COMPILER AND RUNTIME APPROACH FOR
SUPPORTING EFFICIENT EXECUTION OF COARRAY
FORTRAN PROGRAMS

A Dissertation
Presented to
the Faculty of the Department of Computer Science
University of Houston

In Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy

By
Deepak Eachempati
May 2015
COMPILER AND RUNTIME APPROACH FOR SUPPORTING EFFICIENT EXECUTION OF COARRAY FORTRAN PROGRAMS

Deepak Eachempati

APPROVED:

Barbara Chapman, Chairman
Dept. of Computer Science

Edgar Gabriel
Dept. of Computer Science

Lei Huang
Dept. of Computer Science
Prairie View A&M University

Jaspal Subhlok
Dept. of Computer Science

Arthur Weglein
Dept. of Earth and Atmospheric Sciences

Dean, College of Natural Sciences and Mathematics
Acknowledgements

I would like to express here my sincere appreciation for all those who made this PhD work possible. My committee chair and advisor, Dr. Barbara Chapman, supported me throughout this lengthy process, and it was an honor for me to draw from her experience and knowledge during this time working as her student. I would like to thank also my other committee members, Dr. Edgar Gabriel, Dr. Lei Huang, Dr. Jaspal Subhlok, and Dr. Arthur Weglein, who all took the time to review my work and offer their valued feedback.

During this process, I have been extremely fortunate to have this work funded by Total. A huge thanks to Henri Calandra from Total who has been extremely supportive of this effort. Several HPC experts from Total, including Terrence Liao, Saber Feki, Maxime Huges and Mikhail Sekachev, have helped in overseeing and assessing the progress of this project. I would like to thank also the summer interns employed at Total, Asma Farjalla, France Boillod-Cerneux, and Alan Richardson, who have worked with and evaluated our implementation using Total benchmarks and application code. I had also the opportunity to work in the Computer Science and Mathematics Division at Oak Ridge National Laboratory (ORNL) under the guidance of Oscar Hernandez for two summers. For ORNL’s generous support of my research work, they have my gratitude. I would like to also extend my thanks to the participants in the Fortran language standards working group, as they were responsible for the design of several language features explored in this work.

I have had the privilege to work with and receive guidance from a number of senior members in my research group. Dr. Huang was a mentor to me in my early years.
before joining the faculty at Prairie View A&M University. Sunita Chandrasekeran, Abid Malik, and Dr. Yonghong Yan all provided much appreciated suggestions for the content and scope of my dissertation. Tony Curtis has been an invaluable resource for technical information relating to PGAS library implementations. Dounia Khaldi has helped me in the past year on multiple fronts, in particular on the design and evaluation of our teams implementation and spending considerable time in reviewing and offering suggestions on many drafts of this dissertation. I would like to extend my heartfelt thanks to all of them.

As a student in the HPCTools research group, I had the opportunity to work with and receive valuable feedback from several fellow students. During the beginning stages of this project, I worked closely with Hyoung Joon Jun, and together we developed an early prototype of the compiler and runtime system. Debyjoti Majumder helped me greatly in the early development of key runtime components, in particular the memory management and non-blocking support. Siddhartha Jana worked with me on the design of parallel I/O extensions for Coarray Fortran. In the past year, I have had the pleasure to work with Shiyao Ge and Naveen Ravichandrasekaran, who have been instrumental in the support of teams in our runtime and also for retargetting coarray programs to execute over the OpenSHMEM library interface. To all these students, and many more with whom I have engaged in valuable discussions over the years, thank you.

Finally, to my parents, who’s unwavering support and encouragement helped me beyond expression over the years, I am forever grateful.
COMPILER AND RUNTIME APPROACH FOR SUPPORTING EFFICIENT EXECUTION OF COARRAY FORTRAN PROGRAMS

An Abstract of a Dissertation
Presented to
the Faculty of the Department of Computer Science
University of Houston

In Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy

By
Deepak Eachempati
May 2015
Abstract

Fortran remains a very widely used programming language for technical computing. Fortran coarrays are new elements in the Fortran standard which aim to support the development of parallel programs without requiring the use of external constructs such as libraries (e.g. MPI) or directives (e.g. OpenMP). Coarrays provide a Partitioned Global Address Space (PGAS) approach to parallel programming in which the programmer declares arrays to be partitioned across different *images* that collaboratively perform the computations. However, to date coarrays are not widely used in commercial code since a robust implementation has only been available on Cray platforms. As a result, there has been relatively little experience with coarrays, reporting of performance data, and a clear lack of portability.

The work described in this dissertation aims to make a fundamental contribution to the state of the art of parallel programming by providing a robust, open-source implementation of the coarray features in the Fortran standard. It describes an efficient, near-commercial strength implementation technology for coarrays which we developed. An evaluation of this implementation using developed micro-benchmarks is presented, where we show in particular the benefits of our approach for supporting strided communication, synchronization, and collectives compared to other coarray implementations. When running developed benchmarks and codes from real-world applications, we obtained performance with our coarray implementation on par or exceeding that obtained using other implementations. We also present support for additional features expected to be added to the Fortran standard, demonstrating its utility for writing and improving the performance of a range of codes. Finally, we describe an extension for parallel I/O in coarray programs which we designed.
# Contents

1 Introduction 1

1.1 Motivation 2

1.2 Contributions 2

1.3 Dissertation Organization 3

2 Background 5

2.1 Parallel Architectures 6

2.1.1 Parallelism in Processor Architectures 6

2.1.2 Shared Memory Architectures 10

2.1.3 Distributed Memory Architectures 11

2.2 Parallel Programming Models 14

2.2.1 Traditional Parallel Programming Models 14

2.2.2 Partitioned Global Address Space Model 17

2.3 Compiling for Parallel Programming Models 19

2.4 Summary 22

3 Survey of PGAS Models and Implementations 23

3.1 PGAS Libraries 23

3.1.1 Global Arrays 24

3.1.2 OpenSHMEM 25
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>6.1</td>
<td>Overview of our Coarray Fortran Implementation</td>
<td>59</td>
</tr>
<tr>
<td>6.1.1</td>
<td>Front-end</td>
<td>60</td>
</tr>
<tr>
<td>6.1.2</td>
<td>Back-end</td>
<td>60</td>
</tr>
<tr>
<td>6.1.3</td>
<td>Runtime System</td>
<td>64</td>
</tr>
<tr>
<td>6.1.4</td>
<td>Incorporating Teams Support</td>
<td>65</td>
</tr>
<tr>
<td>6.2</td>
<td>Remote Access Memory Management</td>
<td>66</td>
</tr>
<tr>
<td>6.3</td>
<td>Intermediate Communication Buffers</td>
<td>69</td>
</tr>
<tr>
<td>6.3.1</td>
<td>Local Destination Buffers</td>
<td>70</td>
</tr>
<tr>
<td>6.3.2</td>
<td>Local Source Buffers</td>
<td>71</td>
</tr>
<tr>
<td>6.3.3</td>
<td>Cases for which an LCB is not used</td>
<td>71</td>
</tr>
<tr>
<td>6.4</td>
<td>Achieving Communication-Computation Overlap</td>
<td>73</td>
</tr>
<tr>
<td>6.4.1</td>
<td>RMA Completion Semantics</td>
<td>75</td>
</tr>
<tr>
<td>6.4.2</td>
<td>Memory consistency modes</td>
<td>76</td>
</tr>
<tr>
<td>6.4.3</td>
<td>Compiler Directives for CAF</td>
<td>78</td>
</tr>
<tr>
<td>6.4.4</td>
<td>Remote Memory Access Interface</td>
<td>79</td>
</tr>
<tr>
<td>6.5</td>
<td>Supporting Non-contiguous Remote Data Access</td>
<td>81</td>
</tr>
<tr>
<td>6.5.1</td>
<td>Communication Generation</td>
<td>81</td>
</tr>
<tr>
<td>6.5.2</td>
<td>Generating and Executing Strided Array Accesses</td>
<td>84</td>
</tr>
<tr>
<td>6.6</td>
<td>Synchronization</td>
<td>88</td>
</tr>
<tr>
<td>6.6.1</td>
<td>Compiler Support for Image Control Statements</td>
<td>88</td>
</tr>
<tr>
<td>6.6.2</td>
<td><strong>sync all</strong> and <strong>sync team</strong></td>
<td>90</td>
</tr>
<tr>
<td>6.6.3</td>
<td><strong>sync images</strong></td>
<td>93</td>
</tr>
<tr>
<td>6.6.4</td>
<td>Atomics Support</td>
<td>99</td>
</tr>
<tr>
<td>6.6.5</td>
<td>Event Synchronization</td>
<td>99</td>
</tr>
<tr>
<td>6.6.6</td>
<td>Coarray Locks</td>
<td>100</td>
</tr>
<tr>
<td>6.7</td>
<td>Optimized Collectives</td>
<td>103</td>
</tr>
<tr>
<td>6.7.1</td>
<td>Basic Scheme</td>
<td>104</td>
</tr>
</tbody>
</table>
List of Figures

2.1 Shared Memory System ........................................... 11
2.2 Distributed Memory System .................................... 12
2.3 PGAS Programming Model ...................................... 18
5.1 The OpenUH Compiler Infrastructure .......................... 51
6.1 OpenUH CAF Implementation Stack ........................... 59
6.2 Layout of data in remote access memory segments .......... 67
6.3 Using directives to control communication-computation overlap ... 78
6.4 Array sections handled by our strided access descriptor ....... 85
6.5 Strided point-to-point tests on Titan using 256 images across 16 nodes 87
6.6 Team barrier microbenchmark on Whale, using 8 images per node 93
6.7 sync images scaling performance on Titan using 16 images per node 97
6.8 sync images tests from EPCC microbenchmark suite using 1024 images over 64 nodes on Titan ................................. 98
6.9 Coarray locks test from EPCC microbenchmark suite on Titan . 103
6.10 co_sum (allreduce) performance on Stampede using 64 nodes, 1024 images ......................................................... 110
6.11 co_sum (allreduce) performance on Titan using 64 nodes, 1024 images ................................................................. 111
7.1 High Performance Linpack (HPL) Benchmark on Whale ........ 115
7.2 Distributed hash table benchmark on Titan, using 16 images per node 117
<table>
<thead>
<tr>
<th>Section</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>7.3</td>
<td>CAF (Fortran 2008) versions of the NAS Parallel Benchmarks on Titan, using 16 images per node. All benchmarks were run using class size B.</td>
</tr>
<tr>
<td>7.4</td>
<td>CG Benchmark on Whale, using 8 images per node</td>
</tr>
<tr>
<td>7.5</td>
<td>A section of the 2004 BP model image generated by RTM using CAF.</td>
</tr>
<tr>
<td>7.6</td>
<td>Neighbor Communication Time for MPI, CAF GET, and CAF PUT (strided and buffered)</td>
</tr>
<tr>
<td>7.7</td>
<td>Runtime (communication and computation) of the forward stage of CAF and MPI versions of the code, for (a) isotropic and (b) TTI, as the number of processors was increased with a fixed total problem size.</td>
</tr>
<tr>
<td>8.1</td>
<td>Example of a file access pattern by multiple images</td>
</tr>
<tr>
<td>8.2</td>
<td>Example using MPI IO</td>
</tr>
<tr>
<td>8.3</td>
<td>Example using the proposed Fortan parallel I/O</td>
</tr>
<tr>
<td>8.4</td>
<td>The flexibility of the same dataset being represented using different dimension layout allows the adaptability of algorithms semantics to file contents</td>
</tr>
<tr>
<td>8.5</td>
<td>Use of triplet notations to access different data patterns in a file</td>
</tr>
<tr>
<td>8.6</td>
<td>Multiple steps used to access a pattern from a file represented as a one-dimensional array</td>
</tr>
<tr>
<td>8.7</td>
<td>The triplet notation can easily be used to represent the data access pattern if the file is represented logically as a one dimensional array instead of higher dimensions.</td>
</tr>
</tbody>
</table>
## List of Tables

<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>8.1</td>
<td>I/O control specifiers for the OPEN statement</td>
<td>138</td>
</tr>
<tr>
<td>8.2</td>
<td>I/O control specifiers for the READ / WRITE statements</td>
<td>142</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

The driving goal of the High Performance Computing (HPC) research community, particularly in the last few decades, has been to identify more efficient means for exploiting parallelism to solve big problems. To meet these demands, both hardware and the programming models used to exploit this hardware have been in a continuous state of evolution. Architectural trends in processor, memory, storage, and interconnect technology have necessitated corresponding advances in the software, with the need to manage increasing degrees of concurrency on these systems being a particular challenge. Thus, a crucial aspect of the computing environment is the system software – including compilers, libraries, and analysis tools – that provides to the user efficient implementations of parallel programming models.

One of the programming models that has emerged in recent years is the Partitioned Global Address Space (PGAS) model, which provides a means for the programmer to develop scalable parallel algorithms that take into account data-processor
affinity. Several different library interfaces have been developed that supports this model and are widely used. Many new programming languages or language extensions have also been proposed that supports the PGAS model. Language-based implementations have the potential to simplify the burden of writing applications that effectively utilize these feature for the non-expert programmer. However, their adoption has been much slower compared to the library-based implementations.

1.1 Motivation

Fortran is widely used in the HPC arena, and notably Fortran 2008 incorporated a set of new language features which support the PGAS programming model, informally referred to as Fortran coarrays or Coarray Fortran (CAF). However, even though these features have been standardized for several years, to date efficient support for them are largely missing from current Fortran compilers. This points to a need to demonstrate how these features may be effectively implemented in an open-source compiler, thus providing a path for other implementations of these extensions or other, similar, PGAS languages.

1.2 Contributions

To this end, we have created the first fully functioning and portable implementation of these coarray extensions within the OpenUH compiler and used our implementation to explore its performance potential. As part of this work, we explored the
various factors that impact performance of applications utilizing the PGAS programming model. To contend with these issues, we developed a variety of implementation strategies that may be adopted by other compiler and runtime infrastructures. We demonstrate the efficacy of our approach through a comparison against MPI implementations, the state-of-the-art Cray Fortran compiler, and other coarray implementations.

1.3 Dissertation Organization

This dissertation is organized as follows.

Chapter 2 provides background information for this dissertation. We review key concepts in state-of-the-art parallel architecture and programming models. We then present a general overview of compilation and runtime techniques for parallel programming models.

In Chapter 3, we present a survey of PGAS programming models and their implementation. In Chapter 4, we take a closer look at Coarray Fortran, describing the features of this language within the Fortran standard and previewing anticipated parallel processing additions in the next standard revision. Chapter 5 reviews existing compiler and runtime infrastructure that was used for developing the compilation and runtime strategies described in this dissertation.

Chapter 6 describes in detail the design of the compiler and its runtime system for supporting efficient execution of coarray programs. This includes the overall design
of the implementation, extensions made to the compiler to represent, optimize, and generate code for PGAS language features, and the runtime system support that we developed. Several technical challenges for implementing these extension are identified, and we present techniques that were developed as part of this work to meet them.

Chapter 7 provides an evaluation of our implementation based on experimentation using several benchmarks and applications. In Chapter 8, we present a language extension for parallel I/O in CAF programs that we developed. Finally, Chapter 9 concludes this dissertation and includes a description of future work.
Chapter 2

Background

In this chapter, we present an overview of parallel computing systems. We describe parallel architectures, including advances in multi-processor systems, memory subsystems, and interconnect technology. We include as part of this discussion various parallel programming models that are commonly used to implement parallel algorithms on such architectures, and we explain how the Partitioned Global Address Space model is a good match for observable architectural trends. Supporting such programming models requires additional system software support, such as compilers, runtime systems, and assorted tools for performance tuning. We end this chapter by reviewing compilation techniques for facilitating parallel computation.
2.1 Parallel Architectures

2.1.1 Parallelism in Processor Architectures

In discussing the trends towards increasing parallelism in computer architectures, a good starting point is to consider the evolution of processor technology. Historically, processors have supported parallelism at several levels. This can be divided into support for fine-grained, instruction-level parallelism, and support for coarse-grained, task-level parallelism.

Instruction-level parallelism was supported in the following ways:

- **Bit-level Parallelism**: At the lowest level, there is bit-level parallelism. This refers to the fact that processors consist internally of data paths and registers which operate on word-sized bits (typically 32- or 64-bit). Supporting bit-level parallelism required the ability to perform arithmetic computations (e.g. addition, multiplication) on multiple bits in parallel.

- **Pipeline Parallelism**: Pipelining processors became common in order to support handling of multiple instructions simultaneously. The instruction execution path is divided into multiple pipeline stages, each of which may be operating on a different instruction. As uniprocessors became increasingly complex and time per cycle reduced, a common trend was the increase in pipeline stages.

- **Superscalar Parallelism**: Superscalar architectures were eventually introduced, adding support for multiple-issue of instructions, replicated functional
units (e.g. arithmetic logic units, floating-point units, branch units, etc.), and they allowed more than one instruction to be completed per cycle. Dynamic reordering of instructions based on data dependences of instructions allowed increased throughput of instructions. In practice, data dependences among instructions in a sequential stream imposed a constraint on the level of instruction-level parallelism achievable with these techniques.

- **VLIW Instruction Parallelism**: Another direction was support for Very Long Instruction Word (VLIW) instruction sets. On VLIW processors, a serial stream of long instruction words, or tuples, were issued. VLIW architectures relied on static scheduling of instructions by the compiler, rather than dynamic scheduling by the processor. This greatly simplified the microarchitecture design of the processors, particularly for instruction scheduling and issue, but at the cost of putting more burden on the compiler to identify instruction-level parallelism and schedule instructions accordingly.

- **Vector Parallelism**: Vector processing refers to processors supporting instructions which operate on arrays of data (or vectors) in parallel. Such operations were extremely important on array processing applications, used often in scientific applications such as numerical simulation. Historically, CPUs often had coprocessors with specialized support for vector instruction sets, but most modern CPUs include support for vector instruction sets (e.g. VIS, MMX, SSE, AVX).

The degree of parallelism achievable with the above approaches was generally
limited, as it relied on finding independent instructions in a *sequential* instruction stream. Vectorization could be very effective for codes that were amenable to this technique – so-called “data-parallel” parallelism. In modern systems, a high degree of parallelism is achievable on such codes with the use of specialized accelerator processors such as *General Purpose Graphics Processing Units* (GPGPUs).

Support for task-level parallelism within processors, generally through supporting multi-threaded programs, eventually became available:

- **Thread-interleaving Parallelism:** In order to support multiple threads of execution, techniques to allow the processor to efficiently switch between thread contexts were developed. Two interleaving techniques were *cycle-by-cycle interleaving* and *block interleaving*. Cycle-by-cycle interleaving allowed an instruction from a different thread to be fetched into the processor pipeline at each clock cycle. Pipeline hazards were avoided by preventing an instruction from one thread from being fetched until the previous instruction for that thread completed. Therefore, effective utilization of the processor pipeline in this scheme depended on as many threads executing concurrently as there were pipeline stages. In the block interleaving approach, a single thread executes until it reaches a point which triggers a context switch for the processor. This trigger point could be a special type of instruction, or more dynamically an event such as a memory access operation that misses in the cache which would entailed a long latency.
• **SMT Parallelism:** The Simultaneous Multi-Threading (SMT) technique incorporated into superscalar processors the ability to exploit more coarse-grained parallelism in the form of threads. SMT allowed multiple instructions from different threads to be issued in the same cycle and executed simultaneously, utilizing the distinct functional units available in superscalar architectures. Additional hardware resources were added to support multiple, simultaneous threading contexts. This could greatly increase the overall utilization of processor resources, in comparison to the thread-interleaving approach.

• **Multi-core Parallelism:** Multi-core processors, which have become standard in modern processors even for desktop computing, took greater advantage of the ability to pack more transistors on a die by incorporating multiple processor cores on the same chip. The cores, each with their own independent pipelines, allowed a single processor to execute multiple threads or even multiple applications in parallel. Cores typically had their own cache, and could share larger caches with other cores.

To support higher degrees of parallelism in a cost-effective manner, developing uniprocessor technology with ever increasing complexity was not a viable option. For this, parallel computing has evolved to incorporate multi-processing, cluster computing, and even distributed computing. Parallel applications require some level of communication or information transfer between the processors, and supporting this at large scales required evolution in processor interconnect technology.
2.1.2 Shared Memory Architectures

As a typical configuration, we can think of shared memory systems as shown in Figure 2.1. Here, the processors share a single global address memory space, which means that the memory is accessible by all processors. We use the term *Shared Memory Multi-Processor* (SMP) to describe systems organized in this manner. The acronym SMP may also be used to refer to a *Symmetric Multi-Processor* system, in which there is essentially uniform access (i.e. similar latencies) to all shared memory for all processes. We use the former definition to refer also to architectures with non-uniform memory accesses (also called NUMA systems) which are nevertheless shared in hardware. Cache coherence protocols allow processors in an SMP to use local caches and determine which values are read when a cached variable is updated by a different processor. A closely related concept is memory consistency, which defines *when* an updated value in a system would be guaranteed to be visible to other processors in the system. A memory consistency model is a contract between the system and the user, and a cache coherence protocol is implemented by the system to help ensure a particular memory consistency scheme.

It is often not easy for hardware vendors to achieve fast and symmetric access to all memory addresses from all processors, particularly for very large-scale systems. The more processors in the system, the more contention there will be on the memory bus. Sophisticated interconnect technology may resolve some of this latency, but costs can be extremely high. NUMA distributed shared memory systems are often deployed to allow for systems with large numbers of processors that share all memories. In this configuration, each processor has its own local memory that is close
to the processor for relatively fast access. Cache-Coherent NUMA (CC-NUMA) systems refer to distributed systems with caching mechanisms in place to keep distributed, shared data consistent. NUMA systems, in general, use hardware to hide the non-uniform memory characteristics, thus providing a global shared memory space for all processors. The size of such systems have continued to grow, and they have become key targets for many HPC applications. Programming models for such systems would ideally allow the programmer to take advantage of the global address space (i.e. a processor may directly access any globally available data without requiring explicit message passing invocations), while simultaneously making a distinction between accesses to local memory addresses versus remote memory addresses.

2.1.3 Distributed Memory Architectures

In Figure 2.2, we show a typical distributed memory system. In such a system, each processor owns its own local memory that is not directly accessible by other
processors. These processors are connected via an interconnection network, and communication over this network is carried out typically with some form of message passing. In these systems, the processors are working independently on their local data. If a processor needs data that is stored in another processor’s memory, then some combination of software and hardware support is required to execute the necessary communication.

Much as multi-processing eventually superseded increasingly complex and expensive uniprocessors, cluster computing gained in prominence as it was a more cost-effective strategy for scaling parallel systems compared to shared memory or distributed shared memory computers. Beowulf clusters [95], as an example, allowed commodity computing systems to be connected as compute nodes to a network, with system software enabling a parallel application to run over the various nodes in a coordinated manner. Cluster computing has become so ubiquitous in modern times that the list of the Top 500 supercomputers is dominated by them [3].
Configuration for a computer cluster is typically much easier and more cost-effective to configure, since they employ commodity off-the-shelf processors and networks. However, it is still a challenge to obtain efficient utilization of the entire system. Although much effort has been put into providing scalable networks, such as InfiniBand [83], Myrinet [14] and Gigabit Ethernet [31], for these systems, network communication is still very expensive compared to local memory accesses. Thus, to achieve good performance, programmers must pay particular attention to minimizing communication overhead in their applications when targeting these platforms. Data distribution over the system is also crucial to ensure, as much as possible, that computation is performed on locally-owned data.

Much effort has been invested into supporting high-speed interconnects for such systems. Different network topologies may be used to support efficient communication. Some research has gone into the direction of even adapting these topologies to a particular communication pattern. Examples of topologies for interconnect networks include star, ring, tree, hypercube, and n-dimensional (commonly toroidal) mesh. Finally, routing mechanisms are also necessary to facilitate data transfers across interconnect networks between machines that are not directly connected to each other. A key technology in modern interconnects is support for Remote Direct Memory Access (RDMA) [51]. Networks supporting RDMA enable one processor to write/read data across the network to/from a target remote memory without active involvement of the processor owning that target memory. RDMA may be used to support more efficient message passing protocols and is particularly well suited for programming models which make use of single-sided or one-sided messaging.
2.2 Parallel Programming Models

In this section, we describe the distinguishing features of the Partitioned Global Address Space, or PGAS, programming model. To do this, a brief explanation of what we mean by “programming model” is in order. We define a programming model as an abstraction for how the programmer may program a particular class of architectures. The term may be used to describe a broad class of programming interfaces which share some common features and assumptions about the underlying system, in which case it may be alternately referred to as a “programming paradigm”. Or, it may describe a rather specific manner of programming a type of computer system, in which case it may be directly associated with a particular programming language and/or library interface (thus, we sometimes informally refer to MPI [74] or OpenMP [30] or CAF [80] as “programming models”).

2.2.1 Traditional Parallel Programming Models

Programming models will typically define how data objects are referenced and stored (sometimes called a memory model), the manner in which computational tasks are defined and made to execute, and the operations or instructions that are supported. Aspects of this abstraction may be used as the basis for designing architectural features or programming systems (e.g. languages and/or libraries). We define the term execution model, in contrast, as an abstraction of the dynamic execution behavior of a program on a particular class of architectures. There may be a strong one-to-one correspondence between some features of a programming model and execution model.
supported on a system, which causes such confusion over the terms that they are often used interchangeably. Practically speaking, we use “programming model” as an abstraction for programming systems or interfaces, and “execution model” as an abstraction for the underlying machine.

The typical programming model that is commonly used when writing sequential programs would be the Von Neumann model [101]. This model assumes an architecture consisting of a central processing unit (CPU), a memory storage, and a connecting channel between the CPU and the memory (the implicit execution model is therefore sequential). It provides programming mechanisms for evaluating arithmetic and logical expressions, control flow, and moving data between the CPU and the memory. Historically, there has been debate on how appropriate the Von Neumann model is for handling various computing tasks; the main critique has been the bottleneck of the link between the CPU and the memory, typically realized in programming systems as an assignment statement [9]. Despite this, it has remained so ubiquitous that most programmers will naturally develop algorithms adhering to its sequential semantics, even though most computer architectures today consist of myriad parallel processing features.

In the last few decades, various parallel programming models have been proposed to help define and support modern parallel computing systems. Parallel programming models provide the programmer with an abstraction of how the system will execute the parallel application and how its data will be laid out in memory.

Parallel computers have a broad variety of architectural organizations, which we covered briefly in the preceding sections, and consequentially there have been many
different parallel programming models proposed. Clearly, the performance that is achievable with the use of a particular programming model depends on how well it matches the actual physical system that is being programmed. Broadly speaking, we can distinguish three models which have been traditionally used:

1. **Data Parallel Model:** Multiple threads are executing the same instructions, often in lock-step, but are operating on different data. This model is typically used for programming vector machines, or GPUs. We note that the term “data parallel” is sometimes used to describe programming models which support loop-level parallelism or partition computations based on data distribution and accesses.

2. **Shared Memory Model:** Multiple threads are executing their own instruction streams, and all have access to the same shared address space. The shared memory model is popular because it provides a flexible means to write parallel programs exhibiting task-level parallelism while not burdening the programmer with data partitioning responsibilities. The problem, of course, is that while having globally addressable data is convenient, the inability to partition this data to effectively optimize for data affinity and locality of reference can greatly inhibit scalability. Shared memory programming models often employ a fork-join model for parallel execution, where the execution begins with a single thread of execution, and additional threads are forked (or woken up) on demand for the execution of parallel code sections, and then the threads are terminated (or put to sleep).
3. **Distributed Memory Model:** Multiple threads of execution (generally processes with their own address spaces) are executing separate instruction streams, and each thread can only access data in its own address space. Coordination between threads is achieved via message passing, which can entail costly buffering overheads for large transactions and significant messaging overhead for small transactions. Distributed memory programming models commonly use a Single Program, Multiple Data (SPMD) model for parallel execution of the entire program. This means that the same program is executed by multiple processors in parallel, but they are generally executing at different points in the program and operating on different data.

### 2.2.2 Partitioned Global Address Space Model

In the Partitioned Global Address Space (PGAS) programming model, there is a partitioned global memory which may be accessed in a manner akin to shared memory, but with the programmer being able to distinguish between local and non-local accesses of this memory. The model also typically allows for data distribution across the memory partitions. Thus, data movement may be explicitly controlled by the programmer. Additionally, the model may also provide provisions for defining locality domains within the system and spawning tasks at specified locales.

Like other parallel programming models, the PGAS model (depicted in Figure 2.3) supports parallel execution of processing entities (which, depending on the
particular PGAS implementation, may be referred to as “threads”, “images”, “processing elements”, “nodes”, etc.). The distinguishing feature is that it provides mechanisms for directly accessing non-local data, separating accesses that are local from accesses that are non-local. A typical system which supports the PGAS model is expected, at a minimum, to have the following features (either via its architecture or through runtime services):

- a set of compute nodes, each consisting of one or more processors and local memory storage

- a capability for a processor in one compute node to fetch or modify data in a memory belong to a different compute node

- exposure of data affinity to the programming system

In contrast to the message-passing programming model, where non-local data accesses require active participation of a sender and recipient, the PGAS model
allows non-local data access to be achieved in a one-sided manner. This is akin to loading from or storing to an address in global memory (hence, “global address space”). At the same time, unlike the shared memory programming model, the model exposes affinity of data objects to processing entities. If a data object is a type of collection, then it may be distributed across the system in some manner which is exposed to the program. In other words, the program is, in a sense, aware of where its data objects reside in relation to the one or more processing entities that are executing. A consequence of this is that the program is also “aware” of where data objects reside in relation to each other.

Trends indicate that large-scale computers in the near future will be based on distributed memory platforms with an even higher number of nodes than today’s systems, and that the nodes will contain a large number of potentially heterogeneous cores [61]. With the growing core count, these nodes will be NUMA systems. The extent to which they support cache coherency among the cores is not yet clear. Parallel languages and libraries which use a PGAS programming model offer the potential to achieve scalable performance both across nodes as well as within these nodes with large core counts.

2.3 Compiling for Parallel Programming Models

Compilation for parallel systems is a broad topic which broaches several topics in compiler optimization and code generation:
• Instruction-level parallelism can be exposed through instruction scheduling and optimizations like software pipelining [6].

• There are many loop transformations which may be applied to expose loop parallelization opportunities, for which the compiler may generate vectorized code or thread-level parallelism.

• Parallel languages or APIs based on compiler directives (e.g. OpenMP [30], OpenACC [81], HPF [58], CAF [80], and UPC [39]) include special constructs to direct the generation of parallel code. OpenMP helps for generation of multi-threaded code, and OpenACC allows for the generation of code to be executed on an accelerator such as a GPU. Compiling languages such as HPF, CAF, and UPC require support for data distribution and efficient support for generating communication operations.

Programs, including those written in a sequential programming language, typically contain implicit parallelism that may be uncovered by the compiler or hardware. While auto-parallelization has been heavily studied [17, 67, 86], practically speaking progress has been limited. It has been most effective in extracting parallelism from loop nests through dependence analysis. Techniques such as data privatization [45, 68, 100], array padding to alleviate false sharing [59, 88], and cost models for parallelization of loops [34, 65] have also be explored.

There is a vast literature on techniques for compiling parallel programs for distributed memory systems. Much of this work was done in the context of High Performance Fortran (HPF) [4, 16, 21, 52]. This work encompasses the following concepts:
• **Data Distribution**: In some programming models, data distribution is implicit, therefore requiring the compiler to determine an efficient mapping of data structures to processors. \[46, 63, 64, 87\]

• **Computation Partitioning**: Computation partitioning techniques were developed for so-called “data-parallel” programming models, such as HPF. In such models, the compiler is responsible for assigning iterations of a loop to different processors in an efficient manner. The problem is similar to generating multi-threaded code for parallel loops, with the additional challenge of partitioning the iterations based on data that is being accessed. \[52, 92\]

• **Communication Generation**: This refers to the generation of communication, either 2-sided or 1-sided, for parallel programs. Array region analysis is required for some models, such as HPF, where such communication is implicit. Communication is far more expensive compared to local memory accesses. Therefore, key optimizations here include effectively overlapping communication costs with other local operations, as well as eliminating redundant communication. \[8, 20\]

• **Communication Vectorization and Chunking**: Communication vectorization refers to “vectorization” of communication operations within a loop, based on techniques analogous to producing vectorized instructions. This is an important optimization, as transferring large arrays of data can significantly improve utilization of a communication network’s bandwidth versus transferring
individual elements of an array. Other techniques such as chunking communication may also be applied to improve communication-computation overlap. Thus, application of both techniques can help increase bandwidth utilization while also hiding latency. [26,27,32]

- **Synchronization Optimizations**: Synchronization optimizations include the elimination of barriers, or conversion of barriers into more efficient synchronizations. Such optimizations are more relevant for 1-sided communication models rather than for 2-sided message passing models where synchronization is implicit in the point-to-point communication operations. [84,90,99]

### 2.4 Summary

In this chapter, we described the broad architectural trends in parallel computing, followed by a discussion of parallel programming models. We highlighted the relevance of the Partitioned Global Address Space (PGAS) model in light of the observed computing trends. We then briefly described concepts in compiling for parallel systems. In the remainder of this dissertation, we will turn our focus to understanding the PGAS programming model in more detail through examples, and then detail the work done to implement a PGAS language for supporting parallel execution of Fortran programs.
Chapter 3

Survey of PGAS Models and Implementations

In this chapter, we present a survey of some particular programming model implementations of the Partitioned Global Address Space (PGAS) model, introduced in Chapter 2 Section 2.2. We cover a variety of PGAS libraries which are available and widely used in HPC application development. We then describe several PGAS languages which have been proposed and have, to varying degrees, been adopted by the HPC community.

3.1 PGAS Libraries

Several libraries supporting the PGAS programming model are available, and here we light two: Global Arrays (GA) and OpenSHMEM. Both libraries come with
a set of routines for efficient and productive parallel programming on shared as well as
distributed memory systems. Along with one-sided communication operations, they
support remote atomic memory operations, broadcasts, reductions, and a simple set
of ordering, locking, and synchronization primitives.

3.1.1 Global Arrays

The Global Arrays (GA) toolkit [77,78] is a library which supports the development of
highly scalable parallel programs while working with arrays with a global index space.
GA is a PGAS library due to the fact that the user can control how the global arrays
are distributed and identify which portions of the global arrays reside locally. Much
like MPI, a GA program executes in SPMD-style; that is, a number of processes are
launched and may execute instances of the program independently. The GA toolkit
includes language bindings for Fortran, C, Python, and C++. uses ARMCI [75]
as its communication layer, and collective operations are generally supported using
MPI.

GA abstracts away the low-level details of which processes are communicating
with each other. Instead, the programmer defines how their logically global arrays
are to be distributed among the executing processes. Global arrays are created
(nga_create) and destroyed (nga_destroy) as a collective operation among all the
processes, and the arrays may be defined to use a wide range of data types and
practically arbitrary number of array dimensions. The GA toolkit provides routines
to copy data to (nga_put) and from (nga_get) a specified section of the global array. There is also a routine to copy data between two sections of a global array (nga_copy_patch). It is the responsibility of the GA library to take care of reading or writing the accessed data from the owning processes. To give more control to the programmer in managing locality, there are routines to determine which portions of a global array are locally owned (nga_distribute) and to access it (nga_access).

3.1.2 OpenSHMEM

OpenSHMEM [22] is a library interface specification, with bindings for C, C++, and Fortran, that unifies various specifications of the SHMEM programming API and adheres to the PGAS programming model. It is designed with a chief aim of performance, exploiting support for Remote Direct Memory Access (RDMA) available in modern network interconnects and thereby allowing for highly efficient data transfers without incurring the software overhead that comes with message passing communication. OpenSHMEM programs follow an SPMD-like execution model, where all processing elements (PEs) are launched at the beginning of the program; each PE executes the same code and the number of processes remains unchanged during execution. OpenSHMEM PEs are initialized when the start.pes function is called. We present in the following paragraphs the main features of OpenSHMEM.

OpenSHMEM supports the PGAS memory model where it proceeds by one-sided communications to transfer data between PEs. shmem_type_put is the function that copies contiguous data from a local object to a remote object on the destination PE.
**shmem_type.get** copies contiguous data from a remote object on the destination PE to a local object. Symmetric arrays are remotely accessible data objects, where by symmetric we mean that the object has a corresponding object with the same type, size and offset on all other PEs.

OpenSHMEM provides two basic types of synchronization: collective synchronization (e.g. **shmem_barrier.all**), which blocks until all other PEs issue a call to this particular statement, and point-to-point synchronization such as the **shmem_wait** primitive, which causes a PE to block until a specified memory location is updated.

OpenSHMEM includes several collective operations for performing reductions, broadcasts, and allgather operations. One type of collective communications is reduction. It is supported through the **shmem_operator.to.all** routine, which performs a reduction operation where **operator** can be one of several predefined reduction operators. OpenSHMEM currently lacks support for user-defined reductions. Mutual exclusion is implemented using locks of type integer. The **shmem.set.lock** function is used to test a lock and block if it is already acquired; the **shmem.set.unlock** function is used to release a lock.

### 3.2 PGAS Languages

Two prominent PGAS languages arose as extensions to C and Fortran, respectively: Unified Parallel C (UPC) and Coarray Fortran (CAF). There were also research efforts to create a PGAS extension to Java called Titanium. Although this remained a research project, it developed some of the implementation ideas used in adding PGAS
support for C programs. Additionally, several other PGAS languages or language
extensions have been proposed. We describe these languages in this section. More
recently, Asynchronous PGAS (APGAS) languages, with extended features for
defining logical locations (sometimes called ”places” or ”locales”) and spawning tasks
or ”asyncs”, have emerged. Examples include X10 [25], and Chapel [19].

3.2.1 Coarray Fortran (CAF)

CAF was initially a proposed extension to Fortran 95, and has recently been in-
corporated into the Fortran 2008 standard. As in UPC, there are a fixed number
of executing copies called images, and globally-accessible data are declared using
an extended array type called coarrays which have one or more codimensions. One
image may access a coarray on a different image by explicitly specifying its index
along its codimensions. CAF codes can leverage Fortran’s array section syntax to
specify bulk data movement. CAF also includes global and pairwise barriers and a
memory fence as part of the language. We describe the set of features comprising
CAF which are part of the Fortran 2008 standard, as well as a number of further
expected enhancements to the language for Fortran 2015 in Chapter 4. In Chapter 8,
we describe a design for parallel I/O in CAF programs.

3.2.2 Unified Parallel C

There have been many proposals for extensions to the C programming language
to enable its use in parallel programming without directives or other non-standard
features, including C* \[89\], Split-C \[29\] and Unified Parallel C (UPC) \[39\]. UPC stands out among these efforts, as there are multiple compiler projects presently supporting it \[1\],\[106\],\[107\]. However, to date there has been no formal standardization effort, such as that which has adopted coarrays into Fortran, to fold in UPC into the C standard.

UPC extends C with only a handful of new keywords. There are a fixed number of executing threads, and data items intended be globally accessible are declared with the `shared` keyword. It is possible to define how shared arrays are to be distributed across the threads and to query which thread a particular shared data item has affinity to by using the `upc_threadof` intrinsic. UPC also provides routines for barriers, bulk data movement, and work-distributing for-loops. Support for a variety of collectives such as broadcast, scatter, gather, and reduction is also a part of the UPC library API.

UPC is a superset of the C programming language which supports a SPMD execution of the program, and a partitioned global address space memory model. A running UPC program involves the concurrent execution of multiple instances of the same program, each of which is referred to as a UPC `thread`. The predefined constants `MYTHREAD` and `THREADS` return the local thread ID and total number of executing threads for the program, respectively. UPC provides barriers (`upc_barrier`, `upc_notify`, `upc_wait`), where a `upc_barrier` is equivalent to a `upc_notify` followed by a `upc_wait`. Split-phase synchronizations are possible using these primitives. Lock synchronization is also available using `upc_lock`, `upc_lock_attempt`, and `upc_unlock`. Locks may be used to implement critical sections or atomic memory
UPC adds keywords to the language to indicate whether a variable is shared among executing threads, as well as to specify the consistency model. Shared variables reside in a “shared” memory space which is logically partitioned among the threads. Shared arrays are distributed (by default, cyclically) among the memories for each thread, and different cyclic blocking factors may be specified as well. In addition to the shared keyword, a programmer may specify a consistency model on a variable declaration – either strict or relaxed. Accesses to a strict variable are treated as sequentially consistent, meaning all threads will observe accesses to strict variables as occurring in the same order. By contrast, accesses to relaxed variables are weakly consistent, thereby allowing an optimizing compiler to reorder them if it is profitable to do so.

UPC is a flexible model which enables writing parallel programs in many different styles. Achieving good performance depends on utilizing the best matching style for a given architecture. Explicit data transfer operations are available with upc_memget and upc_memput, where the source and destination buffers (respectively) may have affinity to a non-local thread. Another useful routine is upc_threadof, which takes as an argument a shared address and returns the thread ID for the thread which has affinity to this address. UPC also provides a upc_for_all construct, an extension to the standard C for loop, which adds an affinity expression as a fourth argument. If the expression is an integer \(k\), then the thread who’s thread ID is congruent with \(k \mod \text{THREADS}\) for a given iteration will execute that iteration. If the expression is an address, then whichever thread has affinity to that address will execute the
corresponding iteration.

3.2.3 Titanium

Titanium [105] is a parallel programming language, based on the Java dialect, which was designed to address a number of the needs in demanding scientific applications. These include time and space adaptivity, flexible data structures, and nondeterministic dependences. Some of the core language features added by Titanium on top of Java include multi-dimensional arrays, extended value types, and a new, unordered loop construct. Titanium also incorporates a static thread model, in place of Java threads, and a partitioned address space which enables locality optimizations.

Titanium follows the SPMD model of parallelism, where each executing entity is a thread. Barriers in Titanium are used to synchronize all threads, and additional these barriers must be textually aligned. This requirement assures that either all or none of the threads synchronize at a particular barrier. The language adds the notion of “single-valued expressions” (i.e. necessarily holds the same value across all threads), which aids in static detection of synchronization bugs (e.g. codes that can result in deadlocks) and static concurrency analysis. Titanium’s memory model is PGAS, distinguishing a local, private space and a global, shared space for data. The language provides an explicit means for allocating data in either of these spaces or for declaring references which may refer to data in either of these spaces. Hence, programmers have a means for managing and optimizing locality in their programs, essential for
achieving scalable performance on systems with NUMA distributed memory architectures.

3.2.4 CAF 2.0

CAF 2.0 [72] is a PGAS extension to Fortran 95 proposed by Rice University, intended to provide a richer set of capabilities than what was provided by Fortran 2008 coarrays (which we generally refer to simply as CAF). In a critique of the Fortran 2008 coarray model [71], several limitations were pointed out. This includes no support for dealing effectively with processor subsets, a lack of extensible collectives, a lack of latency-hiding synchronization constructs, lack of atomic operations, and no support for programs using logical topologies with periodic boundary conditions or graph topologies. The aim of CAF 2.0 was to address these language shortcomings, inspired by concepts used in MPI and the Asynchronous PGAS languages.

CAF 2.0 adds teams as first-class entities, with team_world being the initial team containing all images, and where new teams may be formed through the team_split or team_merge routines. Coarrays are dynamically allocated with respect to a specified team; if the team is not specified than they are allocated with respect to the “default” team (referred to as team_default). The default team may be changed by using a with team statement.

Additionally, CAF 2.0 provides support for graph or cartesian logical topologies, which may be defined and then associated with a team. All teams consist of a set of P images, with ranks 0 through P-1. A graph topology allows the programmer to treat
these images as nodes in a graph, and define sets of edges among the nodes (each of which is referred to as an edge class). Hence, each image has a set of neighbors with respect to a particular edge class. CAF 2.0 also supports cartesian topologies with a variable number of dimensions, including dimensions with periodic boundary conditions.

Coarrays are restricted to a single codimension, and indexing through this codimension selects an image from a specified team (the default team is assumed if the team is not explicitly specified). If the team does not have an associated topology, then the index specifies the rank of the image relative to the specified team. If the team has an associated graph topology, then the tuple \((e, i, k)\) is used as an index, which means access the kth neighbor of image i with respect to edge class e. And if the team has an associated N-dimensional cartesian topology, then an N-tuple specifying the indices for each dimension is required.

CAF 2.0 includes copointers, a special type of pointer which may point to a non-local coarray, facilitating algorithms that operate on distributed, pointer-based data structures. For mutual exclusion, CAF 2.0 provides locks and locksets. A lock is a global structure which may be held by only one image at a time, and a lockset are a set of locks which may be acquired and released as a group in a safe, deadlock-free manner.

For addressing the synchronization issues that were found in Fortran 2008, CAF 2.0 provides event synchronization and asynchronous versions of barriers and collectives. Events are counting semaphores which may be declared as a coarray, and the language provides \texttt{event.notify} and \texttt{event.wait} operations for point-to-point
synchronization on a specified event. The events feature provides a more flexible means for synchronization among images, and may be used to implement constructs like the Fortran 2008 \texttt{sync images} statement.

CAF 2.0 adds several features to support an asynchronous parallel execution model \cite{103}. It allows for non-blocking \textit{put} or \textit{get} communication through the \texttt{copy_async} routine. This routine also takes up to three event arguments: a predicate event which is required before the transfer takes place, a source event which will signal completion of the read at the source of the transfer, and a destination event which will signal completion of the write at the destination of the transfer. CAF 2.0 provides function shipping capabilities through the asynchronous \texttt{spawn} statement. Such a facility can help reduce data movement when needing to operate on and update remote data. CAF 2.0 provides support for a \texttt{finish} block with a team argument, which ensures global completion of all asynchronous operations across all members of the specified team. A \texttt{cofence} statement is also available, which provides a flexible means for controlling local data completion of asynchronous operations.

Split-phase barriers consist of two separate operations: \texttt{team_barrier_async} and \texttt{event_wait}. The first call will initiate the sending of notifications to other images in the team that the current image reached the barrier, and \texttt{event_wait} will block until all images in the team are known to have reached the barrier. In this manner, it would be possible to hide some of the synchronization cost of a barrier by inserting computation that does not depend on the completion of the barrier between the two operations.
For collectives support, CAF 2.0 proposes a variety of collectives, including broadcast, gather, allgather, scatter, reduce, allreduce, scan, shift, and sort. Moreover, for reductions the user may provide a user-defined function which implements a commutative operation on two inputs, as well as specify that the reduction operation is *replication oblivious* and may be consequentially implemented in a more efficient manner. As with barriers, CAF 2.0 provides asynchronous variants for the collectives, allowing each image to wait on an event signifying the completion of the operation.

### 3.3 Summary

In this chapter, we reviewed some prominent programming interfaces based on the PGAS model. This survey, while not by any means exhaustive, covers a representative sample of PGAS libraries and languages which have been well studied. The common feature of these programming interfaces is their emphasis on 1-sided, asynchronous remote memory accesses. We did not mention in this chapter the Message Passing Interface (MPI), which in its version 3 has greatly enhanced its support for such operations [53], and can therefore also be considered a “PGAS” library. Looking ahead, we expect that utilizing the PGAS programming paradigm, whether through MPI-3 or one of the other interfaces described in this chapter, will be an increasingly common practice in HPC. An important factor in how quickly such practices will be adopted is the standardization of these models. To date, only one of the “PGAS” languages described in this chapter is part of an ISO international standard. We cover this language in more detail in the next chapter.
Chapter 4

Coarray Fortran: A Closer Look

In this chapter, we describe the Coarray Fortran (CAF) parallel programming language. CAF is a subset of the Fortran 2008 standard which adds parallel processing capabilities to the base language by adopting a SPMD model for work decomposition and coarrays for data decomposition. CAF originated internally at Cray Research as an extension to Fortran known as $F--$, a parallel Fortran extension designed for programming the Cray T3D, and soon after evolved into Co-Array Fortran [80]. Support for coarrays in the Cray Fortran compiler has been available, in one form or another, for over two decades. Coarrays were eventually incorporated into the Fortran 2008 standard, though certain features found in the earlier Co-Array Fortran language were excluded, such as codimension sections, team synchronization, and parallel I/O. It is expected that the upcoming Fortran 2015 standard will include more additions to the CAF programming model, though the precise form these additions will take are still under discussion. We describe in this chapter some of the
4.1 Parallel Execution

A coarray program consists of a set of images running in parallel, following the SPMD mode (as is used, for example, in MPI programs). The number of images that is expected to run is specified outside the program code in some implementation-dependent manner, and it is fixed during the execution of the program. It may, for example, be set via a compiler option, through an environment variable, or by options passed to a program job launcher. Each image that comprises the running program has an unique image index which is between 1 and the number of images, and both the image index and the total number of images can be queried with the intrinsic subroutines this_image and num_images, respectively. Each running image is generally presumed to be running on a separate processor (though this is not required), and it performs computations on the data residing in its local memory. The data contained in an image’s local memory may be also accessible from other images – a single-sided remote access which is characteristic of PGAS models.

4.2 Coarrays

Coarrays are data entities (variables or variable components) which are declared with the codimension attribute specifier. Codimensions are analogous to array dimensions (given by the dimension specifier), in that they may be used to describe a
multi-dimensional rectangular shape associated with a data entity. While array dimensions on a variable describe the rectangular shape of the variable in the image’s local memory, codimensions describe the rectangular coshape of the set of images which each contain the (coarray) variable in its memory (which, as of the current version of the standard, is presumed to be all images).

Coarrays may appear as dummy arguments in a procedure, so long as the associated actual argument is also a coarray. Otherwise, a coarray may be declared with either the save or allocatable attribute. For non-allocatable coarrays, the codimension specifier should indicate the size of all codimensions, except the last one for which the upper-bound must be specified as an asterisk (since the total number of images is not to be specified in the code). Allocatable coarrays have a deferred coshape; the codimension bounds are specified on the allocate statement, again with an asterisk used for the last codimension. Just as the lbound and ubound intrinsic functions return bounds information for a given array variable with the dimension specifier, there are corresponding lcobound and ucobound intrinsic functions which return codimension bounds information for coarray variables.

Codimensions associated with a coarray provide a means to uniquely refer to a corresponding coarray on another image – referred to as image selection. When a coarray variable is referenced with cosubscripts (“cosubscripted” or “co-indexed”), expressed by trailing subscripts contained within square brackets, this denotes a remote memory access to the coarray at the image selected by the cosubscripts. The intrinsic function image_index may be used to return the image index of the image containing a specified coarray variable with a specified set of cosubscripts.
Cosubscripts must uniquely refer to a single image (i.e. cosubscripts may not contain subscript triplets and vector subscripts). Therefore, it is not possible to refer to coarrays across multiple images at once using a single coarray reference with cosubscripts. Coarrays are the mechanism in Fortran that enables images to perform remote memory accesses. In Fortran, a data entity is remotely accessible if it is (1) a coarray, (2) a pointer or allocatable component of a coarray, or (3) an object with the target attribute that is pointer associated with a pointer component of a coarray.

4.3 Atomic Subroutines

Atomic variables in Fortran are variables of integer or logical type which respectively have the atomic_integer_kind or atomic_logical_kind kind attribute. Currently there are only two atomic subroutines defined in the standard. The atomic_define subroutine may be used to atomically define an atomic variable; the variable may not be accessed by another atomic subroutine while it is being defined. The atomic_ref subroutine may be used to atomically reference an atomic variable; the variable may not be accessed by another atomic subroutine while it is being referenced.

1 These are introduced in Fortran 90 for referencing a multi-dimensional section of an array variable
2 Pointer assignment for coarray pointer components was added in Fortran 2008 and was not allowed in the original Co-Array Fortran specification
4.4 Image Control Statements

Coordinated execution among images is achieved through *image control statements*. Image control statements demarcate regions of code, referred to as *execution segments*, where an image is guaranteed to have exclusive access to any local or remote data objects it is modifying without the use of an atomic subroutine. The language contains several types of image control statements that provide various synchronization facilities to the programmer. The most fundamental of these is the *sync memory* statement, which acts as a local memory barrier (preventing the compiler or the hardware on which it is running from reordering local memory operations around it) as well as a *fence* for remote memory accesses, though it does not on its own perform any synchronization action with respect to a different image. The semantics for a *sync memory* is implicit in all image control statements, except *lock*, *unlock*, *critical*, and *end critical*. A *sync memory* may be used in conjunction with code that performs a *cooperative synchronization* action (utilizing an atomic subroutine) to realize custom implementations of synchronization primitives such as *notify/wait*.

A *sync all* statement is a synchronizing barrier; for all \( n \), any image that executes the \( n^{th} \) *sync all* will wait until all other images have reached their \( n^{th} \) *sync all* statement. There is also the *sync images* statement, which performs a pairwise barrier operation with every image referred to by the image indices in its *image-list* argument (that is, image \( Q \) waits for each image to execute a matching *sync images* statement with an *image-list* including \( Q \)). The *lock* and *unlock* statements may
be used in conjunction with coarray variables of type `lock_type` (which we refer to as lock variables). Locks are required to be coarrays, and a lock on any image may be acquired and released independently of the corresponding locks on other images (e.g. $L[P]$ and $L[Q]$ may be held by two different images, not necessarily $P$ and $Q$). The `critical` and `end critical` statements are used to define a critical section in the program which must be executed by only one image at any time.

### 4.5 Termination

A coarray program may terminate in one of two modes – *normal termination* or *error termination*. Normal termination is initiated by an image when it reaches the end of a program or executes the `stop` statement, and it proceeds in three steps – initiation, synchronization, and completion. All images in the program must synchronize at the second step to ensure that all images complete normal termination. Error termination occurs when an image meets an error condition. This could mean that it it encounters an error state as defined by the standard, meets some implementation-specific error state, that the program executed an `error stop` statement, or that it is notified that some other image is in error termination. When any image initiates error termination, the implementation should attempt to shutdown all images as early as possible and return an error status code.
4.6 Upcoming Features

The set of features introduced in Fortran 2008 for writing coarray programs provides a simple syntax for expressing one-sided communication and basic coordination mechanisms among executing images. However, as pointed out in [71], it lacks several important features required for writing scalable parallel programs. A technical specification document, *TS 18508*, is in the drafting process by the Fortran work group [102], and its purpose is to address many of these limitations. It describes features for coordinating work more effectively among image subsets, for expressing common collective operations, for performing remote atomic updates, and for expressing point-to-point synchronization in a more natural manner through events. In this section, we review the main additions being proposed.

4.6.1 Teams in Coarray Programs

Team support is being proposed for addition into the standard, and it provides a capability similar to the teams concept in CAF 2.0, described in Section 3.2.4. The initial team contains all the images and executes at the beginning of the program. All the images may join a new team with the `form team` statement. This statement will create a set of new sibling teams, such that all the images in the current team are uniquely assigned to one of these new teams. The operation is similar to the `mpi_comm_split` routine in MPI, or the `team_split` routine in CAF 2.0. An image has a separate `team variable` for each team it is a member of.

An image may change its current team by executing a `change team` construct,
which consists of the change team statement (with a team variable supplied as an argument), followed by a block of statements, and ending with the end team statement. The change team construct may also be nested, subject to the following constraints: (1) the team it is changing to must either be the initial team or a team formed by the currently executing team, and (2) all images in the current team must change to a new team. The image inquiry intrinsics this_image() and num_images() will return the image index and total number of images for the current team, respectively, though an optional team variable argument may also be supplied to return the image index or number of images for another team. Image selection via cosubscripts also will be relative to the current team, though syntax for allowing cosubscripts to select an image with respect to a specified team is also expected to be available. Collective operations are assumed to be among all images in the current team, including barriers via the sync all statement. The sync team statement will be added for an image to perform a barrier synchronization with a specified team, rather than only the current team.

4.6.2 Collectives Support

One of the major omissions in the coarrays feature set provided in Fortran 2008 was collective operations. To partially address this, support for broadcasts and reductions are expected to be added in the next standard. For reductions, the user may use one of the predefined reduction subroutines – co_sum, co_min, and co_max – or a more generalized reduction routine – co_reduce – which accepts a user-defined function as an argument. All of the reduction subroutines may optionally specify a result_image
argument, indicating that the reduction result need only be returned to one image (a *reduce* operation, rather than an *allreduce*). For broadcasts, the user may use the **co_bcast** intrinsic subroutine. The advantage of using these intrinsic routines rather than collectives support from a library such as MPI, is that the compiler can issue warnings or errors for incorrect usage. For each of these collective subroutines, there is no requirement that the arguments be coarrays, though an implementation may optimize for this case.

### 4.7 CAF Memory Model

An optimized coarrays implementation requires understanding the Fortran memory model. The memory model defines the rules governing how an implementation may handle memory accesses, including in particular their ordering with respect to one another. While this model is not made explicit in the standard, it can be reasonably inferred. For this discussion, we assume for now that memory accesses are not made using atomic subroutines.

Each image has its own local memory. This memory can be further divided into a *public* and *private* local memory. An image has exclusive access to its private local memory, and this access is not dependent on actions taken by other images. The public memory, in contrast, is accessible by all images, though concurrent access (i.e. accesses by two different images occurring in unordered execution segments) must be read-only (non-updating).

---

3While atomic memory accesses to the same variable in execution segments which are unordered with respect to eachother is permitted, these accesses are not sequentially consistent.
The program order sequence of memory accesses made by an image, whether to local memory or to a remote public memory, has the happened before relation. This means that the results of all memory accesses made by an image will be visible to that image in the same order in which those accesses were specified in the program. Suppose, for example, than an image writes a value to a data object in the public memory of another image. If it subsequently performs a read access to the same remote address, a value will be read only after the prior write to this address has completed.

If two images P and Q access the same variable X in the memory of some image, and one of the accesses updates the value of X, then the accesses must occur in ordered execution segments. If an execution segment S1 on image P is ordered before an execution segment S2 on image Q, this means that all statements occurring in S1 must complete before execution of the first statement in S2 if not doing so could alter the results of the program (i.e. all statements in S1 have the happened before relation with respect to the statements in S2). The implementation need not make any guarantees with regards to the order in which updates to variables are made visible (or whether they are made visible at all) to other images within unordered execution segments. However, a valid program will not attempt to access a variable that is updated by another image in an unordered segment; the implication of this is that in valid programs accesses to common variables by multiple images will be causally ordered and therefore it would not be possible for images to observe a different ordering of updates to any variable.

In effect, we could describe the consistency model as a release consistency model.
Such a model provides ample opportunity for employing data access optimizations by the system (be it in the compiler, the runtime system, or in the underlying hardware). When an image is seeking to update some variables, it first acquires write access to those variables. The updates are only made visible to another image once the writer “releases” its write access to those variables and the reader “acquires” read access to those variables. When the reader has completed reading the variables, it must “release” its read access. The model allows multiple images to simultaneously have read access to a variable. But if any image holds write access to a variable, no other image may have write or read access to it. In coarray programs, image control statements are used to partition the program into execution segments and define a partial ordering of execution segments across all images. The transition from one execution segment to another by an image includes an implicit acquisition of read/write access to variables that are (non-atomically) read/written in the next segment. Similarly, it entails an implicit release of read/write access to variables that are (non-atomically) read/written in the preceding segment.

**Semantics of sync memory** The `sync memory` statement is a memory barrier; it ends one execution segment on the image and starts another. All memory operations to an image Q that are in execution segments that precede the `sync memory` must complete before any memory operations to image Q in execution segments that succeed the `sync memory` are initiated, unless relaxing this constraint will not affect processing on other images. This includes memory operations to objects in the image, as well as memory operations to objects in different images (coarrays or coarray sub-components). The other constraint is that all memory operations on variables in
execution segments preceding a sync memory should complete before those variables are accessed by any image in subsequently ordered execution segment.

The simplest way to implement sync memory is to wait for completion of all local and remote memory operations. This is in accordance with the eager release consistency model, where updates made by an image with write access is guaranteed completion at the release of its write access. Alternatively, a lazy release consistency model guarantees that updates made by an image with write access are completed upon the subsequent acquisition of write or read access by some other image. Such a model could be potentially supported by an implementation, with the following rules:

1. All remote writes from the preceding segments must initiate and a record of these pending writes must be kept.

2. Notifications for all remote writes in preceding segments should be sent to and received by the respective target images.

3. (optional) Wait for all remote reads to complete.

4. All local memory operations should complete.

Then, we would need to additionally do the following:

1. On any remote access, all pending remote writes from a preceding execution segment should complete.
2. The system triggers completion of pending writes (potentially from other images, as notified) to a registered memory space when it is accessed by either a local or remote image.

4.8 Past and Present Coarray Implementation Efforts

There have been few implementations supporting coarrays in Fortran to date. A group at Rice University developed CAFC [33], a prototype implementation for a subset of Co-Array Fortran. It consisted of a source-to-source translator based on open-source Open64 compiler, with runtime support based on ARMCI [75] or GAS-Net [15]. The CAFC project is no longer active, having been supplanted by the CAF 2.0 effort. CAF 2.0 [72] uses the ROSE source-to-source compiler infrastructure, and the runtime is based on GASNet. Neither CAFC nor the current CAF 2.0 implementation supports the Fortran 2008 coarray specification. G95 [12] is another compiler that provides a partial coarray implementation, though runtime support for distributed memory execution is closed-source and the project is no longer active as of this date. G95 allows coarray programs to run on a single machine with multiple cores, or on multiple images across homogeneous networks. In this second mode, images are launched and managed via a G95 Coarray Console.

There has been an on-going effort to implement coarrays in GFortran [40]. The GFortran implementation uses the OpenCoarrays [41] runtime, which may be built on
top of MPI, GASNet, or ARMCI (the ARMCI implementation is mostly incomplete). The GFortran implementation is aiming to support Fortran 2008 coarrays, but as of this writing it has several limitations, including lack of support for vector subscripts, limited support for coarrays of derived type including no support for allocatable or pointer components, and incomplete support for locks or critical sections. Among vendor compilers, Cray has support coarrays for over two decades and is generally considered to be state of the art. Cray’s implementation is notable for being fully standard-adhering and delivering strong performance [94]. Intel has also provided support for Fortran 2008 coarrays, but performance has been reported to be very poor at this stage [93], and certain features (such as locks) are still unsupported.

4.9 Summary

In this chapter we presented an overview of Coarray Fortran, the set of language features added to Fortran 2008 for supporting the execution of coarray program. These features collectively allow data to be explicitly decomposed, provides an intuitive and syntactically concise mechanism for remote memory access, and provides image control statements for coordinating work among images. Beyond these core features defined in Fortran 2008, there are several language anticipated language features described in this chapter which we have implemented.
Chapter 5

Existing Infrastructure

In this chapter, we present OpenUH as an example of a modern compiler infrastructure which we have used to support programming models research, both for exploration of parallel language extensions as well as development of compilation and runtime system techniques for supporting applications in an HPC setting. This compiler infrastructure is the basis for the practical work presented in this dissertation and has been widely used by research groups in academia and industry. We also describe the runtimes libraries we have used to implement our CAF runtime. We close the chapter by briefly describing the experimental platforms we used for our evaluations.
5.1 OpenUH Compiler Infrastructure

OpenUH \[23, 66\] is a branch of the open-source Open64 compiler suite which researchers in the HPCTools group at the University of Houston have developed and used to support a range of research activities in the area of programming model research. A diagram depicting the overall compiler infrastructure for OpenUH is shown in Figure 5.1. The most prominent feature of OpenUH is its modern and complete framework for inter- and intra-procedural state-of-the-art analyses and optimizations. OpenUH uses a tree-based IR called WHIRL which comprises 5 levels, from Very High (VH) to Very Low (VL), to enable a broad range of optimizations. This design allows the compiler to perform various optimizations on different levels.

The major functional parts of the compiler are the C/C++ and Fortran front-ends, the interprocedural analyzer/optimizer (IPA/IPO) and the middle-end/back end, which is further subdivided into the loop nest optimizer (LNO), global optimizer (WOPT), and code generators (CG) for 32-bit and 64-bit x86 platforms. Additional features provided by this compiler infrastructure include the ability to emit source code from an optimized intermediate representation, as well as to selectively instrument the lowered intermediate code for low-overhead performance profiling.

The HPCTools group has undertaken a broad range of infrastructure development in OpenUH to support important topics such as language research, static analysis of parallel programs, performance analysis, task scheduling, and dynamic optimization \[5, 49, 50, 56, 57\]. We also in the past have investigated techniques for retargeting
OpenMP applications to distributed memory architectures and systems with heterogeneous cores [21,35].

OpenUH provided a solid base infrastructure for exploring implementation strategies for Coarray Fortran. The Fortran 95 front-end, originating from Cray, was already capable of recognizing coarrays and parsing the cosubscript syntactic extension. We were able to make use of this as a starting point in our implementation. The multi-level WHIRL IR, used throughout the back-end of the compiler, provides rich support for a wide range of program abstractions – from language level down to machine level. At its highest level of abstraction, VH-WHIRL, it is capable of representing Fortran array sections. This allowed us to design our back-end translator to directly map array section remote memory accesses into bulk communication transfers. The comprehensive optimization infrastructure available in OpenUH also provides a means for us to generate highly optimized coarray programs. OpenUH
also includes its own Fortran runtime libraries, providing support for the myriad intrinsic routines defined in Fortran, memory allocation, I/O, and program termination. We chose to implement our CAF runtime outside these other Fortran runtime libraries and reduce as much as possible its dependence on them. This would allow us to very easily port our CAF runtime to be used with different compilers.

The author’s contributions to OpenUH, apart from developing support for Fortran coarrays, include:

- Implementing compiler and runtime support for running OpenMP programs in a cluster environment \[35\]
- Extending the control flow graph and SSA IRs in the OpenUH global optimizer to model concurrency in shared memory parallel programs \[54, 55\]
- Developing efficient runtime support for task parallelism in the OpenUH OpenMP runtime \[43\]

## 5.2 GASNet

GASNet \[2, 15\], or the Global Address Space Networking library, is a language-independent runtime library developed and maintained by a team of researchers at the University of California Berkeley and Lawrence Livermore National Laboratory. The design of GASNet evolved to support the requirements of the UPC and Titanium implementations by the Berkeley team, but it has been widely used to support a variety of PGAS implementations including the Co-Array Fortran and CAF 2.0
efforts from Rice University, Cray’s UPC and CAF compilers for the CrayXT series, the Cray Chapel compiler, and the OpenSHMEM reference implementation developed by the University of Houston. GASNet has an API specification which defines interfaces for compilers or runtime systems to use, which we will briefly review in this section. GASNet was defined with portability as well as performance in mind, and it includes implementations for many network APIs covering the common cluster interconnects, as well as specialized interconnects from IBM and Cray.

A running program which uses GASNet, called the client program, consists of a set of operating system processes, or nodes. These nodes may furthermore be grouped into supernodes, which means they are controlled by the same operating system instance. Each node may attach a segment to its address space, which is a range of virtual addresses that are remotely access across GASNet nodes. Furthermore, the client program may execute in one of three threading modes – (1) GASNET_SEQ allows only a single thread on a node to make any GASNet calls, (2) GASNET_PARSYNC allows only one thread to make GASNet calls at a time, and (3) GASNET_PAR provides full multi-threaded support.

The GASNet API consists of a core API and an extended API. The core API provides all the facilities necessary to manage parallelism in the application, including runtime initialization and finalization, querying the number of GASNet nodes and the node ID, and mutexes for multi-threaded client programs. Perhaps the most significant portion of the core API is support for active messages (AM). The AM support provides means to invoke registered AM handlers on a remote GASNet node (which may in turn invoke a response handler on the requesting node). The AM
handlers are restricted to using only a subset of the core API, and in particular they may not use the extended API for communicating with other GASnet nodes. The core API also provides support for named or unnamed split-phase barriers.

The Extended API provides support for remote memory access (RMA) operations. This includes blocking get and put operations, explicit-handle non-blocking Get and Put, implicit-handle non-blocking get and put, and synchronization routines for waiting on completion of explicit-handle and implicit-handle RMA operations. While the specification presently allows only contiguous remote data transfers, Berkeley’s GASNet implementation also supports non-contiguous remote data transfers (referred to as “Vector, Indexed, Strided” or VIS).

5.3 ARMCI

ARMCI \[75,79\] is a portable remote memory copy library for accessing data from local and remote memory in a distributed memory system. It provides facilities for accessing remote memory without requiring the explicit cooperation of the process that owns the memory in a distributed memory environment. Hence, it decouples the sender and the receiver involved in the communication. Support of processor groups has been added to ARMCI more recently to support programming models that use multi-level or hierarchical parallelism. This feature was used in GA \[78\] where it was shown to be effective in improving scalability.

ARMCI has been used in the implementation of parallel distributed array libraries such as Global Arrays, as well as compiler runtime systems such as the
PCRC Adlib runtime system [18]. The ARMCI library is designed for optimizing non-contiguous data communication by aggregating small data packages into as few, large data packages as possible. The major ARMCI features include data transfer operations (e.g. get, put, and accumulate), synchronization operations (e.g. local and global fence, and atomic read-modify-write), memory utilities, and error handling.

ARMCI includes a number of mechanisms for hiding latency that can be exploited by high-level libraries or languages. These include (1) non-blocking data transfers for overlapping communication and computation, (2) a weakly consistent memory model with explicit interfaces for synchronizing memory operations, and (3) aggregation of small data transfers into larger messages to minimize sensitivity to network latency. The data transfer aggregation can also work with different types of get/put operations (i.e., regular, strided, and vector).

5.4 Experimental Platforms

In the next few chapters, we provide performance assessments of the developed methods for implementing CAF. The results were obtained using multiple systems, ranging from a small compute cluster to a state of the art Cray supercomputer. We describe these platforms in the remainder of this section.

**Whale** is a modestly sized cluster which was donated by Total to the University of Houston. It consists of 44 compute nodes, each with dual quad-core AMD Opteron processes running at 2.2GHz and 16GiB of main memory per node. The nodes are connected through a 4xDDR InfiniBand switch. Whale includes software
installations of GASNet 1.22.x, Open MPI 1.8.3, and MVAPICH2-X 2.0 (beta). For our experiments, we also locally installed the OpenUH compiler and runtime, as well as the CAF 2.0 compiler (version 1.14.0) and its software dependences.

**Stampede** is a supercomputer at the Texas Advanced Computing Center (TACC). It uses Dell PowerEdge server nodes, each with two Intel Xeon E5 Sandy Bridge processors (16 total cores) and 32 GiB of main memory per node. Each node also contains a Xeon Phi coprocessor, but we did not use this in our experimentation. The PowerEdge nodes are connected through a Mellanox FDR InfiniBand network, arranged in a 2-level fat tree topology. Stampede includes MPICH2 and MVAPICH2-X. We installed OpenUH and GASNet 1.22.4 on Stampede for our evaluations.

**Titan** is a Cray XK7 supercomputer at the Oak Ridge Leadership Facility (OLCF). Each compute node on Titan contains a 16-core 2.2GHz AMD Opteron 6274 processor (Interlagos) and 32 GiB of main memory per node. Two compute nodes share a Gemini interconnect router, and the pair is networked through this router with other pairs in a 3D torus topology. Titan nodes also contain NVIDIA Tesla GPUs, allowing the system as a whole to have a theoretical peak of 27 petaflops, but in this work we used only the host CPU. Titan includes the Cray Compiling Environment (CCE) which includes full Fortran 2008 support, including Coarray Fortran. Cray has supported coarrays in their compiler for two decades, and our goal was to develop an open-source implementation that offered performance on par with or exceeding their commercial product. Additionally, the Cray MPICH MPI implementation and the Cray implementation of OpenSHMEM are available. We installed OpenUH and GASNet 1.22.4 on Titan for our experiments, where we compare the
performance achieved with OpenUH to the performance using the Cray compiler. We also installed the recent GFortran 5.1 and OpenCoarrays implementation and incorporated its results in some of the tests. However, we found that the GFortran implementation as of this writing could generate correct code only for a portion of our tests, so we did not include it in all of our comparisons.

5.5 Summary

This chapter described the foundational software infrastructure that we used to build our implementation of Coarray Fortran. We selected OpenUH as the baseline Fortran compiler primarily because of its modular back-end support for high-level intermediate representations and its well-developed optimization phases. For the runtime system, we required a portable library designed to offer low-overhead 1-sided communication. GASNet and ARMCI are well established libraries, each used in multiple other PGAS implementation projects, and furthermore their interfaces were very similar. This made them both good candidates to use as a communication layer. We provide a compiler option to select between these two layers, as well as a third layer (OpenSHMEM). We also described three computing environment which we used for carrying out our experiments. In the next chapter, we describe the technical details of the techniques we developed for our implementation based on this infrastructure.
Chapter 6

Methods for Compiling and Executing Coarray Fortran Programs

In this chapter, we describe in detail the compiler and runtime methods we developed for compiling and executing CAF programs. We start with a high-level overview of the implementation, and then we cover the details of our runtime’s memory management scheme, compiler-generated intermediate buffering, hiding communication costs, supporting efficient strided communication, various techniques for supporting synchronization, and finally our method for supporting optimized collectives. The work discussed in this chapter was discussed, in part, in our publications [23, 36, 37, 60].
6.1 Overview of our Coarray Fortran Implementation

In this section, we present an overview of our CAF implementation in OpenUH, or UHCAF. This implementation comprises three areas: (1) an extended front-end that accepts the coarray syntax and related intrinsic functions/subroutines, (2) back-end translation, optimization, and code generation, and (3) a portable runtime library that can be deployed on a variety of HPC platforms. A diagram depicting the overall structure of this implementation is provided in Figure 6.1.

Figure 6.1: OpenUH CAF Implementation Stack
6.1.1 Front-end

We modified the Cray Fortran 95 front-end that comes with OpenUH to support our coarray implementation. Cray had provided some support for the CAF extended syntax. It could parse the [] syntax, it handled certain CAF intrinsics, and it translated the CAF language features to SHMEM-based runtime calls for targeting distributed systems with global address spaces. In order to take advantage of the analysis and optimizing capabilities in the OpenUH back-end, we needed to preserve the coarray representation into the back-end. To accomplish this, we adopted a similar approach to that used in Open64/SL-based CAFC compiler [33]: the front-end emits an intermediate representation which retains the cosubscripts as extra array subscripts. Furthermore, we extended our front-end to support the set of new intrinsics subroutines/functions defined in Fortran 2008 for the CAF model, added support for dynamically allocating coarrays (where the lower and upper bounds for all dimensions and codimensions are specified at runtime), and we extended our compiler’s internal representation for allocatable pointers/arrays to support coarrays.

6.1.2 Back-end

In the back-end, we added an initial phase call Coarray Prelower which normalizes the intermediate code (represented as an abstract syntax tree (AST) and emitted from the front-end) to facilitate dependence analysis. The primary purpose of the Coarray Prelower phase is to normalize the IR in order to simplify the translation of co-indexed references to the CAF runtime library communication interface. For certain
statements the Prelower phase will generate calls to this communication interface, but otherwise it will simply normalize the IR with translation being deferred until the Coarray Lower phase. The Prelower phase does the following:

1. Translation for certain Fortran intrinsic function calls specific to CAF

2. Translation to make use of local communication buffers (LCBs), where appropriate, for staging data to be communicated to and from the local image

3. Translation of save coarray and target object references to external, global pointers which will be defined at runtime

4. Translation of certain co-indexed references with array section subscripts, possibly non-contiguous, to array memory transfers via the CAF runtime library communication interface.

Next, we added a Coarray Lower phase in which the compiler will generate communication based on remote coarray references. This step currently occurs prior to the Fortran 90 (F90) lowering phase, during which elemental array operations and array section references are translated into loops. We make use of the higher-level F90 array operations, supported by our compiler’s very high-level internal representation, for generating bulk communication. AST nodes for representing single-element array references and array section references are processed to determine if they include co-subscripts, in which case it is a co-indexed reference. A co-indexed coarray variable signifies a remote access. A temporary local communication buffer (LCB) is allocated for either writing or reading the remotely accessed elements and functions as a
source or destination buffer, respectively. After coarrays are lowered in this fashion, their corresponding type in the compiler’s symbol tables are adjusted so that they only contain the local array dimensions.

Suppose the Coarray Lower phase encounters the following statement:

\[ A(i, j, 1 : n)[q] = B(1, j, 1 : n)[p] + C(1, j, 1 : n)[p] + D[p] \]

This means that array sections from coarrays B and C and the coarray scalar D are brought in from image p. They are added together, following the normal rules for array addition under Fortran 90. Then, the resulting array is written to an array section of coarray A on image q. To store all the intermediate values used for communication, temporary buffers must be made available. Our translation creates 4 buffers – t1, t2, t3, and t4 – for the above statement. We can represent the transformation carried out by the compiler as:

\[
A(i, j, 1 : n)[q] \leftarrow t1 = t2 \leftarrow B(1, j, 1 : n)[p] + t3 \leftarrow C(1, j, 1 : n)[p] + t4 \leftarrow D[p]
\]

For each expression of the form \( t \leftarrow R(...)[...] \), the compiler generates an allocation for a local communication buffer (LCB) \( t \) of the same size as the array section \( R(...) \). The compiler then generates a get runtime call. This call will retrieve the data into the buffer \( t \) using an underlying communication subsystem. The final step is for the compiler to generate a deallocation for buffer \( t \). An expression of the form

62
\( L(...) [\ldots] \leftarrow t \) follows a similar pattern, except the compiler generates a \textit{put} runtime call.

One of the key benefits of the CAF programming model is that, being language-based and syntactically explicit in terms of communication and synchronization, programs are amenable to aggressive compiler optimizations. This includes hoisting potentially expensive coarray accesses out of loops, message vectorization where the Fortran 90 array section syntax is not specified by the programmer, and generating non-blocking communication calls where it is feasible and profitable. For instance, a subsequent compiler phases may then convert \textit{get} and \textit{put} calls to non-blocking communication calls and use data flow analysis to overlap communication with computation and potentially aggregate messages.

The CAF Runtime Library public interface, used by the compiler back-end, is the principle translation target for the Coarray Prelower and Lower phases. It defines routines for:

- initializing and shutting down the CAF runtime
- allocation and deallocation of objects within the remote-accessible memory segment (used for data that may have the \texttt{target} attribute)
- acquiring and releasing local communication buffers
- initiating remote read (\textit{get}) and remote write (\textit{put}) memory accesses, with variants allowing for contiguous versus strided memory transfers, as well as blocking versus non-blocking memory transfers
• waiting on all outstanding non-blocking transfers to complete or a specified non-blocking transfer to complete

6.1.3 Runtime System

This work entails memory management for coarray data, communication facilities provided by the runtime, and support for synchronizations specified in the CAF language. We have also added runtime support for teams, reductions and broadcasts, point-to-point synchronization using event variables, and several atomic operations (including atomic op, atomic fetch-and-op, and atomic cas) – features which are expected to be part of the next Fortran standard revision. Our implementation relies on an underlying communication layer provided by GASNet [2], ARMCI [75], or OpenSHMEM [22]. We have adopted these libraries for the communication and synchronization operations required by the CAF execution model. While both GASNet and ARMCI provide similar APIs with respect to remote memory access operations, we found that GASNet offers additional flexibility for expressing asynchronous communication, and additionally its active message API was useful for implementing remote atomics. For this reason, our ARMCI implementation does not fully support the features described in Chapter 4. We therefore focus on our discussion in this chapter to our implementation on top of GASNet. Our runtime implementation over OpenSHMEM was a recent effort to take advantage of optimized OpenSHMEM implementations on certain platforms such as Cray’s. In particular, OpenSHMEM provides support for remote atomic operations which we could leverage for our implementation of event synchronizations, locks, and critical sections.
6.1.4 Incorporating Teams Support

OpenUH is able to parse the `form team, change team, end team` and `sync team` constructs. We added the new type `team_type` to the type system of OpenUH and support for `get_team` and `team_id` intrinsics. We also extended the usual CAF intrinsics `this_image, num_images` and `image_index` for teams. During the back-end compilation process in OpenUH, team-related constructs are lowered to subroutine calls which constitute the UHCAF runtime library interface. In the runtime, we add a `team_type` data structure for storing image-specific identification information, such as the mapping from a new index to the process identifier in the lower communication layer. Also we provide support for team-related intrinsics, for example `get_team` and `team_id`.

Before team support was added into our implementation, coarray allocation was globally symmetric across all images, with each coarray allocated at the same offset within a managed symmetric heap. With teams, however, this global symmetry can be relaxed. According to the Fortran language specification, symmetric data objects have the following features, which simplify the memory management for teams: First, whenever two images are in the same team, they have the same memory layout. Second, an image can only change to the initial team or teams formed within the current team. Third, when exiting a given team, all coarrays allocated within this team should be deallocated automatically. And fourth, if an image will refer to a coarray of another image located in a sibling team, the coarray should be allocated in their common ancestor team.
6.2 Remote Access Memory Management

In this section, we describe strategies for managing allocation of remote-accessible data objects in coarray programs. We distinguish two types of data entities. They may be remote inaccessible, meaning that they are only accessible by the local image, or they may be remote accessible, meaning they may be accessed by more than one image. In CAF, coarrays provide a mechanism for an image to access data belonging to another image. Data entities, which may be either static or dynamically allocated, are identified as remote accessible if they are one of the following: a coarray, a subcomponent of a coarray, or a target that is a pointer associated with a subcomponent of a coarray.

Enabling support for remote accessibility of data entails special runtime support. On shared memory systems, this may be achieved by utilizing shared memory segments (e.g. POSIX shared memory) and allocating remote accessible data within this segment. On distributed memory systems, a one-sided communication model providing efficient support for put, get, and perhaps also accumulate, may be used. Communication libraries providing one-sided support such as GASNet, ARMCI, OpenSHMEM, or MPI generally expose interfaces to request the allocation of a memory segment that supports efficient remote access, or (in the case of MPI) to even specify that an address range should be remotely accessible. Runtime systems which provided support for active messages (such as GASNet) can allow the invocation of a request handler on the target image. This enables more flexible accessibility to the target’s address space, though with the performance penalty that
comes with actively involving another processor to complete the access.

The use of a runtime-allocated memory segment for remote accesses will provide the most efficient means for remote memory access operations on distributed memory systems that support Remote Direct Memory Access (RDMA). Such a memory segment is pinned to physical memory, and its virtual address to physical address mapping is registered with the network adaptor. Efficient, asynchronous one-sided data transfers can be achieved by using this memory both at the sender side and the receiver side. If a target address in a remote memory access does not reside in this memory, then the implementation may fall back to utilizing a target-side request handler to complete operation. This is a viable option for supporting large remotely accessible data structures that cannot be accommodated in physical memory.

![Diagram of remote access memory segments](image)

**Figure 6.2:** Layout of data in remote access memory segments

In our implementation, we target distributed memory systems as the general case. At program startup, the runtime system allocates an equally sized remote access memory segment for each image. The size of the allocated segment is sum of the
combined total total size of the remotely accessible static data (determined by the compiler) and the image heap size which is set by of a user-controlled environment variable. By default, all remote accessible data is allocated within these segments, as shown in Figure 6.2. Once the segments are allocated, there is an all-to-all communication of the base addresses of these segments to all images. The images then set the base addresses of static remote accessible data (equally sized on all images) such that they are symmetrically positioned at the top their respective remote access segments. After this, each image reserves an equally sized range of addresses for coarray allocations within teams.

The remainder of the remote access segment is treated as a bidirectional remote access memory pool for either symmetric, synchronized allocations (from the top) for coarrays by the initial team, or for non-symmetric and unsynchronized allocations (from the bottom) for non-coarray data. This is handled using a data structure for managing both symmetric (i.e., coarrays) and non-symmetric (e.g., associated data of a coarray pointer component, or intermediate local communication buffers) allocations. The data structure is implemented as a doubly-linked list of slots, each of which describes a range of addresses which may be either allocated or non-allocated. The slots list is ordered based on the address range referenced by each slot. The allocation strategy is to efficiently make use of the available space in the managed heap. This is achieved by allocating from the top of the heap for symmetric allocations and from the bottom of the heap for non-symmetric allocations. One node in the slots list, designated the common slot, serves as the last available slot for either symmetric or non-symmetric allocations. When an object is allocated from this heap,
we search for an empty slot referencing an address range of sufficient size. We then either use this slot or split the slot in two if the address range exceeds the amount of memory being requested. When an object allocated from this heap is deallocated, the corresponding slot is marked as empty and merged with empty slots that may immediately precede or follow it in the slots list.

If a symmetric allocation is requested from this pool and there is not sufficient memory to satisfy the request at some image, then this image will abort with an error and send a signal to the other images to initiate error termination. If a non-symmetric allocation will result in a percentage of the heap being used which exceeds a specified threshold, then it may fall back to allocating the data out of system memory. In this case, remote access to this system memory is supported by our implementation by using an active message request handler.

6.3 Intermediate Communication Buffers

During the execution of a coarray program, there often arises the need to make use of intermediate local communication buffers. These refers to buffers which are not explicitly declared by the program but are rather implicitly created by the implementation to facilitate some remote memory access. Local communication buffers (LCBs) may be required to stage data to be communicated to and from the local image for a given encountered statement. The size of the LCB must be large enough to support the implicitly communicated data through a co-indexed reference in the program. An LCB may serve as the local destination buffer when the programmer
intends to “get” data from an image specified by the cosubscripts, or it may serve as the local source buffer when the programmer intends to “put” data to an image specified by the cosubscripts.

We describe in this section how the compiler translates various statements that may be encountered in a CAF program to employ the use of these communication buffers. Because there is an overhead associated with the allocation/deallocation of these buffers, as well as the memory copy between these intermediate buffers and a source or destination user buffer, we define when the use of such buffers is required by the implementation, and also when it may not be required by is nevertheless beneficial for performance.

### 6.3.1 Local Destination Buffers

For each co-indexed term that is read by the original statement, the program may acquire a corresponding $LCB_{dest}$. For each $LCB_{dest}$, the compiler inserts two statements before the original statement. First, an $LCB_{dest}$ is acquired through a runtime call to `__acquire_lcb` which specifies the size of the buffer requested. After this runtime call, the compiler inserts an additional assignment statement, by which the contents of the co-indexed term are written into the $LCB_{dest}$. This is eventually compiled to a `get` operation. Following this is the original statement, for which the compiler replaces each co-indexed term being read with a references to its corresponding $LCB_{dest}$. After the statement executes, each $LCB_{dest}$ is released using a runtime call to `__release_lcb`. 
6.3.2 Local Source Buffers

If the original statement writes to a co-indexed term (i.e., the term is on the left-hand side (LHS) of an assignment statement), the compiler may insert a call to acquire a corresponding $LCB_{src}$ using the same \texttt{acquire\_lcb} runtime call. Then, the term in the LHS of the original statement is replaced with this $LCB_{src}$. After this statement executes, the data to be put to the image specified by the co-indexed term is now in $LCB_{src}$. The compiler inserts a new assignment statement after the original statement, in which the contents of the $LCB_{src}$ are written to the co-indexed term, which will eventually compile to a put operation. Only once the put to the target image completes locally can the $LCB_{src}$ be safely released. The compiler does not assume that execution of this operation will guarantee local completion, allowing the runtime to execute it in a non-blocking manner. Therefore, the compiler will not insert the call to \texttt{release\_lcb} for $LCB_{src}$, but it will ensure that $LCB_{src}$ is not subsequently referenced. It is the responsibility of the runtime to “release” $LCB_{src}$ when it is safe to do so.

6.3.3 Cases for which an LCB is not used

Because there is an additional runtime cost in acquiring, copying data to/from, and releasing an LCB, we try to avoid using them when it's not required or when it does not facilitate early generation of bulk memory transfers. An LCB is not required for a statement if all of the following criteria are satisfied:

1.a The statement is an assignment statement.
1.b The RHS of the statement contains only one term, a scalar or array subobject (i.e. it is an l-value or an array of l-values).

1.c Either RHS contains a co-indexed reference or LHS contains a co-indexed reference, but not both.

1.d The co-indexed term, whether its in the LHS or RHS, does not include a type conversion.

1.e It is not the case that the LH has array section subscripts while the RHS does not.

To facilitate early generation of bulk transfers during the prelowering translation phase, we also will still use an LCB for the RHS of an assignment statement satisfying the above criteria if the following criteria are satisfied:

2.a The side having the co-indexed term has array section subscripts but does not contain a vector subscript.

2.b The side which does not have the co-indexed term has at least one vector subscript.

Example statements for which an LCB is not created during Coarray Prelowering phase (i.e. satisfies criteria 1.a through 1.e, but not 2.a and 2.b) include:
Furthermore, the compiler will use an LCB for the following statements (and similar statements), even though it is not strictly necessary to do so (satisfies criteria 1.a through 1.e, 2.a and 2.b):

\[
x(z(:)) = y(:)[i] \quad \rightarrow \quad LCB_{dest} = y(:)[i] \\
x(z(:)) = LCB_{dest} \\
y(:)[i] = x(z(:)) \quad \rightarrow \quad LCB_{src} = x(z(:)) \\
y(:)[i] = LCB_{src}
\]

### 6.4 Achieving Communication-Computation Overlap

We describe in this section how we designed our implementation to enable coarray programs to effectively exploit communication-computation overlap. The CAF weak
memory model, discussed in Section 4.7, provides ample opportunity for an implementa-
tion to overlap communication with ensuing local computation – that is, to
initiate communication in a non-blocking manner and only wait for its completion
when necessary. Since remote memory accesses, particularly in distributed memory
systems, are an expensive operation, hiding the latency by utilizing non-blocking
remote data transfers is important for achieving good performance.

Both the compiler and the runtime play a role in supporting communication-
computation overlap. The compiler is responsible for generating either a blocking
one-sided communication call, or a handle-returning non-blocking communication
call. In the latter case, wait operations on the returned handle must be inserted
into the code to ensure program correctness. Remote read operations need to return
data to the origin/destination buffer before it is read or rewritten (this is a local true
or output dependence constraint). The compiler may either insert an explicit wait
operation for the data transfer to be completed at the latest possible point while pre-
serving this dependence, or the runtime would have to block until the completion of
the read on the read call itself. Remote write operations must locally complete prior
to the origin/source buffer being modified (a local anti-dependence). The compiler
may insert explicit wait operations for the data transfer to locally complete at the
latest possible point while preserving this dependence. Alternatively, the runtime
call for a remote write operation may block until local completion.
6.4.1 RMA Completion Semantics

In a sequential program, two accesses to a common memory location X are expected to execute in program order if at least one of the accesses is a write. For coarray programs, this expectation extends to accesses in local memory as well as remote memory (i.e. the memory of a different image). This rule thus imposes an ordering constraint for remote memory accesses on the implementation. We enumerate the 6 ordering constraints with respect to remote memory accesses:

C1 All remote read operations must complete before the local destination is read or modified (local true dependence/output dependence).

C2 All remote write operations must locally complete before the local source of the write is modified (local anti dependence).

C3 All remote read operations must complete before any subsequent write operation to the same location initiates (remote anti dependence).

C4 All remote write operations must complete before any subsequent read or write operation to the same location initiates (remote true/output dependence).

C5 All remote read operations must complete before the image finishes execution of any subsequent image control statement\(^1\) (remote anti dependence).

---

\(^1\)More precisely, remote read operations must complete before the image finishes execution of any subsequent image control statement that implies sync memory semantics or must initiate after the previous lock statement or critical statement. Remote write operations must complete before the image finishes execution of any subsequent image control statement that implies sync memory semantics or must complete before a subsequent unlock or end critical statement.
All remote write operations must remotely complete before the image finishes execution of any subsequent image control statement (remote true/output dependence).

6.4.2 Memory consistency modes

We now define a number of memory consistency modes supported in our runtime which abide by the completion constraints described above. These modes start from a strong consistency model and gets progressively more relaxed.

1. All remote accesses block until global completion.

2. All remote read accesses block. All remote write accesses block until local completion. All remote write accesses must remotely complete prior to initiation of any subsequent remote read or write access. An image control statement will additionally ensure all pending remote write accesses have completed.

3. All remote read accesses block. All remote write accesses block until local completion. All remote write accesses must remotely complete prior to initiation of any subsequent remote read or write access on the same image. An image control statement will additionally ensure all pending remote write accesses have completed.

4. All remote read accesses block. All remote write accesses block until local completion. All remote write accesses must remotely complete prior to initiation of any subsequent remote read or write access of the same object in memory.
An image control statement will additionally ensure all pending remote write accesses have completed.

5. The compiler will generate a handle-returning non-blocking read, and a wait on the returned handle will be inserted at the latest position along any path extending from the read which statically guarantees adherence to rules C1 and C3. Similarly, the compiler will generate a handle-returning non-blocking write, and a wait on the returned handle (which ensures remote completion) will be inserted at the latest position along any path extending from the write which statically guarantees adherence to rules C2 and C4. The non-blocking write generated by the compiler may, for small-message writes, copy to an internal buffer and return a null handle, indicating to the program that the source buffer is free to be modified. Note that in this implementation strategy, the compiler takes care of inserting wait operations which satisfy C1-C4. Thus, the runtime need only take care of ensuring that all pending read and write accesses have completed at image control statements.

In our approach, the CAF runtime will perform remote read operations in a non-blocking manner, assuming a wait or synchronization operation has been inserted correctly either by the programmer or the compiler to preserve local true and output dependencies. Remote write operations will, by default, block until local completion. For small message transfers, this may be achieved by locally buffering the data to be transmitted. For large message transfers, however, where a zero-copy protocol is typically employed, blocking until local completion may entail essentially waiting for the entire message to be delivered to the target. To mitigate this cost when
it can more precisely determine the point to which a wait can be postponed, the compiler may optionally generate non-blocking write calls which do not wait for local completion.

### 6.4.3 Compiler Directives for CAF

CAF does not provide a language mechanism to explicitly control the use of non-blocking communication, as for instance is available with MPI using `mpi_isend` and `mpi_irecv`. This puts responsibility on the compiler to automatically make use of non-blocking `get` and `put` communication while preserving correctness using data flow analysis. While these optimizations are still in development, we have in place a directive-based mechanism to give the programmer some measure of control in affecting communication-computation overlap.

```plaintext
1 !dir$ caf_nowait(h1)
2 y(:) = x(:,p)
3 !-------------------
4 ... computation ...
5 !-------------------
6 !dir$ caf_wait(h1)
7 z(:) = y(:) * j
```

Figure 6.3: Using directives to control communication-computation overlap

Figure 6.3 illustrates the usage of these directives. `h1` is a declared variable of an intrinsic type, implemented internally as a pointer, that acts as a handle for representing a single outstanding non-blocking `get` operation. The `caf_nowait` directive says that the following statement should initiate a non-blocking read but the compiler should not generate a corresponding `caf_wait` (i.e. wait on the access to
Completion of the operation will only be ensured if a subsequent synchronization statement is encountered or a `caf_wait` directive with the specified handle is encountered. We only support the use of `caf_nowait` directive for assignment statements with a co-indexed variable as the only the term on the right hand side. This is sufficient to allow the user to specify and control the completion of non-blocking read operations.

### 6.4.4 Remote Memory Access Interface

A `get` or `put` remote memory access, at minimum, is specified with the target image index and the source and destination base addresses. For a contiguous memory transfer, the number of bytes to transfer is given. For a strided memory transfer, instead of the total number of bytes to transfer, the following 4 parameters are given: (1) a count array of positive integers, which when multiplied together is equal to the total number of bytes to be transferred, (2) the number of stride levels, equal to the length of the count array minus one, (3) a source strides array, and (4) a destination strides array. Taken together, these 4 parameters describe a multidimensional array section within the source and destination buffers.

To support non-blocking memory transfers, additional arguments must be given to control completion. Here, “completion” means that the transferred data has been fully received into the destination buffer. First, we give as an additional argument the address of an uninitialized non-blocking communication handle. The runtime will
initiate a non-blocking transfer and copy the contents of an internal communication-tracking handle into this address if the transfer has not completed, or a nil handle (all bits set to 0) into the address if the transfer has completed. If the address given is null, then a non-blocking memory transfer will be initiated, but the runtime will not provide the communication handle back to the program.

For supporting non-blocking put accesses, another parameter to be given is a boolean ordered which indicates whether the put access should be completed in order with respect to a subsequent access by the local image to the same address. By setting ordered to false, the ordering constraint is relaxed and the put is permitted to be asynchronous with respect to subsequent accesses. Only a synchronization statement or wait operation will force its completion.

Finally, we add variants of the put routines for when the source buffer is a local communication buffer. These variants will make the runtime responsible for “releasing” (or deallocating) these buffers once its associated put access has completed. Furthermore, these routines may safely assume a contiguous source buffer which will not be modified by the program. Hence they may return as soon as the transfer is initiated ensuring local completion (i.e., the source buffers contents have been written out or copied by the runtime).

**Waiting on Completion of Non-Blocking Accesses** The runtime exposes a routine for waiting on the completion of a prior non-blocking access or all outstanding non-blocking accesses. The routine takes a single parameter, handle, which is a non-blocking communication handle from the program. If handle is a wait_all handle
(all the bits are set to 1), then the runtime will wait on all outstanding non-blocking accesses to complete. If it is a nil handle (all the bits are set to 0), then the runtime will do nothing and return. Otherwise, it is assumed that it is a valid handle which tracks a single outstanding access, and the runtime will wait for this access to complete.

### 6.5 Supporting Non-contiguous Remote Data Access

Our design philosophy here is to efficiently handle communication patterns which can be easily and concisely expressed using CAF syntax. This requires recognizing and efficiently supporting multi-dimensional strided transfers observable in the program. Both GASNet and ARMCI provide interfaces for multi-dimensional strided transfers. We describe our compiler algorithm to map generalized remote array section accesses containing triplet subscripts to this abstraction.

#### 6.5.1 Communication Generation

We assume that the code has been normalized to use LCBs where appropriate, as discussed in Section 6.3. Thus, all co-indexed references with array section subscripts in the IR will occur in assignment statements. These statements will be of one of the following forms, where NVAS stands for “Non-Vector-subscript, Array Section,” and VAS stands for “Vector-subscript, Array Section”.

81
The runtime does not currently provide interfaces for non-contiguous, vector array memory transfers, though this may be supported in the future. Therefore, the Prelower phase will translate a co-indexed reference with array section subscripts to an array memory transfer only if neither the LHS or RHS term has a vector subscript. So, that would be statements of these forms:

<table>
<thead>
<tr>
<th>RemoteRead</th>
<th>RemoteWrite</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \text{ArrRef}<em>{NV\ AS} = \text{ArrRef}</em>{NV\ AS}[...] )</td>
<td>( \text{ArrRef}<em>{NV\ AS}[...] = \text{ArrRef}</em>{NV\ AS} )</td>
</tr>
<tr>
<td>( \text{LCB}<em>{\text{dest}} = \text{ArrRef}</em>{NV\ AS}[...] )</td>
<td>( \text{ArrRef}<em>{NV\ AS}[...] = \text{LCB}</em>{\text{src}} )</td>
</tr>
<tr>
<td>( \text{ArrRef}<em>{\text{NV\ AS}} = \text{ArrRef}</em>{\text{VAS}}[...] )</td>
<td>( \text{ArrRef}<em>{\text{VAS}}[...] = \text{ArrRef}</em>{NV\ AS} )</td>
</tr>
<tr>
<td>( \text{ArrRef}<em>{\text{VAS}}[...] = \text{ArrRef}</em>{\text{VAS}}[...] )</td>
<td>( \text{ArrRef}<em>{\text{VAS}}[...] = \text{ArrRef}</em>{\text{VAS}} )</td>
</tr>
<tr>
<td>( \text{LCB}<em>{\text{dest}} = \text{ArrRef}</em>{\text{VAS}}[...] )</td>
<td>( \text{ArrRef}<em>{\text{VAS}}[...] = \text{LCB}</em>{\text{src}} )</td>
</tr>
</tbody>
</table>

If, after normalization in the Prelower phase, either side of the assignment statement containing a co-indexed reference contains a term with a vector subscript, then we defer its translation until the Coarray Lower phase in the compiler back-end.

A contiguous array memory transfer may be used when the compiler can statically determine that both the source (RHS) and destination (LHS) buffer in the assignment statement reference a contiguous array section in memory. If the compiler cannot
statically verify that the assignment involves contiguous array sections, then it will make use of the strided array memory transfer interfaces\(^2\).

The algorithm for statically determining whether either side of the assignment statement \textit{may not} refer to a contiguous array section is given in Algorithm \(^1\).

\begin{algorithm}
\caption{Algorithm used to statically check for potentially strided transfer}
\begin{algorithmic}
\Procedure{possibly_not_contiguous}{access\_range, extent, first\_dim, last\_dim}
\State integer : \texttt{requires\_complete\_access}
\State \texttt{contig\_so\_far} \equiv \text{TRUE}
\For {d = last\_dim \text{ downto first\_dim}}
\If {access\_range(d) < extent(d) and \texttt{contig\_so\_far}}
\State \Return \text{FALSE}
\ElsIf {access\_range(d) == 1}
\State \Continue
\ElsIf {access\_range(d) > 1}
\State \texttt{contig\_so\_far} \equiv \text{TRUE}
\EndIf
\EndFor
\State \Return \text{TRUE}
\EndProcedure
\end{algorithmic}
\end{algorithm}

A contiguous array memory transfer, which may be either a \textit{get} or \textit{put} call (\texttt{comm\_nbread} and \texttt{comm\_write}, respectively), is specified with the target image index, source and destination base address, number of bytes to transfer, a non-blocking transfer handle, a boolean \textit{ordered}. This last item is used to indicate whether the non-blocking transfer should implicitly complete on a subsequent access to the destination buffer (for \textit{put} only). If it so happens that the source buffer is an \textit{LCB\_src} for a \textit{put} call, then we utilize the \textit{put} call variant for writing from an LCB (\texttt{comm\_write\_from\_lcb}).

\footnote{We are assured that non-contiguous array memory transfers may be executed using the strided APIs if neither array section reference contains a vector subscript.}
6.5.2 Generating and Executing Strided Array Accesses

An optimized strided array memory transfer is used if either the source (RHS) or destination (LHS) buffer in the assignment statement may reference a non-contiguous array section in memory (as determined by the algorithm given in Algorithm 1) and no term on either side contains a vector subscript. Translating a given assignment statement into a strided array memory transfer requires a careful processing of the array subscripts of each term on the RHS and LHS. The information that needs to be supplied to the runtime, as described in Section 6.4.4, is: the base addresses of the source and destination buffers, a count array indicate the extent along each dimension of the array section to be transferred, a \( \text{stride\_levels} \) argument which is equal to the length of the count array minus one, and \( \text{src\_strides} \) and \( \text{dest\_strides} \) arrays of length equal to \( \text{stride\_levels} \). There are two cases that we need to handle as depicted in Figure 6.4: (a) neither the RHS nor the LHS contains a triplet subscript with a non-unit stride, and (b) at least one triplet subscript with a non-unit stride is present in the RHS or LHS.
**Runtime Support for Strided Communication** The runtime implementation of the strided interface described in Section 6.4.4 leverages the strided interfaces provided by the underlying GASNet or ARMCI communication layer. These lower layers can speed up strided communication by exploiting underlying hardware...
support in some cases. More generally, it will perform packing/unpacking of non-contiguous data on the remote target, utilizing an active message handler or a helper thread. While there is an overhead in the packing and unpacking operations and may potentially limit asynchronous progress, the bandwidth benefits of sending a larger, contiguous buffer can make this an attractive solution.

Performance Assessment We evaluated the performance of our strided communication support using the EPCC Fortran Coarray microbenchmark suite\cite{48} on Titan using 256 total images across 16 nodes, and show the results in Figure 6.5. For these benchmarks, a total of 32K double precision elements are written or read across the Gemini network, with varying strides between each consecutive data item in the source and destination buffers. The single tests (Figures 6.5a and 6.5b) entail only communication from image 1 to image 256, while the multiple tests (Figures 6.5c and 6.5d) have the first 128 images communicating with the last 128 images.

In our evaluation we compared the Cray compiler implementation (CCE 8.3.4), the recent GFortran implementation (5.1) which uses OpenCoarrays, and our OpenUH implementation executing over GASNet and over OpenSHMEM. The Cray implementation relies on the scatter/gather network support. The GFortran implementation uses OpenCoarrays, which in turn used MPI-3 (via Cray MPICH) for the 1-sided communication. Our implementation, when using GASNet, will employ active messages to either unpack the written data at the target, or to pack the data to be read at the target. When implementing over Cray OpenSHMEM, we make use of the \texttt{shmemb\_double\_iput} and \texttt{shmemb\_double\_iget} operations, which in turn will use the network support for scatter/gather operations. We can see from the results
that for strided put our implementation performs the best when using GASNet, and when using OpenSHMEM it performs close to the Cray compiler implementation. For strided get, our GASNet implementation performs significantly better than the Cray implementation, and our OpenSHMEM implementations performs identical to the Cray implementation. The GFortran strided support, which utilizes MPI derived data types and RMA support, performs exceptionally well for strided get, but relatively poorly for strided put.
6.6 Synchronization

PGAS languages separate the cost of data transfers and the cost of synchronization, in contrast to 2-sided communication programming models where these two operations are implicitly coupled. Till now, we have discussed issues relating to efficient communication generation. In this section, we describe support for efficient synchronization in CAF. We consider here support for barriers (\texttt{sync all} and \texttt{sync team}), pairwise synchronization (\texttt{sync images} as well the event synchronization extensions), and mutual exclusion (\texttt{lock/unlock}). As these are image control statements, there is also an implicit memory synchronization action included in their semantics, which we describe. The \texttt{sync all}, \texttt{sync team}, and \texttt{sync images} entail waiting on some set of other images to reach a matching \texttt{sync all}, \texttt{sync team}, and \texttt{sync images}, respectively. The language requires that these statements abort if an image detects that it is attempting to synchronize with a \textit{stopped} image, so we discuss our approach to handling this behavior in an efficient manner as well.

6.6.1 Compiler Support for Image Control Statements

As described in Chapter 4, image control statements are used to partition a program in \textit{execution segments} and implicitly have memory fence semantics. An optimizing compiler must consider these semantics to avoid potentially unsafe code motion across an image control statement. Even if the statement is translated into an opaque library call, the compiler may still (unsafely) assume that non-global variables (for instance, a procedure-scope coarray with the \texttt{save} attribute) will not be modified
upon completion of the image control statement. It may consequentially perform an illegal optimization, such as moving an assignment to the coarray across the image control statement.

One method for preventing such illegal optimizations would be to introduce an optimization barrier into the AST, before and after our image control statements. OpenUH provides two types of optimization barriers, forward barriers and backward barriers. As the names imply, these barriers prevent the optimizer from moving program statements across them, in either the forward or backward direction. We could conservatively place a forward barrier before all image control statements, and a backward barrier after all image control statements. For the lock and critical statements, which are used to enforce exclusive execution of a series of statements, it would be sufficient to just insert a backward barrier after the unlock and end critical we could just insert a forward barrier before them. That is, we prevent remote accesses occurring within the guarded region in the program from being executed outside this region to ensure these accesses are mutually exclusive.

OpenUH provides a more refined means for more precisely modeling side effects of image control statements than what may otherwise be ascertained by the data flow analyzer, through the Hashed SSA (HSSA) representation [28]. In HSSA, a \( \chi(a) \) node may be attached to a statement, signifying that the statement may define the variable \( a \). We assume the same semantics for image control statements with respect to remotely accessible variables – they may be modified by some image upon completion of the statement. Hence, any statement which accesses a remotely accessible variable will not be reordered with respect to an image control statement. In order to further
reduce the set of potentially modified variables and make our enforcement of
dependences more precise, a concurrency analysis would be required. We consider
this as a future enhancement we would like to explore.

6.6.2 **sync all and sync team**

The **sync all** and **sync team** statements require that all images in a team, the
current team for **sync all** and the specified team for **sync team**, wait for completion
of any pending communication and then synchronize at a barrier. Moreover, if
while executing the barrier an image is detected to have entered a termination state,
then the other images must abort the barrier operation and will either enter error
termination or return a **stat_stopped_image** status to the program.

To execute a barrier for all images in the initial team, the typical case when teams
are not created by the program, we could simply use the barrier routines provided
by the underlying runtime library (GASNet, ARMCI, or OpenSHMEM). However,
these barriers would only permit the detection of stopped images prior to entering
the barrier, but not during the execution of the barrier itself. For the more general
case of executing a barrier among images in a non-initial team, we implemented
a dissemination barrier using one-sided communication facilities provided by the
underlying runtime. The dissemination barrier proceeds in log $P$ steps for a team of
$P$ images, where on step $k$ image $i$ will send a notification to image $i + 2^{k-1} \mod P$
and wait for a synchronization notification from image $i - 2^{k-1} \mod P$ or a signal
that some image in the team has stopped. If an image receives a notification from
image $i - 2^{\log P-1} \mod P$ on the final step, it can be assured that all $P$ images in the team have reached the barrier. Furthermore, if at least one image in the team terminates without reaching the barrier, then this information will propagate to all other images in the team by the final step.

We enhanced our barrier implementation by taking advantage of node locality information provided by the underlying runtime layer. Within the runtime descriptor for a team there is a list of image indices in the team that reside on the same node and a list of image indices in the team which are designated leaders of their respective nodes. Using this structure, we implemented a 2-level team barrier algorithm as follows. When an image encounters the barrier, it will either notify the leader image of its node, or if it is itself a leader it will wait on notifications from the non-leaders. Once the leader image has collected notifications from all other images in the same node, it engages in a dissemination barrier with other leader images in the team. Finally, once the dissemination barrier completes the leaders will notify its non-leaders that all images in the team have encountered the barrier. We implemented two approaches for the delivery of barrier notifications within a node. In the shared counter scheme, each non-leader image will atomically increment a shared counter residing on the leader image within the node to signal that it reached the barrier, and it will wait for a synchronization flag to be toggled by the leader to signal that the rest of the images have also reached the barrier. In the multi-flag scheme, each non-leader image will set its own synchronization flag to indicate that it has reached the barrier, and the leader image of the node will cycle through each of its non-leader’s flags until they are all set. The non-leader will then simply wait for its
synchronization flag to be unset by the leader, indicating that all the rest of the images in the team have arrived. We found in our experiments that neither scheme had a clear advantage over the other. We used the shared counter 2-level algorithm for the following evaluation.

**Performance Assessment** To assess the performance of our 2-level barrier algorithm, we ran experiments on Whale. We compared the implementation in OpenUH with two other implementations we used for this evaluation: (1) a CAF 2.0 version, for the source-to-source Rice CAF 2.0 compiler based on ROSE \[85\], using GFortran as the backend compiler, and (2) an MPI version using MPI communicators rather than temas, which we ran using MVAPICH2. Both the OpenUH and Rice CAF 2.0 implementations rely on GASNet’s InfiniBand Verbs runtime implementation. In Figure 6.6, we compare the various implementations to highlight the benefits of the 2-level implementation for team barriers in our CAF runtime. The classical dissemination algorithm is used in both the UHCAF Dissemination and CAF 2.0 implementations. Our 2-level barrier implementation significantly improved on this performance, getting us performance on par or exceeding the MVAPICH communicator-based barrier implementation.
6.6.3 sync images

The `sync images` statement provides a means for an image to synchronize with a list of other images in the current team. The statement implies the semantics of `sync memory`, forcing completion of any remote memory accesses. The image will then perform a pairwise synchronization with every image listed in the statement's argument list. In contrast to the `sync all` barrier statement, `sync images` is not a collective operation, but rather a point-to-point operation.

For the statement `sync images(L)`, where $L$ is a list of images, the implementation must send a `sync-images` notification to all images in $L$ and then wait for a
sync-images notification from all images in L. Our initial algorithm for sync images, which we refer to as the counter algorithm, utilizes a synchronization buffer holding status variables corresponding to every running image (so, O(N) space requirement per image). Sending a notification to an image I is simply a matter of atomically incrementing the status variable assigned to it at image I. Waiting on a notification from image I requires waiting until the local status variable assigned to I is non-zero and then atomically decrementing this value. In this scheme, every status variable has only one image that may increment it, and only the local image may decrement it. It is therefore guaranteed that the status variable value will be 0 when there has been an equal number of notify and wait actions on it, and it will be greater than zero when there are more notify actions than wait actions. It can never be less than zero, because the local image will only decrement the count once it observes that it is greater than zero, and no other image may decrement it. Moreover, we know that the value can never be greater than 2, since this would indicate a scenario in which another image has sent a second notification before the local image could finish waiting on the first notification. Hence, we see that a synchronization status variable, under this algorithm, can be in only one of 3 states:

- **state 0**: equal number of notify and wait actions performed on it
- **state 1**: one more notify than wait actions performed on it
- **state 2**: two more notify than wait actions performed on it

The counter algorithm is straightforward to implement and can work sufficiently
well on systems that offer efficient remote atomic increment support. However, because GASNet does not as of yet provide this, we devised another algorithm which does not require remote atomic updates, which we call the sense-reversing algorithm. In this algorithm, we exploit the fact that pairwise synchronization ensures that a pair of synchronizing images will necessarily synchronize with each other the same number of times. We can therefore allocate a sense variable which keeps track of whether the images have synchronized with each other an even (sense value = $E$) or odd (sense value = $O$) number of times. Instead of sending a notification by performing an atomic increment of the remote status variable, we write the status value $E$ or $O$, alternating between the two for each successive notification action on the variable. Waiting on a status variable entails blocking while the variable is non-zero, and then atomically exchanging a value of 0 into the status variable. The value returned may be either $E$ or $O$. If the value matches the locally maintained sense variable for the pair, then both images are synchronized. If the values do not match, then the other image must be one notification “ahead”. In this case, we simply write back the value into the status variable. Hence, as in the case of the counter algorithm, the sense variable may again be in one of 3 states, namely 0, $E$, or $O$.

**Performance Assessment** To evaluate the performance of the sync images algorithms, we ran microbenchmarks on Titan. For these experiments, we used 16 images per compute node. We tested four patterns, with results shown in Figure 6.7. In the all-to-all pattern, every image performs pairwise synchronizations with all other images. In the many-to-1 pattern, image 1 performs pairwise synchronizations with all other images. In the ring pattern, for a team of $P$ images, each image $i$
synchronizes with (1) image $i - 1$ or (if $i = 1$) image $P$ and (2) image $i + 1$ or (if $i = P$) image 1. And in the recursive doubling pattern, for $P$ images where $P$ is a power of 2, each image will execute $\log P$ steps, synchronizing with one other image on each step. In step $s$, image $i$ will synchronize with image $i + 2^{s-1}$ if $\text{mod}(i, 2^s) \leq 2^{s-1}$, or with image $i - 2^{s-1}$ if $\text{mod}(i, 2^s) \geq 2^{s-1}$.

The all-to-all and many-to-1 patterns scale linearly with the number of participating images, and is proportional to the cost of sending a notification to the synchronizing partner. The high-overhead associated with executing remote atomic increments using active messages with GASNet results in poor performance at scale for the counter algorithm. The sense-reversing (SR) algorithm performs on par with Cray’s CAF implementation for these two patterns. For the ring pattern, most of the synchronizations are occurring among images residing on the same compute node. Both our counter and sense-reversing algorithms outperform the Cray implementation by utilizing shared memory to synchronize with images in the same node. Finally, for the recursive doubling pattern, we again can see the advantage of using the lower-overhead sense-reversing algorithm for sending notifications on each step.
In Figure 6.8, we compare the performance of all sync images tests from the EPCC Fortran Coarray microbenchmark suite on 64 compute nodes from Titan. In this comparison we used the sense-reversing algorithm in our implementation, the Cray Fortran compiler, and the recent GFortran compiler which uses the OpenCoarrays runtime. OpenCoarrays was configured to use the MPICH MPI implementation provided by Cray. The first test, a pairwise synchronization, has each image synchronization with one other image. In the ring tests, the \( p \) images are logically arranged in a ring where image 1 is connected to image \( p \) and image 2, image \( p \) is connected
Figure 6.8: **sync images** tests from EPCC microbenchmark suite using 1024 images over 64 nodes on Titan

to image 1 and image \( p - 1 \), and otherwise image \( i \) is connected to images \( i - 1 \) and \( i + 1 \). In test **ring** \((n)\), each image synchronizes with its \( n \) closest neighbors in the ring. In the **random** \((n)\) tests, each image synchronizes with \( n \) other images, where all synchronization pairs are chosen randomly. Finally, the **3D grid** has each image logically arranged into a 3D torus image grid, where each image has up to 6 other images to which it connects. Across all these synchronization patterns, our **sense-reversing** implementation is consistently the fastest.
6.6.4 Atomics Support

In our implementation, we support 32-bit and 64-bit remote atomic operations. Our (default) GASNet implementation provides support for atomic definition and reference, atomic updates (add, and, or, and xor), atomic fetch-and-updates which return the old value, and atomic compare-and-swap, through the use of active messages. Our alternative ARMCI implementation only supports atomic definition and references, atomic add, and atomic fetch-and-add, since ARMCI does not provide remote atomics for other operations nor does it provide a functional remote procedure call facility. The use of active messages for support atomic operations is less than ideal, however, since it requires involvement from the handler thread at the target image. In the future, it is expected that GASNet will provide native support for remote atomics which should deliver significantly better performance. In our more recent OpenSHMEM-based implementation, we directly use the corresponding atomic operations defined in the OpenSHMEM specification, thus taking advantage of implementations which use the network’s native support for remote atomics.

6.6.5 Event Synchronization

Events variables are treated as a counting semaphores in our implementation, and are required to be coarrays. The event post statement is implemented as a memory synchronization which waits for completion on all communication, followed by an atomic increment operation on the event variable at the target image. The event wait statement, which may only operate on the event variable in the local image,
will busy-wait until the count is non-zero, at which point it will perform an atomic
decrement operation. Finally, the `event_query` intrinsic will simply return the count
of the specified event variable through an atomic reference.

### 6.6.6 Coarray Locks

The MCS queue-based lock [73] is a well known locking algorithm for scalable shared
memory systems which avoids spinning on non-local memory locations. At any time,
an object corresponding to the lock variable may be associated with a queue of
threads which are trying to acquire the lock. The queue has the following properties:

- it is first-in, first-out
- each node in the queue belongs to a unique thread, and is referred to as a `qnode`
- each `qnode` has a pointer to its successor
- the first `qnode` (head of the queue) belongs to the thread holding the lock
- the last `qnode` (tail of the queue) is pointed to by the lock object itself

Each image owns a lock L, and this lock may be acquired by any image.

**lock:** When a thread tries to acquire a lock, it specifies the address of the lock
along with its qnode. The thread will first push its `qnode` onto the current queue
of threads waiting for the lock. This is achieved with an atomic swap operation on
the lock object; namely, the thread will swap in the address of its `qnode` and receive
the address of its predecessor’s \textit{qnode}. The thread then will write its qnode’s address into the \textit{next} field of the qnode belonging to its predecessor. It will then wait until it has received the lock from its predecessor, by polling the \textit{locked} field in its \textit{qnode} until it is zero.

\textbf{unlock:} When a thread tries to release a lock, it first needs to check if it is the last thread in the queue. This is done with an atomic compare-and-swap operation on the lock object. If the lock object is still pointing to its \textit{qnode}, then a NIL value will be written into it. If the lock object is pointing to a different \textit{qnode}, then that means there is a successor in the queue that is waiting for the lock. The \textit{next} field in the thread’s \textit{qnode} will (eventually) point to the successor’s \textit{qnode}. The thread will wait until this \textit{next} field is updated, again a local spin, and then it will notify its successor that unsetting the \textit{locked} field of the successor’s \textit{qnode}.

The original algorithm was designed for shared memory systems, and we extended it for use in our CAF runtime system. The following changes were made:

1. The runtime manages allocation and deallocation of qnodes (from the non-symmetric registered memory space), rather than the client program providing them. A hash table is used to keep track of currently allocated qnodes, with the key being the location of the lock (its offset within the remote access segment, and the image it resides on).

2. The tail pointer within the lock object now needs to refer to a \textit{qnode} at a particular image. We use a tuple containing the index for the image (with respect to the initial team) and the offset of its \textit{qnode} within its remote access
segment. We reserve 20 bits for the image index, and 36 bits for the offset, with the remaining 8 bits used for the status of the lock.

3. The qnode itself has 4 fields: locked, image, ofst, and done, where the tuple $(image, ofst, done)$ represent the next field in the original algorithm.

**Performance Assessment** In Figure 6.9, we used the lock synchronization test from the *EPCC Fortran Coarrays microbenchmark suite* to assess the performance of our locking implementation in comparison to the implementation provided by the Cray compiler. In this test, every image competes to acquire a lock on image 1, then it performs some work that takes a fixed amount of time, and finally it releases the lock. We calibrated the benchmark so that the time taken to do the work upon acquiring the lock was the same when using the OpenUH and Cray compilers. The plots show the added overhead in acquiring and releasing the locks. We ran the tests using our CAF implementation running over GASNet as well as our CAF implementation running over the Cray OpenSHMEM library. Since our locking algorithm uses the remote atomic swap and compare-and-swap operations, the performance of our GASNet implementation suffers since it relies on active messages, particularly when we scale up to 16 images per node. The implementation on top of Cray’s OpenSHMEM allows us to perform these atomic updates in a truly 1-sided, asynchronous manner (especially important for PGAS implementations). The results show that both our GASNet- and OpenSHMEM-based implementations performed better than the Cray CAF implementation when using a single image per node, as the node count increased up to 1024. When using 16 images per node, the GASNet implementation of our runtime scaled very poorly due to the active messaging overhead, while the
OpenSHMEM-based implementation outperformed the Cray CAF implementation for 16K images.

![Figure 6.9: Coarray locks test from EPCC microbenchmark suite on Titan](image)

(a) 1 image per node  
(b) 16 images per node

6.7 Optimized Collectives

While there was no provision for collective subroutines in Fortran 2008, these are expected to be added to the language in a future revision. In anticipation of this, we developed support for fast collectives within our runtime. We currently support `reduce`, `allreduce`, `broadcast`, and `allgather` collective operations. The `reduce` and `allreduce` support handles both pre-defined reduction operations (`sum`, `min`, and `max`) as well as user-defined reduction operations. Among these, the `allgather` support is used exclusively to facilitate formation of teams, and was implemented using Bruck’s algorithm with 1-sided communication. The `reduce`, `allreduce`, and `broadcast` implementations are used for the corresponding intrinsic subroutines – `co_reduce`, `co_sum`, `co_min`, `co_max`, and `co_broadcast`. We implemented the respective binomial tree
algorithms for reduction and broadcast, and the recursive doubling algorithm for allreduce. Each of these algorithms complete in $\log P$ steps for $P$ images in a team, where on each step pairs of images are communicating – either a write from one image to the other (reduce and broadcast), or an exchange between the images (allreduce). Our collectives require that all images in the current team participate. Special care is needed to ensure that the writes among the images are well-synchronized. We developed a number of techniques to make these operations fast, in particular to hide or eliminate synchronization costs among the communicating images, which we describe in this section.

### 6.7.1 Basic Scheme

Our collectives implementation makes use of a collectives buffer space, a fixed size, symmetric region which is reserved from the remote access memory segment. If this buffer space is large enough, it will be used to carry out any of the data transfers required during the execution of a collective operation. Otherwise, all the images in the team will synchronously allocate a symmetric region of memory from the heap to serve as the buffer space for the execution of the operation. In order to carry out a reduce, broadcast, or allreduce operation, each image will reserve from its buffer space a base buffer. The base buffer is used to hold the result of each step in the collective operation. Additionally, for reduce and allreduce each image will reserve at least one work buffer. The work buffers are used to hold data communicated by a partner on each step, which will be merged into the base buffer using the specified reduction operation. Our binomial tree reduction and broadcast algorithms assume that the
root image will be image 1 in the team. Therefore, we incur the cost of an additional communication to image 1 (for broadcast) or from image 1 (for reduce) when this is not the case. In the case that these operations are operating on large arrays which cannot be accommodated within the buffer space available, we can break up these arrays into chunks, and perform our algorithm on each of these chunks in sequence.

6.7.2 Multiple Work Buffers

For the reduce and allreduce operations, on each step intermediate results are written by some subset of images into their partner’s work buffer in that step. Supposing each image reserved only a single work buffer, an image must first wait for a notification that its partner’s work buffer is not being used before initiating this write. In order to reduce this synchronization cost, we can have each image reserve up to Q work buffers, where \(1 \leq Q \leq \log P\). Images can send out notifications that their work buffers are ready to be written into by their next Q partners on just the first step of every Q steps of the reduce or allreduce operation. In this way, the cost of the notifications can be amortized over the execution of Q steps.

6.7.3 Hide Startup Synchronization Cost

Even if we can partially hide the cost of \((Q - 1)/Q\) notifications using the multiple work buffers strategy, we would still incur the synchronization penalty for the first step of the reduction operation, and the same would apply for broadcasts. This is because an image should not start writing into its partner’s buffer space until
it is guaranteed that the partner is not using this space (i.e. the partner is not
still executing a prior collective operation which is using the space). If the images
performed a synchronous heap allocation of its buffer space because there was not
sufficient room in the pre-reserved collective buffer space, then all images can safely
assume that the new space is not being used for a prior operation. If operating with
the collectives buffer space, then some synchronization would be needed on the first
step, which we refer to as the startup synchronization cost.

To hide this startup cost for the reduce, allreduce, and broadcast operations,
we employ a double-buffering scheme. In this scheme, we partition our collectives
buffer space into three pairs of buffer spaces. These are a pair of reduce buffer
spaces (reduce_bufspace1 and reduce_bufspace2), a pair of allreduce buffer spaces
(allreduce_bufspace1 and allreduce_bufspace2), and a pair of broadcast buffer spaces
(bcast_bufspace1 and bcast_bufspace2). If at any time more than one collective oper-
ation by a team is in progress, then we enforce the constraint that these operation-
specific buffer spaces may be used only for their respective collective operation.
Hence, these spaces are not exclusively reserved for their respective collective op-
erations, but rather are prioritized for overlapped execution of those operations.

For an allreduce operation, our double-buffering scheme allows us to eliminate
the need for any startup synchronization, so long as the allreduce buffer space is of
sufficient size. If we consider the execution of 3 consecutive allreduce operations,
the first operation may use reduce_bufspace1 and the second operation may then use
reduce_bufspace2. For allreduce, it is guaranteed that the second operation will not
complete until all the images have completed the first operation. Consequentially,
the third *allreduce* operation may freely use *reduce_binspace1* without concern that the space may be still being used by a prior operation.

For the *reduce* and *broadcast* operations, our binomial tree algorithms still would require a startup synchronization, but this cost may be effectively hidden. Specifically, once an image completes the execution of either *reduce* or *broadcast* while operating on either buffer space 1 or 2, it will send out notifications indicating this buffer space is free to all images that will write to it on a subsequent *reduce* or *broadcast* operation. The cost of these notifications may therefore be amortized over the period between the operation’s completion, and start of the next operation of the same type which will use the same buffer space.

### 6.7.4 Reducing Write Completion Synchronization

Because we use 1-sided communication for all data transfers during the execution of our collective operations, the receiver of the data must wait for some notification from the sender that the data transfer has completed. The image performing the write could perform a blocking *put* for the write, followed by a notification that the *put* has completed at the target. However, this would incur the additional cost of waiting for the completion of the first *put* and sending the completion notification back to the target. To reduce this additional overhead, we take advantage of the cases where the delivery of data into the target memory is guaranteed to be byte ordered, which to the best of our knowledge is the case for all current hardware implementations of the InfiniBand standard.
For platforms where *inter-node* data delivery is known to be byte ordered, we provide the *use_canary* setting for our collective operations. With this setting, all buffers which are used, the base buffer and the work buffers, are initialized to hold only zero values. This is ensured by always zeroing out the buffers upon completion of a collective operation. Furthermore, we reserve an extra trailing byte for all the buffers. When performing an inter-node write operation during the execution of a collective, an image may then append a value of 1 to the write message which will occupy this last byte at the destination buffer. The receiver need only poll on this byte, waiting for it to become non-zero, and it can then be guaranteed that the entirety of the message was delivered.

### 6.7.5 Exploiting Shared Memory

The latency for inter-node remote memory accesses is typically significantly higher compared to intra-node memory accesses. Therefore, a reasonable strategy for collective operations that exhibit a fixed communication structure (which is the case our *reduce*, *allreduce*, and *broadcast* algorithms) is to restructure the communication in such a way that minimizes that required inter-node communication. This is especially important when dealing with collective operations for a team of images, where member images may be distributed across the nodes in a non-uniform manner. We therefore developed 2-level algorithms for these collective operations, similar to the 2-level dissemination barrier algorithms we described earlier in the chapter. Each compute node which has at least one image in the team has a designated leader image, and all the leaders in the team have a unique *leader index*. 
When performing the reduce or allreduce operation, there are three phases. In the first phase, team members residing on the same compute node will reduce to the leader image. In the second phase, the leaders, will carry out either the reduce or allreduce operation among themselves. After this second phase, the first leader has the final result for reduce operation, and all leaders have the final result for the allreduce operation. In the third and final phase, for the reduce operation the first leader will write the final result to the root image, if it is not itself the root image. For the final phase of the allreduce operation, each leader image will broadcast its result to other images on its compute node. Depending on the particular topology of the compute node, this intra-node broadcast may be implemented using a binomial tree algorithm, or by simply having all the non-leaders on a node read the final result from the leader with the requisite synchronizations.

For the broadcast operation, the three phases are as follows. In the first phase, the source image will first write its data to the first leader image in the team. In the second phase, the leaders will collectively perform a binomial tree broadcast. In the final phase, each leader will broadcast its result to the non-leaders on its node.

### 6.7.6 Experimental Evaluation

In Figure [6.10] we show timings for the co_sum operation on Stampede with various optimization incrementally applied. uhcaf-baseline refers to the basic scheme for our collectives implementation including the double-buffering technique for hiding
startup synchronization costs. `uhcaf-workbuf` shows the performance when each image additionally uses \( \log P \) work buffers. `uhcaf-2level` furthermore adds the shared memory support within each node. Finally, `uhcaf-usecanary` incorporates our optimization for write completion synchronization, described above, which we verified as safe for the InfiniBand network on this system.

We observe that adding the shared memory ("2-level") support to our implementation had the biggest impact, consistently across all message lengths. For smaller
message lengths (less than 64K), the use of additional work buffers provided some improvement, but for larger message lengths the overhead of managing multiple work buffers offsets reduction in synchronization. The use_canary setting for reducing write completion synchronization also provided benefits for smaller message sizes, but this advantage vanished when dealing with larger messages.

![Graphs showing performance comparison]

Figure 6.11: co_sum (allreduce) performance on Titan using 64 nodes, 1024 images

Figure 6.11 shows timings for the allreduce co_sum operation using 1024 images over 64 compute nodes on Titan (so, 16 images per node). We compare our implementation with the Cray compiler’s CAF implementation. Cray’s implementation is well-tuned for reductions with very small message lengths (≤ 16 bytes), but for larger messages performs poorly relative to our implementation. The OpenUH implementation benefits here from the use of double-buffering to hide startup synchronization, multiple work buffers, and using shared memory for reduction within each compute node. On this platform, the optimization for write completion synchronization was not used, as Titan’s Gemini interconnect does not guarantee byte ordering of writes to remote memory.
6.8 Summary

In this chapter, we described in detail the design and implementation of a compiler and runtime system for Coarray Fortran (CAF) programs. We described our technique for managing the allocation of data to be remote accessible by other images. We presented our compiler translation approach for creating intermediate communication buffers for staging data to be sent to or received from another image. Our approach for supporting asynchronous communication, allowing the overlap of communication and computation, was then described. Next, support for non-contiguous, remote data access was presented. We then described in detail our support for synchronization constructs in CAF, including preserving correctness through our compiler translation, and synchronization algorithms we developed in our runtime. Finally, we detailed our method for optimized collective operations in our runtime system.
Chapter 7

Experimental Evaluation with OpenUH

In this chapter, we present results from several benchmarks and applications which we have used to study the effectiveness of our CAF implementation strategies. Our evaluation using the reverse time migration code from Total was detailed in [37].

7.1 High Performance Linpack

We implemented a Coarray Fortran version of High Performance Linpack (HPL) [82], used to solve systems of linear equations and provides a good benchmark for exploiting temporal and spatial locality. We based our version of HPL on its CAF 2.0 port, described in [44]. HPL makes use of row and column “teams” for performing updates of the matrix data. Beside the initial teams, the coarrays implementations of HPL
uses four total teams and two levels of nested teams.

We ran our HPL experiments on the Whale cluster. We compared the CAF version of these benchmarks using OpenUH with the CAF 2.0 version developed at Rice University. Both the OpenUH and Rice CAF 2.0 implementations use GASNet’s InfiniBand Verbs runtime implementation. We used GASNet 1.22.2 for our experiments. Figure 7.1 compares the performance results using the two-level approach in UHCAF, the one-level approach in UHCAF, CAF 2.0 using GFortran as backend compiler, and CAF 2.0 using OpenUH as backend compiler. The -O2 option is passed to the compilers in Figure 7.1a, while -O3 is used in Figure 7.1b. These preliminary results show that using the two-level approach in UHCAF provides up to 32% improvement over a typical one-level approach. Overall, we obtained 95 GFLOPS/s when using either -O2 or -O3 on 256 cores, as compared to 29.48 GFLOPS/s obtained with the CAF 2.0 implementation (with -O2 or -O3 for the GFortran backend) and 80 GFLOPS/s (with -O2 or -O3 for the OpenUH backend).

7.2 Distributed Hash Table

The benefits of the 1-sided communication support provided by PGAS languages like CAF can be easily understood for codes that exhibit unstructured or unpredictable remote access patterns. A typical example would be an application that uses a distributed structure that requires random access. For this purpose, we made use of a coarray version of a distributed hash table code, described in [69]. In this code, each image will randomly access and update a sequence of entries in the distributed hash
Figure 7.1: High Performance Linpack (HPL) Benchmark on Whale
table. Note that such a code would be difficult to write using a 2-sided send/receive model, since the target processor is not involved at all in the update. In order to prevent simultaneous updates to the same entry, some form of atomicity must be employed.

In the original coarray code, based on Fortran 2008, this was achieved using locks. This benchmark, therefore, is a good stress test for our implementation of coarray locks. For each new entry an image wishes to insert, it will calculate the image and position that the entry maps to, and then it will acquire a lock for that image’s local hash table. Next, it will read the current entry at the specified position. If the position is unfilled, it will write the new entry into it, set a counter for that position (also at the target image) to 1, and then release the lock. If the position already contains the same entry, it will increment the counter for that position, indicating that the entry has been inserted multiple times into the table, and then release the lock. Finally, if the position is occupied with a different entry, then the image will release the lock and attempt to insert the entry into the next position in the distributed hash table (which may be at the same image, or another image).

We then rewrote the code to make use of extended atomic operation features. Specifically, we used the \texttt{atomic}\_\texttt{cas} and \texttt{atomic}\_\texttt{add} intrinsic subroutines. An atomic compare-and-swap (CAS) operation is used to check if the position at the target image is empty, and if so it will write the entry into it. The return value of the CAS operation will indicate whether the update was successful or if the position already contained the same entry. In either of these cases, the image will atomically
increment the corresponding counter for the position. Otherwise, the image will repeat the process for the next position in the hash table. Therefore, using these new atomic operations, we can reduce the total number of remote accesses required for an insertion that does not result in a collision from four to two.

We show our evaluation of this benchmark on Titan. As can be seen, our locks implementation delivers similar performance compared to Cray’s Fortran compiler. When we use the atomics extensions proposed for TS 18508 but not supported by Cray at this time, there is a significant performance improvement.

Figure 7.2: Distributed hash table benchmark on Titan, using 16 images per node.
7.3 NAS Parallel Benchmarks

The NAS Parallel Benchmarks (NPB) are a set of benchmarks intended for performance evaluation of large-scale parallel systems and are maintained and developed by the NASA Advanced Supercomputing (NAS) Division at NASA Ames Research Center. The suite consists of several benchmarks, primarily based on computational aerophysics, but are intended to be representative of a large class of scientific applications. There are serial versions of the benchmarks, along with versions written in MPI, OpenMP, and HPF. Other development efforts have created versions in other parallel programming models, including OpenSHMEM, UPC, and a non-standardized variant of Co-Array Fortran. For our evaluation, we ported all the Fortran NPBs to use Fortran coarrays, adhering to the Fortran 2008 specification. The original 8 benchmarks from the suite are comprised of five kernel benchmarks and three simulated computational fluid dynamics (CFD) applications.

The kernel benchmarks were:

- **EP**: This is an “embarrassingly parallel” benchmark which requires no interprocessor communication apart from coordination on generating pseudo-random numbers at the beginning, and a collective operation at the end to collect results.

- **MG**: This benchmark performs a simplifid multi-grid calculation and is characterized by structured communication, stressing the system’s capability for short and long distance data transfer.
• **CG:** This benchmark uses the *conjugate gradient* method to approximate the eigenvalue of a sparse, symmetric positive definite matrix, and makes use of unstructured matrix vector multiplication.

• **FT:** This benchmark uses *fast Fourier transforms* (FFTs) to solve the 3-D Poisson partial differential equation, and is considered a good evaluation for communication performance.

• **IS:** This benchmark performs a large *integer sort*, and tests performance of integer computation as well as communication.

The simulated CFD applications were:

• **LU:** This benchmark uses a block *lower-triangular, upper-triangular* factorization to solve a discretization of the 3-D compressible Navier-Stokes equations.

• **SP:** This benchmark uses Gaussian elimination to solve the 3-D compressible Navier-Stokes equations using three independent systems of linear equations which are each *scalar pentadiagonal*.

• **BT:** This benchmark uses Gaussian elimination to solve the 3-D compressible Navier-Stokes equations using three independent systems of linear equations which are each *block-tridiagonal* with 5x5 blocks.

In the official NPB distribution, there are OpenMP and MPI versions of these benchmarks, and all of them except IS were written using Fortran 77. We ported these seven benchmarks to use Fortran 2008 coarrays to evaluate the performance
Figure 7.3: CAF (Fortran 2008) versions of the NAS Parallel Benchmarks on Titan, using 16 images per node. All benchmarks were run using class size B.
of our CAF implementation. We present the results for six of these benchmarks (EP, being an uninteresting case for evaluating our implementation, is not shown) in Figure 7.3 when running on Titan. As can be seen, the performance of these benchmarks using our compiler was nearly identical to the performance achieved when using the Cray Fortran compiler.

Figure 7.4: CG Benchmark on Whale, using 8 images per node

**Application of Teams and Collectives for CG** To demonstrate the benefits of the teams and collectives features (discussed in Sec. 4.6) which are expected to be added into a future version of Fortran, we made use of them for the CG benchmark. We grouped the images into *row teams*, and during the execution of the conjugate gradient method we performed the sum reductions with respect to these teams,
using our optimized reduction implementation described in Sec. 6.7. In Figure 7.4, we show the benefits achieved with this new implementation on Whale, compared to our original Fortran 2008 version of the benchmark and an MPI version of the benchmark using Open MPI 1.8.3.

7.4 Reverse Time Migration

We describe now how we applied Coarray Fortran to implement an algorithm of relevance to the Oil and Gas industry. The code that was developed performs Reverse Time Migration (RTM) [1]. This method is widely used in subsurface exploration via seismic imaging. A source emits a wave pulse, which reflects off of subsurface structures and is recorded by an array of receivers. RTM transforms the recorded data into an image of the reflector locations. It is suitable for parallelization by domain decomposition, and is often executed on distributed memory systems due to the large volumes of data involved.

RTM uses the finite difference method to numerically solve a wave equation, propagating waves on a discretized domain. It consists of two stages. The first is referred to as the forward stage as it propagates an approximation of the source wave forward in time. In the second, backward stage, the recorded data is propagated backward in time. The RTM algorithm assumes that the forward propagated source wavefield and backward propagated data wavefield overlap in space and time at the location of reflectors. This imaging condition is evaluated during the backward pass. As it involves the wavefield from the forward pass, a technique must be employed to
allow this wavefield to be recovered at each step during the backward stage. This could be achieved by saving the entire forward wavefield at each time step, and then reloading the appropriate data when it is needed. More sophisticated methods, such as only saving the boundaries of the forward wavefield \[97\], reduce the memory requirement at the expense of additional computations by recomputing the forward wavefield during the backward pass.

Typical seismic experiments involve several thousand shots. Each shot consists of a source wave pulse, usually at a distinct location, and the associated receiver data. A single shot in a production survey typically contains several gigabytes of data. The total acquisition data size is usually in the terabyte range. For example, one dataset acquired from 30 blocks (3 × 3 square miles per block) consists of 82K shots, 460M traces with 3600 samples per trace with total data size of about 6.6 TB. Shots are processed individually, but potentially in parallel, each requiring tens of thousands of time steps of a 3D eighth-order finite difference propagator on a 3D domain containing billions of cells. Depending on the storage strategy used, significant volumes of data may need to be saved during each time step of the forward pass and recovered during the backward pass to enable the application of the imaging condition. RTM processing is therefore among the most data intensive applications used in industry.

The simplest implementations of the RTM algorithm make the assumption that the Earth is isotropic, this means that wave propagation speed is not dependent on angle. In reality this is often not a valid assumption as rocks such as shale, a common material at the depths of interest to hydrocarbon exploration, can exhibit strong
anisotropy. Since hydrocarbon exploration frequently occurs in regions covered by sedimentary rocks, which were laid down in layers, a more plausible assumption is that of transverse isotropy, where waves can propagate at different speeds parallel and perpendicular to bedding planes. To account for buckling, uplift, and other processes which can change the orientation of the rock layers, tilted transverse isotropy (TTI) may be assumed. The TTI implementation of RTM is considerably more compute and communication intensive than the isotropic version.

7.4.1 Implementation

A full RTM code was developed using Coarray Fortran, capable of processing production data. Both isotropic and TTI propagators were included. A section of the image produced by the code of the 2004 BP model, is displayed in Figure 7.5. This is a synthetic model designed to be representative of challenging sub-salt environments currently being encountered in exploration of the Gulf of Mexico and off-shore West Africa. It consists of more than 1000 source locations spread over 67 km, on a grid discretized into 12.5m × 6.25m cells.

It was possible to design the code in such a way that only the communication modules were specific to a coarray implementation of the algorithm. This permitted an equivalent MPI version to also be created for performance comparison by merely writing MPI communication modules.

At each time step in the code, wavefield propagation is carried out in parallel by all images, utilizing halo exchanges to communicate face (and, in the case of
Figure 7.5: A section of the 2004 BP model image generated by RTM using CAF.

TTI model, edge) data amongst neighbors. For the forward stage of the isotropic model, this entails a total of six halo exchanges for one wavefield, while for the TTI model there are eighteen halo exchanges of two wavefields. In the backward stage, if boundary swapping is being employed, then the total number of halo exchanges for each processor is doubled for the isotropic and TTI models. In both the CAF and MPI implementations, the halos from all of the wavefields to be transmitted to a particular processor were packed into a single buffer and then unpacked at the destination. This reduces the number of messages ultimately sent over the network, potentially improving communication performance. A further advantage of this approach is that the wavefields themselves do not need to be declared as coarrays in the CAF implementation, only the buffers, simplifying the creation of an MPI version free of coarrays.
As CAF currently lacks parallel I/O, for this RTM implementation we utilized conventional sequential I/O and performed a post-processing step to merge the output. Two different inputs are required for RTM. One is the earth model. This consists of a model of the wave speed in the region of the earth being imaged, and may also contain models of other parameters. Each of these parameter files is a single large file, containing a 3D block of floating point numbers. Each process only needs access to the part of the model that corresponds to the part of the domain that it has been assigned. The models only need to be read once by each process, and are then stored in memory for the duration of the execution. This was implemented as each process calling a function to read the relevant parts of the files by looping over two dimensions, performing contiguous reads in the third, fast dimension. Additional input that is required includes the data recorded by receivers. These are the data containing recordings of waves that reflected in the subsurface and returned to the surface. RTM uses these recordings to try to construct an image of where the reflections took place. Processes only need access to data recorded by receivers located within their local domain. Typically a single file contains the data needed by all of the processes. This was again implemented by having each process repeatedly call a function that performs a contiguous read. This also only needs to be done once by each process. The time required for reading input data is typically quite small compared to the overall runtime, as once it is loaded from disk into memory it is used in a large number of computations.

The output from RTM is the image of the reflectors in the earth. Each process produces an image for its part of the domain. In this implementation each process
writes its image as a separate file, with a header file describing which part of the domain the data corresponds to. The RTM code is executed many times, once for each shot. A large number of output files are therefore created. Post-processing is performed to load all of these, and combine them into a single image file. In a production environment this may require loading hundreds of thousands of files, potentially straining the filesystem. Parallel I/O would have been advantageous for this situation, as it would enable the files from all of the processes working on a single shot to be joined before being written to disk, so the total number of files would be divided by the number of processes working on each shot.

7.4.2 Performance

![Figure 7.6: Neighbor Communication Time for MPI, CAF GET, and CAF PUT (strided and buffered)](image)

In Figure 7.6 we depict communication time between neighbors. We compare communication using MPI send and receive (where non-contiguous data are packed and unpacked as necessary into contiguous buffers), and using 1-sided get and put.
via co-indexed accesses. For the latter, we considered (1) the “buffered” approach where the program packs all the data to be communicated into a contiguous buffer, and (2) the “strided” approach where the non-contiguous data to be communicated are represented directly using Fortran array-section notation and no additional buffer copy is used by the programmer (or generated by the compiler). All the CAF variants for communication worked well for smaller message sizes (less than 400 elements per side). For larger message sizes, we see the CAF `get` variants, and particularly the buffered version, delivering performance similar to or exceeding the MPI communication. Similar results reflecting the advantage CAF has for smaller message sizes over MPI have been reported for other applications [96] [47] [10]. The performance benefit shown here in using contiguous buffers for CAF communication motivated our decision to employ such buffers for implementing halo exchanges in the RTM code.

The plots in Figure 7.7 show the runtime per time step of the CAF and MPI versions of the code, for (a) isotropic and (b) TTI, as the number of processors was increased with a fixed total problem size.
versions during the forward stage of RTM. For the MPI version we used the Intel Fortran compiler, and for the CAF version we used the OpenUH compiler with comparable optimization options. A fixed domain size consisting of $1024 \times 768 \times 512$ cells was used, on a cluster with Intel Nehalem CPUs and QDR Infiniband interconnect. These measurements appear to show good performance of the CAF code: it was up to 32% faster for the isotropic case, while approximately equal to the MPI code for TTI. Recall that for each time step, each image performs halo exchanges of its wavefield data with its neighbors. For the forward stage, the TTI model requires eighteen halo exchanges for two wavefields, while the isotropic model requires only six halo exchanges for one wavefield. As these performance results was obtained prior to the availability of directives for hiding communication latency, the benefits observed for CAF in the isotropic case are offset by the significantly higher communication latency costs of the halo exchanges in the TTI case. Nevertheless, we believe this demonstrates that a production-level industrial code can be developed using CAF, with performance similar to that of an MPI implementation being achievable while retaining the benefits of a compact, clean syntax that is more accessible to the non-expert programmer.

7.5 Summary

In this chapter, we evaluated the use of Coarray Fortran for a number of benchmarks and application codes. Our evaluation consisted of performance comparisons to equivalent MPI and CAF 2.0 versions of the code, as well as comparisons to programs
compiled with the Cray Fortran compiler which has supported CAF for more than a decade. The results of our evaluation showed the effectiveness of the new teams, collectives, and atomics support in our implementation. Our application study using a reverse time migration code from Total provided confirmation that CAF can be an effective programming interface for real codes used in the Oil and Gas industry. However, there is scope for additional improvement in the language features. In particular, while CAF extends Fortran for parallel execution, it does not yet provide an effective means to manage parallel I/O. In the next chapter, we discuss such an extension.
Chapter 8

A Parallel I/O Extension for Coarray Programs

In this chapter, we describe extensions we have proposed for supporting parallel I/O in the Coarray Fortran programming model. Although CAF currently lacks support for parallel I/O, for the RTM implementation described in the preceding chapter we were able to utilize traditional sequential I/O. Each processor wrote its output to a unique file and post-processing was performed to obtain the final resulting image by combining these outputs. To load the relevant portion of the input data on each processor, strided reading was employed. For the number of processors tested, combined with the use of a Lustre parallel file system, this was not found to create a bottleneck. Thus, it was possible to develop a parallel RTM code using CAF, despite the latter’s lack of support for parallel I/O. It was suggested, however, that
the file system may be overloaded by this approach on larger systems used in production environments. More generally, we believe that in the petascale/exascale era, I/O will increasingly become the primary performance bottleneck for data-intensive applications. Therefore, a sound parallel I/O mechanism is a crucial aspect for future parallel programming models and implementations. The work discussed in this chapter was described in our publication [37].

8.1 Design Issues for Parallel I/O

We describe several design points for parallel I/O that we consider especially relevant.

1. **File Handlers: Individual versus Shared** While working on a file, multiple images may use separate file handlers, one per image, to keep track of individual progress of read/write operations within a file. In this case, each image remains unaware of the progress of other images. However, the drawback of this approach is that additional communication needs to be performed to obtain the position of the shared file handler within the file. An alternative approach can be to read/write the same file using a handler which is shared by all the images. In this case, since the progress of the read/write operations is indicated by a shared resource, each image has to communicate with other images to determine the position of the handler. In a two-way communication model, such small communications can prove to be extremely inefficient. A more favorable approach would be to use direct-access file access operations which would enable every image to explicitly specify an offset within the file by
specifying a record number. As a result, there does not arise a need to include shared file pointer operations for parallel I/O for CAF.

2. **Concurrent I/O: Independent versus Collective** One of the primary reasons for performance degradation of file I/O operations is the high overhead caused due to the seek latency involved with multiple reads/writes. This worsens when the read/write requests arrive at the file system nodes in a discrete unordered sequence. Since such requests come from images that are unaware of the progress of other images, the disk header might have to repeatedly move back and forth through the same sectors of the disk to service the requests. With increase in the number of images, the performance takes a clear toll. An alternative approach is for all the images to aggregate their requests such that the range of the new request spans over contiguous sections on the disk, thereby dramatically increasing the performance. This can be achieved by collective operations and has been explored heavily over the past few years. A consideration while using such a collective operation is that faster images would wait to synchronize with the slower images so that the request for file operations can be collectively put forth to the file system. However, in general the benefits of aggregated requests have been shown to supercede the costs of this synchronization for such collective operations by MPI-I/O [98], OpenMP-I/O [70], and UPC-I/O [38].

3. **Synchronous versus Asynchronous I/O** To avoid idle CPU cycles during the servicing of data transfer requests, images could benefit if such requests could proceed in an asynchronous manner. In a multi-processing environment,
this leads to an opportunity of overlapping the I/O operations with useful computation and communication among the images.

4. **File Consistency Among Processors** In data-intensive HPC applications, performing repeated disk accesses to access consecutive sections of the disk will lead to a severe degradation of performance. There have been efforts to exploit the principle of locality by caching sections of the file which are frequently accessed by different images and to service the read/write operations more efficiently at the risk of relaxed consistency between the cache buffers and the physical non-volatile disks. This means that images might benefit from having a local copy of a section of the file and translate the read/write operations in the form of quick memory transfers. This involves additional pre- and post-processing at regular intervals to provide a consistent view of the shared file.

WG5, the working group responsible for the development of Fortran standards, has not yet settled upon a solution for parallel I/O to be part of the Fortran standard. One approach would be to follow a somewhat modest extension for parallel I/O that was originally part of Co-Array Fortran [80]. Here, the `OPEN` statement is extended to allow multiple images to collectively open a single file for shared access. Files are represented as a sequence of records of some fixed length. Once the file is opened, each image may independently read or write to it, one record at a time. It is the programmer’s responsibility to coordinate these accesses to the files with the use of appropriate synchronization statements. Some details on how precisely the extension should be incorporated into the standard are still unresolved and under discussion, and hence it has not been slated for inclusion in the Technical Specification for
extended features. We point out some potential drawbacks from our perspective:

- Files are organized as a 1-dimensional sequence of records. Higher-level representations of data (e.g. multi-dimensional view of records) would be easier to deal with for programmers and potentially enable more optimized I/O accesses.

- Records may be read or written by an image only one at a time. This hinders the ability of the programmer (and, to a degree, the implementation) to aggregate reads and writes for amortizing the costs of I/O accesses.

- There is no mechanism for collective accesses to shared files. Collective I/O operations are extremely important, particularly as we scale up the number of images that may be doing I/O. New collective variants for the \texttt{READ} and \texttt{WRITE} statements will allow the implementation to funnel I/O accesses through dedicated nodes to reduce excessive traffic to secondary storage.

8.2 Incorporating Parallel I/O in Coarray Fortran

Before a detailed description of our proposed extensions, let us consider the file access pattern illustrated in Figure 8.1 in which multiple images simultaneously access the non-contiguous, shaded portions of the file. The code snippet in Figure 8.2 uses MPI I/O to enable each image to independently access this section of the input file. For comparison, we show code snippet in Figure 8.3 for reading the same file without changing the logical data layout in Figure 8.1 using our CAF extension for parallel I/O.
It can be observed that the use of the extensions improves usability by dramatically reducing the size of the code.

### 8.2.1 OPEN Statement

The OPEN statement is used to open an external file by multiple images. Fortran files which are used for parallel I/O should contain only unformatted binary data. Fortran 2008 forbids the simultaneous opening of files by more than one image. However, to add support for parallel I/O, this restriction can be lifted with the \texttt{TEAM=\textasciitilde yes} specifier, which indicates that all images in the currently executing team will collectively open the file for access. Table 8.1 lists the different types of control specifiers within the OPEN statement which are relevant to parallel I/O support.

Some of the key specifiers which have been extended or added have been discussed below:
INCLUDE mpif.h.

MPI_INT blklens(3)
MPI_TYPE old_types(3)
MPI_INT indices(3)
MPI_INT ierr

....

! MPI_TYPE_VECTOR(count , blklen, str, old_type, new_type, err)
call MPI_TYPE_VECTOR(2, 2, 4, MPI_REAL8, type1, ...)
! MPI_TYPE_VECTOR(count , blklen, str, old_type, new_type, err)
call MPI_TYPE_VECTOR(3, 1, 2, type1, type2, ...)
! MPI_TYPE_VECTOR(count , blklen, str, old_type, new_type, err)
call MPI_TYPE_VECTOR(3, 1, 2, type1, type2, ...)

count = 3
blklens = (/1, 1, 1/)
indices = (/0, 48, 384/) ! in bytes
old_types = (/MPI_LB, type2, MPI_UB)

call MPI_TYPE_STRUCT(count, blklens, indices, old_types, &
                     newtype, ...)
call MPI_TYPE_COMMIT(newtype)

! setting the view of the file for the image
call MPI_FILE_OPEN(MPI_COMM_WORLD, "/file_path/...", &
                  MPI_MODE_RD_ONLY, ..., fhdl, ...)
call MPI_FILE_SET_VIEW(fhdl, 0, MPI_REAL8, newtype, &
                       "native", ...);

! reading begins
call MPI_FILE_READ(fh, buff, 1,newtype,...)
....
call MPI_FILE_FREE(newtype)
call MPI_FILE_CLOSE(fhdl)

Figure 8.2: Example using MPI IO

OPEN(UNIT=10, TEAM='yes', ACCESS='DIRECT', ACTION='READ', &
     FILE='/file_path/...', RECL=16, NDIMS=2 DIMS=(\3\))
READ(UNIT=10, REC_LB=(/1,2/), REC_UB=(/3,6/), &
     REC_STR=(/2,2/)) buff(:)
CLOSE(10)

Figure 8.3: Example using the proposed Fortan parallel I/O
Table 8.1: List of the primary specifiers related to the *modified* OPEN statement. The list includes existing specifiers as well as the newly proposed ones.

<table>
<thead>
<tr>
<th>Specifier</th>
<th>Same value across all images (Yes/No)</th>
</tr>
</thead>
<tbody>
<tr>
<td>UNIT1</td>
<td>YES</td>
</tr>
<tr>
<td>FILE1</td>
<td>YES</td>
</tr>
<tr>
<td>IOSTAT1</td>
<td>NO</td>
</tr>
<tr>
<td>TEAM2</td>
<td>YES</td>
</tr>
<tr>
<td>NDIMS2</td>
<td>YES</td>
</tr>
<tr>
<td>DIMS2</td>
<td>YES</td>
</tr>
<tr>
<td>OFFSET2</td>
<td>YES</td>
</tr>
<tr>
<td>ACCESS1</td>
<td>YES</td>
</tr>
<tr>
<td>ACTION1</td>
<td>YES</td>
</tr>
</tbody>
</table>

1 This specifier already exists in the current Fortran 2008 standard. However, the restrictions applicable to this specifier has been modified as part of this proposal.

2 This specifier does not exist in the Fortran 2008 standard and has been proposed as an extension.

- **UNIT**: While **OPEN**ing a file collectively, all images must use the same unit number in the statement.

- **FILE**: This includes the full path of the file in the file system. While using a remote file system in a clustered environment, the programmer must ensure that the path-name provided should be the same and should evaluate to the same physical file on all the nodes on which the images execute.

- **IOSTAT**: If an **OPEN** statement (with a valid UNIT number) is executed and if the associated file has already been opened by the same executing image, the *stat-variable* becomes defined with the processor-dependent positive integer value of the constant `STAT_FILE_OPENED_BEFORE` from the intrinsic module `ISO_FORTRAN_ENV`. Also, if an OPEN statement is executed by an image when
one or more images in the same TEAM (described below) has already terminated, the stat-variable becomes defined with the processor-dependent positive integer value of the constant STAT_STOPPED_IMAGE.

- TEAM: The TEAM=’yes’ specifier is used to indicate that a file is to be opened in a parallel I/O context (i.e. shared access between multiple images in the current team). Using team objects for parallel I/O allows the user to limit the I/O operations to a subset of images. In the absence of this specifier, the image is connected to the file with exclusive access.

- NDIMS: This specifier accepts a positive integer which specifies the number of dimensions of the representation of the data in the file.

- DIMS: This provides the programmer with the flexibility of representing sequential records in the file in the form of multi-dimensional arrays. The array-expr passed to the specifier contains a list of extents along each dimension except the last (which can be calculated internally using the record length and the total size of the file).

- OFFSET: This specifier accepts a positive integer which defines the offset within the file (in file storage units, typically bytes) at which the multi-dimensional file data starts. It may be used to reserve space in the file for header information.

- ACCESS: Out of the two access modes in Fortran - direct and sequential, performing a shared file access in parallel should be limited to direct-access mode. The WG5 standards committee suggests that sequential access mode should not be supported for parallel I/O.
• **ACTION**: All images must pass the same value (*read*, *write*, or *readwrite*) for the specifier.

Data-intensive scientific applications which require representations with multiple dimensional-views of the same data-set can represent the layout while using the `OPEN` statement. For example, Figure [8.4] represents how a series of flat 24 records may be viewed as 8x3, or a 4x2x3 array of records.

![Diagram](image)

*Figure 8.4: The flexibility of the same dataset being represented using different dimension layout allows the adaptability of algorithms semantics to file contents*
8.2.2 READ/WRITE Statements

One of the main concerns for programming models that support parallel I/O for data intensive applications is the scalability of I/O requests and the actual transfer of the data among the multiple nodes and the I/O servers. To reduce excess network traffic in large scale systems, the runtime could take advantage of I/O requests from different images targeting adjacent sections in the shared file. This allows opportunities for combining the large number of requests to a few. This helps to avoid bottlenecks at the I/O servers, reducing average latency among the I/O servers and the nodes. To meet this end, we extend the READ and WRITE statements to enable collective accesses to different (though not necessarily disjoint) sections of the multi-dimensional data, using the optional COLLECTIVE='yes' specifier.

8.2.3 Asynchronous Operations

I/O intensive applications alternate between computation and I/O phases. As a result, it is not essential that all the images have the same view of the file during the complete execution of the application. To support this, the ASYNCHRONOUS specifier as included by the Fortran 2003 standard can be extended for parallel I/O. The value YES needs to be passed to this specifier while connecting to a file (using OPEN). Additionally, the programmer has the flexibility to choose which I/O operation on an already connected file needs to be made asynchronous. This is achieved by explicitly specifying ASYNCHRONOUS='yes' for every I/O statement on that file. Images may write to disjoint sections of the file without its changes being immediately visible to
Table 8.2: List of the primary specifiers related to the *modified* READ/WRITE statement. The list includes existing specifiers as well as the newly proposed ones.

<table>
<thead>
<tr>
<th>I/O statement specifier</th>
<th>(if COLLECTIVE=YES) Same value across all images (Yes/No)</th>
</tr>
</thead>
<tbody>
<tr>
<td>UNIT(^1)</td>
<td>YES</td>
</tr>
<tr>
<td>ASYNCHRONOUS(^1)</td>
<td>YES</td>
</tr>
<tr>
<td>IOSTAT(^1)</td>
<td>NO</td>
</tr>
<tr>
<td>REC_LB(^2)</td>
<td>NO</td>
</tr>
<tr>
<td>REC_UB(^2)</td>
<td>NO</td>
</tr>
<tr>
<td>REC_STR(^2)</td>
<td>NO</td>
</tr>
<tr>
<td>REC_STR(^2)</td>
<td>NO</td>
</tr>
<tr>
<td>COLLECTIVE(^2)</td>
<td>YES</td>
</tr>
</tbody>
</table>

1 This specifier already exists in the current Fortran 2008 standard. However, the restrictions applicable to this specifier has been modified as part of this project.

2 This specifier does not exist in the Fortran 2008 standard and has been proposed as an extension.

other images sharing the same file. Such semantics enables the runtime to delay the completion of the effects of read/write operations until a definitive point is hit. This point in the program can be the existing **FLUSH** statement.

Table *8.2* enlists the different types of control specifiers within the READ / WRITE statements which are relevant to parallel I/O support.
8.3 Access Patterns Using the Triplet Representation

As described before, the DIMS specifier enables the programmer to set the layouts of the records of a file as a multi-dimensional array. Once a file is connected, the access pattern is represented by specifying a triplet \(<\text{lower bound}, \text{upper bounds}, \text{and strides}\>\) along each dimension. These can be defined within the I/O \texttt{READ} and \texttt{WRITE} statements using the \texttt{REC_LB}, \texttt{REC UB}, and \texttt{REC STR} specifiers.

(a) Different triplet notations can target the same file sections for different data layouts

(b) File sections that cannot be accessed using a single triplet notation can be rearranged using a different layout to allow ease of access

Figure 8.5: Use of triplet notations to access different data patterns in a file

Large-scale scientific applications deal with complex data patterns based on a multi-dimensional view of the algorithmic data structures. Consider Figure 8.5a, where the highlighted sections of the arrays are accessed by an image. In this, the data layouts represent the usefulness of the triplet representation when compared to the flat sequential records view of the current Fortran standard or the streaming-bytes view of POSIX compliant C programs.
While most access patterns can be easily represented using a single triplet notation across each dimension, there exist corner cases, where multiple disjoint accesses become necessary to completely represent the complex patterns using a standard triplet notation. For example, consider Figure 8.6. Here, the elements accessed (3, 4, 7, and 8) do not have a uniform stride between them when the file is represented as a 1x8 array. Using this layout, an application programmer might have to resort to performing multiple I/O operations (two steps in this case), to completely read the data. Such multiple I/O requests for small chunks of data clearly would take a toll on the overall performance. Instead, if the programmer were to choose to represent the layout of all the records as a 2x4 array (Figure 8.5b), the access pattern can be easily represented using a single triplet representation. Figure 8.7 illustrates this point further.

Figure 8.6: Multiple steps used to access a pattern from a file represented as a one-dimensional array.
8.4 Summary

In this chapter, have defined an extension to Fortran for enabling parallel I/O. Our initial evaluations indicated that additional support for handling large data sets would be extremely beneficial if coarrays are to be used on very large scale computations. As the size of such data sets, and the computational capacity of platforms, continues to grow, there is an increasing gap between this and the rate at which data is input and output from and to disk. In order to accommodate it, we believe that there is a need for parallel I/O features in Fortran which complement the existing parallel processing features provided by the coarray model.
Chapter 9

Conclusion

In this dissertation, we have described several methods that were developed for efficiently compiling and executing parallel Coarray Fortran programs, based on the Fortran standard. The contributions of this dissertation are summarized as follows:

- We created a compiler and runtime implementation of these new language features, the first open-source implementation to fully support the coarray model described in Fortran 2008 [36,37]. We developed several techniques, including designing a symmetric memory pool for efficient allocation and deallocation of remotely accessible data, efficient support for code generation and runtime support for non-contiguous transfer, compiler and runtime support for overlapping communication and computation, efficient synchronization algorithms, and several optimizations for implementing collective operations using 1-sided communication.
• We developed several benchmark and application codes that use these language features which demonstrate the effectiveness of Coarray Fortran as a parallel programming model.

• We developed the first implementation of the anticipated teams feature expected to be added to Fortran 2015 [60], in addition to implementing support for collectives, atomics, and event synchronization. We demonstrated the effectiveness of these new features in parallelizing a distributed hash table code, the High Performance Linpack benchmark, as well as the CG benchmark from the NAS Parallel Benchmark suite.

• We proposed an extension to Fortran for seamlessly incorporating parallel I/O into coarray programs [37].

9.1 Future Work

During the course of this work, we developed a robust and performant implementation for Fortran coarrays within the OpenUH compiler. There are a number of research directions we intend to explore, building on the work completed.

We developed profiling support for CAF programs, allowing the detection of performance overheads relating to communication, synchronization, and memory allocation of coarrays and communication buffers. We also developed compiler directives for instructing the compiler how to generate asynchronous calls for overlapping communication with computation. We intend to incorporate this profiling system into
a feedback-based optimization framework. In this framework, feedback information can be used to guide the insertion of compiler directives, and we will explore additional directives for generating non-blocking synchronization and collectives calls as well. We also intend to explore runtime optimizations based on node and network topology. We have implemented early work in this area, expanding our algorithms for barrier, broadcast, and reductions to exploit shared memory versus inter-node communication. We plan to generalize this multi-level approach and adapt our algorithms for collective communication and synchronization for complex memory and node layout configurations.
Bibliography


153


