<tr>
<td>48K</td>
<td>512M</td>
<td>0.0331</td>
<td>0.0144</td>
</tr>
<tr>
<td>64K</td>
<td>1G</td>
<td>0.0535</td>
<td>0.0174</td>
</tr>
<tr>
<td>80K</td>
<td>2G</td>
<td>0.0760</td>
<td>0.0241</td>
</tr>
</tbody>
</table>
Table 6.3 shows communication time for ‘faces’. There are only 6 faces, but the amount of data is significantly higher than ‘edges’. Figure 6.16 demonstrates the performance gain (n times) of UHCAF over Intel MPI.

Table 6.3: Communication time (seconds) for faces (6) with 8 processes

<table>
<thead>
<tr>
<th>Buffer Size</th>
<th>Domain Size</th>
<th>Intel MPI</th>
<th>UHCAF</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.75M</td>
<td>64M</td>
<td>0.2322</td>
<td>0.1210</td>
</tr>
<tr>
<td>1.25M</td>
<td>128M</td>
<td>0.5288</td>
<td>0.2305</td>
</tr>
<tr>
<td>2M</td>
<td>256M</td>
<td>1.1650</td>
<td>0.4566</td>
</tr>
<tr>
<td>3M</td>
<td>512M</td>
<td>1.2634</td>
<td>0.6596</td>
</tr>
<tr>
<td>5M</td>
<td>1G</td>
<td>1.5935</td>
<td>1.1459</td>
</tr>
<tr>
<td>8M</td>
<td>2G</td>
<td>3.3760</td>
<td>2.0643</td>
</tr>
</tbody>
</table>
Table 6.4: Communication time (seconds) for corners (6) with 8 processes

<table>
<thead>
<tr>
<th>Buffer Size</th>
<th>Intel MPI</th>
<th>UHCAF</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.25K</td>
<td>0.0017</td>
<td>0.0023</td>
</tr>
</tbody>
</table>

6.2.2 Isotropic Forward Wave Equation

This is the simplest case of the wave equation as the velocity of sound does not change in the isotropic medium. In this program, two 3-D matrices are used to calculate...
2-way wave equation using scalar acoustic wave equation.

Table 6.5 shows the time taken (seconds) to execute the isotropic forward wave equation solver on 8 nodes. The computation timings are only for the kernel code, which does not include file I/O. The Intel MPI version is 12 and UHCAF uses GASNET. Both use optimization level O3. The buffer size is the total size of the data that is being communicated by each process. The domain size is the total size of the 3-D matrix. The ‘compute’ column contains the average of the computation time among all processes. The ‘comm’ column contains the average of the communication time among all processes. The ‘Total’ column contains the total time, which is same on all processes due to synchronization.

Table 6.5: Isotropic forward wave equation solver timings (sec) with 8 processes

<table>
<thead>
<tr>
<th>Buffer Size</th>
<th>Domain Size</th>
<th>Intel MPI</th>
<th>UHCAF</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Compute</td>
<td>Comm</td>
<td>Total</td>
</tr>
<tr>
<td>0.75M</td>
<td>64M</td>
<td>1.55</td>
<td>0.24</td>
</tr>
<tr>
<td>1.25M</td>
<td>128M</td>
<td>3.09</td>
<td>0.52</td>
</tr>
<tr>
<td>2M</td>
<td>256M</td>
<td>6.18</td>
<td>1.12</td>
</tr>
<tr>
<td>3M</td>
<td>512M</td>
<td>11.21</td>
<td>1.40</td>
</tr>
<tr>
<td>5M</td>
<td>1G</td>
<td>22.14</td>
<td>1.75</td>
</tr>
<tr>
<td>8M</td>
<td>2G</td>
<td>49.66</td>
<td>3.33</td>
</tr>
<tr>
<td>12M</td>
<td>4G</td>
<td>96.78</td>
<td>6.25</td>
</tr>
<tr>
<td>20M</td>
<td>8G</td>
<td>184.46</td>
<td>7.37</td>
</tr>
<tr>
<td>32M</td>
<td>16G</td>
<td>410.53</td>
<td>33.64</td>
</tr>
</tbody>
</table>

Even though the total execution time of the UHCAF version is more than the Intel MPI version (Figure 6.17), the communication time is significantly less as shown in the Figure 6.18.
Figure 6.17: Kernel execution time (includes communication)

Figure 6.18: Communication time (seconds) with 8 processes
Figure 6.19 shows the increase in performance more clearly. For 32M buffer UHCAF is three times faster than Intel MPI. This shows that UHCAF makes better utilization of one-sided RDMA capabilities provided by the Infiniband interconnect. Another interesting point is that MPI uses non-blocking communication while CAF uses blocking. As discussed before, Infiniband buffers large messages and into smaller blocks. This reduces the performance benefit of using non-blocking for larger message sizes.

![Figure 6.19: Communication time ratio (MPI/UHCAF) with 8 processes](image)

Table 6.6 shows the timing (seconds) when the program is executed using a fixed problem size of 16GB. This is used to measure the speed-up. Four processes are used for the first run and increased to 128 processes. The buffer size is not uniformly distributed among all processes for some cases. The computation and communication
times are the average of all processes. Figure 6.20 shows the speed-up when the number of processes doubles. The speed-up is more than 2 in some cases.

Table 6.6: Timing (seconds) for domain size 16GB

<table>
<thead>
<tr>
<th>Buffer Size</th>
<th>Process</th>
<th>Intel MPI Compute</th>
<th>Intel MPI Comm</th>
<th>Total</th>
<th>UHCAF Compute</th>
<th>UHCAF Comm</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>32M</td>
<td>4</td>
<td>892.35</td>
<td>21.44</td>
<td>923.47</td>
<td>1356.79</td>
<td>2.12</td>
<td>1358.92</td>
</tr>
<tr>
<td>32M</td>
<td>8</td>
<td>410.53</td>
<td>33.64</td>
<td>452.40</td>
<td>492.63</td>
<td>10.41</td>
<td>544.66</td>
</tr>
<tr>
<td>20M-28M</td>
<td>16</td>
<td>194.19</td>
<td>11-12</td>
<td>208.05</td>
<td>261.23</td>
<td>5.4-7.3</td>
<td>274.91</td>
</tr>
<tr>
<td>12M-20M</td>
<td>32</td>
<td>100.07</td>
<td>6-10</td>
<td>110.66</td>
<td>141.26</td>
<td>3-5.7</td>
<td>148.08</td>
</tr>
<tr>
<td>8-16M</td>
<td>64</td>
<td>45.68</td>
<td>6-7</td>
<td>55.72</td>
<td>61.45</td>
<td>3-7.4</td>
<td>70.93</td>
</tr>
<tr>
<td>5-10M</td>
<td>128</td>
<td>21-23</td>
<td>3-3.6</td>
<td>27.34</td>
<td>27-30</td>
<td>1.5-3.6</td>
<td>34.79</td>
</tr>
</tbody>
</table>

Figure 6.20: Speedup over 4 processes using fixed domain size of 16GB
6.2.3 Tilted Transverse Isotropic (TTI) Wave Equation

This program is much more complex than the Isotropic program as it models anisotropic media, which has a lot more parameters to consider. This program requires six 3-D matrices to store the timing data, which is subdivided to be processed by each image. After each iteration the ghost cells is exchanged. Due to huge memory requirement, the program cannot be executed with less than 16 images. The OpenMPI version uses traditional assumed shape array declarations instead of dynamic allocation (to prevent performance impact). The program is executed twice with Intel MPI, with and without the xhost flag. The xhost flag tells the compiler to optimize for the specific hardware. Table 6.7 and Figure 6.21 compares the communication time between UHCAF and MPI using 16GB domain size. The matrix dimensions are 1024x2048x2048 with 4 ghost points. The buffer size in the table is the sum of all the communication buffer of all processes. Table 6.8 compares the memory requirement of TTI with the isometric forward wave equation.

<table>
<thead>
<tr>
<th>Buffer (GB)</th>
<th>#Processes</th>
<th>UHCAF</th>
<th>OpenMPI</th>
<th>Intel MPI</th>
<th>Intel MPI xhost</th>
</tr>
</thead>
<tbody>
<tr>
<td>75.54</td>
<td>16</td>
<td>23.17</td>
<td>34.67</td>
<td>34.13</td>
<td>27.28</td>
</tr>
<tr>
<td>101.03</td>
<td>32</td>
<td>13.22</td>
<td>17.45</td>
<td>19.48</td>
<td>19.23</td>
</tr>
<tr>
<td>152.20</td>
<td>64</td>
<td>17.87</td>
<td>20.60</td>
<td>18.84</td>
<td>14.14</td>
</tr>
<tr>
<td>203.96</td>
<td>128</td>
<td>8.84</td>
<td>9.18</td>
<td>9.41</td>
<td>9.12</td>
</tr>
</tbody>
</table>
Table 6.8: Memory usage comparison between TTI and isometric

<table>
<thead>
<tr>
<th>np</th>
<th>nx</th>
<th>ny</th>
<th>nz</th>
<th>xdim</th>
<th>ydim</th>
<th>zdim</th>
<th>TTI</th>
<th>ISO</th>
<th>Size(MB)</th>
</tr>
</thead>
<tbody>
<tr>
<td>16</td>
<td>2</td>
<td>2</td>
<td>4</td>
<td>512</td>
<td>1024</td>
<td>512</td>
<td>10.16</td>
<td>2.08</td>
<td>96.7</td>
</tr>
<tr>
<td>32</td>
<td>2</td>
<td>4</td>
<td>4</td>
<td>512</td>
<td>512</td>
<td>512</td>
<td>5.1</td>
<td>1.05</td>
<td>64.7</td>
</tr>
<tr>
<td>64</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>256</td>
<td>512</td>
<td>512</td>
<td>2.56</td>
<td>0.53</td>
<td>48.7</td>
</tr>
<tr>
<td>128</td>
<td>4</td>
<td>4</td>
<td>8</td>
<td>256</td>
<td>512</td>
<td>256</td>
<td>1.29</td>
<td>0.27</td>
<td>32.6</td>
</tr>
</tbody>
</table>

Table 6.9 and Figure 6.22 compares the total execution time of the TTI program. Note that it does not include file IO. The buffer size in the table is the sum of all the communication buffer of all processes. Figure 6.23 shows the speedup with respect to 16 processes.
Table 6.9: Total execution time (seconds) for 16 GB domain size

<table>
<thead>
<tr>
<th>Buffer (GB)</th>
<th>#Processes</th>
<th>UHCAF</th>
<th>OpenMPI</th>
<th>Intel MPI</th>
<th>Intel MPI xhost</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.08</td>
<td>16</td>
<td>2084.81</td>
<td>3149.93</td>
<td>2248.07</td>
<td>2128.65</td>
</tr>
<tr>
<td>1.15</td>
<td>32</td>
<td>1094.02</td>
<td>1559.49</td>
<td>1247.73</td>
<td>1172.55</td>
</tr>
<tr>
<td>0.61</td>
<td>64</td>
<td>519.54</td>
<td>866.08</td>
<td>622.67</td>
<td>528.76</td>
</tr>
<tr>
<td>0.26</td>
<td>128</td>
<td>276.01</td>
<td>449.17</td>
<td>328.40</td>
<td>271.15</td>
</tr>
</tbody>
</table>

Figure 6.22: Total time (seconds) for TTI
Figure 6.23: Speedup over 16 processes
Chapter 7

Conclusion and Future Work

The need for high performance is ever increasing. Exascale systems are around the corner. There is a lack of robust and efficient programming models which can exploit the full potential of both clusters and many-core systems. PGAS languages can be effective in bridging this gap.

There are many scientific applications written in Fortran which can benefit greatly by parallel execution. However, it is complicated to inject external library calls inside these applications to make it run in parallel. Converting these applications to CAF can be much more easier.

Experiments show that UHCAF runtime achieves good performance compared to MPI and other PGAS implementations. There is however, a lot of scope for optimizations. The runtime library can be extended to support extra features like collective broadcast and reduction operations. Support for teams of images might
also be useful on large clusters. The runtime can automatically convert write operations to non-blocking, but it cannot do so for read operations. Adding non-blocking reads to CAF will be useful. Using pthreads instead of GASNet/ARMCI for SMP should be explored to see if there is any performance improvement. Implementing the runtime over bare network API instead of using GASNet/ARMCI might be useful on some network interconnects. Using one-sided MPI should also be explored, especially when MPI 3.0 is available.

The data structures used in the runtime grows linearly with the number of processes executing the application. For example, in section 4.3 the starting memory address of the pinned memory segment of all images are communicated to every other image. This can cause huge performance bottleneck when there are thousands of images. In order make the UHCAF runtime implementation more scalable, it must use tree-based algorithm instead of linearly storing metadata. Ways to implement fault tolerance in CAF must also be explored to make it work on large clusters.
Appendix A

Runtime API

This appendix describes the interface provided by the runtime library. The compiler inserts these function invocations in the program executable.

A.1 Program Initialization

```c
void caf_init();
```

Initializes the SPMD execution and sets up the registered/pinned memory segment for remote memory access.
A.2 Synchronization

void sync_all()
Implements CAF ‘sync all’. It is a global barrier.

void sync_memory()
Implements CAF ‘sync memory’. It is a no-op because most hardwares provide
cache-coherency.

void sync_images_( int *imageList, int imageCount)
Implements CAF ‘sync images’ which is a point-to-point or one-to-many barrier.
The array of images (imageList) and their count (imageCount) are passed in the
parameter.

void sync_images_all()
This additional function is provided to implement ‘sync images (*)’, which is not
same as ‘sync all’ because it might be a one-to-all barrier.

A.3 Image Inquiry

int image_index_(DopeVectorType *diminfo, DopeVectorType *sub)
Implements the image_index(coarray,sub) intrinsic of CAF. It returns the linearized
image id from the co-dimension information. Dopevector diminfo is the coarray
dopevector, while sub is the dopevector containing the cosubscripts passed to the image_index in the CAF program.

```c
int this_image3_(DopeVectorType *diminfo, int* sub)
```

Implements this_image(coarray, dim) where diminfo is the dopevector representing the coarray and sub is a pointer to an integer denoting the corank. It returns an integer value denoting the index of the co-dimension of the invoking image.

```c
void this_image2_ (DopeVectorType *ret, DopeVectorType *diminfo)
```

Implements this_image(coarray) where diminfo is the dopevector representing the coarray and ret is an empty dopevector where the array of coranks will be returned.

```c
int lcobound2_(DopeVectorType *diminfo, int *sub)
```

Implements lcobound(coarray, dim) where diminfo is the dopevector representing the coarray and sub is a pointer to the corank passed in dim. It returns the lower cobound of the coarray for that rank.

```c
void lcobound_ (DopeVectorType *ret, DopeVectorType *diminfo)
```

Implements lcobound(coarray) where diminfo is the dopevector representing the coarray and ret is an empty dopevector where the lcobound values of all the ranks will be returned.
int ucobound2_(DopeVectorType *diminfo, int *sub)
Implements ucobound(coarray,dim) where diminfo is the dopevector representing
the coarray and sub is a pointer to the corank passed in dim. It returns the upper
cobound of the coarray for that rank.

void ucobound_(DopeVectorType *ret, DopeVectorType *diminfo)
Implements ucobound(coarray) where diminfo is the dopevector representing the
coarray and ret is an empty dopevector where the ucobound values of all the ranks
will be returned.

A.4 Local Buffer

These functions are used to create local buffers for remote read/write operation.

void acquire_lcb_(unsigned long buf_size, void **ptr)
Allocates buf_size to the ptr.

void release_lcb_(void **ptr)
Deallocates ptr.
A.5 Remote Read

void coarray_read_(void * src, void * dest, unsigned long xfer_size, unsigned long img)
Implements remote read of contiguous memory. Src is the remote address, dest is the local address, xfer_size is the size of data in bytes, and img is the image index of the remote image.

void coarray_read_src_str_(void * src, void *dest, unsigned int ndim, unsigned long *src_strides, unsigned long *src_extents, unsigned long img)
Implements remote read for strided memory. The local memory is contiguous. Src is the remote memory address, dest is the local address, ndim, src_strides, and src_extents provide the stride information; ndim is the number of memory chunks; src_stride is the array specifying strides for each chunk; src_extent is the array specifying the size of each chunk.

void coarray_read_full_str_(void * src, void *dest, unsigned int src_ndim, unsigned long *src_strides, unsigned long *src_extents, unsigned int dest_ndim, unsigned long *dest_strides, unsigned long *dest_extents, unsigned long img)
Implements remote read for strided memory, where the local memory is also strided. Src is the remote memory address, dest is the local address, src_ndim and dest_ndim is the number of memory chunks for source and destination; src_stride and dest_stride is the array specifying strides for each chunk; src_extent and dest_extent is the array
specifying the size of each chunk.

A.6 Remote Write

void coarray_write_(void * dest, void * src, unsigned long xfer_size, unsigned long img)
Implements remote write of contiguous memory. Dest is the remote address, src is the local address, xfer_size is the size in bytes that are to be written.

void coarray_write_dest_str_(void * dest, void *src, unsigned int ndim, unsigned long *dest_strides, unsigned long *dest_extents, unsigned long img)
Implements remote write for strided memory. The local memory is contiguous. Dest is the remote memory address, src is the local memory address; ndim is the number of memory chunks; dest_strides is the array specifying the stride for each chunk; dest_extents is the array specifying the size of each chunk.

void coarray_write_full_str_(void * dest, void *src, unsigned int dest_ndim, unsigned long *dest_strides, unsigned long *dest_extents, unsigned int src_ndim, unsigned long *src_strides, unsigned long *src_extents, unsigned long img)
Implements remote write when both local and remote memory is strided. Dest is the remote address, src is the local address; dest_ndim and src_ndim is the number
of memory chunks that are to be transferred; dest_strides and src_strides are the arrays specifying the stride for each chunk; dest_extents and src_extents are the arrays specifying the size of each chunk.

A.7 Exit

void caf_exit_(int status)
Implements exit with an error code. Ensures freeing of pinned-down memory and other data structures.

void caf_finalize_()
Implements normal exit. Ensures freeing of pinned-down memory segment and other data structures.
Bibliography


