IMPLEMENTATION AND EVALUATION OF ADDITIONAL PARALLEL FEATURES IN COARRAY FORTRAN

A Thesis

Presented to

the Faculty of the Department of Computer Science

University of Houston

In Partial Fulfillment

of the Requirements for the Degree

Master of Science

By

Shiyao Ge

May 2016

IMPLEMENTATION AND EVALUATION OF ADDITIONAL PARALLEL FEATURES IN COARRAY FORTRAN

Shiyao Ge

APPROVED:

Barbara Chapman, Chairman
Dept. of Computer Science

Edgar Gabriel
Dept. of Computer Science

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

Dean, College of Natural Sciences and Mathematics

Acknowledgements

I would like to express my deep gratitude to my advisor, Prof, Barbara Chapman, for her guidance and support in my work and her patience during the time being her student. I would also thank my other committee members, Prof. Edgar Gabriel and Dr. Mikhail Sekachev for taking time to review my work and give valuable feedback.

I would give special thanks to my mentor Deepak Eachempati and Dounia Khaldi for their valuable and constructive suggestions during the design and development of this research work. A huge thank to Tony Curtis, who gives many help in my early stages in HPCTools group, helping with techinical issues and reviewing my work.

I would also like to extend my thanks to TOTAL for funding our work. Thanks to Mikhail Sekachev, Henri Calandra and Maxime Hugues for allowing me to access to their machines to verify our work and giving me feedback. Thanks Texas Advanced Computing Center (TACC) for providing computing resources.

It is an honor for me to work within the HPCTools research group. I would thank Siddhartha Jana for his countless valuable feedback and discussion. Also I would thank Pengfei Hao for helping me resolve techinical problem and giving wise advices.

Finally, I am very grateful to my parents for their care and support all the time.

IMPLEMENTATION AND EVALUATION OF ADDITIONAL PARALLEL FEATURES IN COARRAY FORTRAN

An Abstract of a Thesis

Presented to

the Faculty of the Department of Computer Science

University of Houston

In Partial Fulfillment

of the Requirements for the Degree

Master of Science

By

Shiyao Ge

May 2016

Abstract

The Fortran 2008 language standard added a feature called "coarrays" to allow parallel programming in Fortran with only minimal changes to existing sequential Fortran programs.

Coarrays turn Fortran into a Partitioned Global Address Space (PGAS) language, following the Single Program, Multiple Data (SPMD) model.

The next revision of the Fortran standard is expected to introduce some more sophisticated coarrays language features. One feature is the "team"; a way of grouping components (images) of parallel Fortran programs. Teams can, for example, be allocated different sub-tasks.

Proposed team support in the standard includes statements for forming image teams, reassigning membership of teams, and statements for performing communication and synchronization with respect to image teams. These features are collected and discussed in the Fortran Technical Specification Draft.

In this thesis, we will present implementation and evaluation of some of these new features. The open-source compiler, OpenUH, developed by this research group is extended to implement support for *team* and *collective*.

We discuss two optimizations we have applied in order to reduce network communication and local memory footprint in the compiler's Coarrays runtime.

Experimental results using several micro-benchmarks, one benchmark from the NAS Parallel Benchmark suite and High Performance Linpack suite show that new features make the program logic more concise, while achieving good performance.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

In past few decades, people defined several parallel programming models to help abstract the parallel computing system interfaces. In recent years, a programming model referred to as *Partitioned Global Address Space* (PGAS) has engaged much attention as a highly scalable approach for programming large-scale parallel systems. The PGAS programming model is characterized by a logically partitioned global memory space, where partitions have affinity with the processes/threads executing the program. This property allows PGAS-based applications to specify an explicit data decomposition that reduces the number of remote accesses with longer latencies. This programming model marries the performance and data locality (partitioning) features of distributed memory model with the programmability and data referencing simplicity of a shared-memory (global address space) model.

Several languages and libraries follow the PGAS programming model. OpenSHMEM[10] and Global Arrays[41] are examples of library-based PGAS implementation, while

Unified Parallel C (UPC)[8], Titanium[26], X10[12], Chapel[9] and Coarray Fortran (CAF)[42] are examples of PGAS-based languages. Compared with the library-based implementation, which assumes the programmer will use the library calls to implement the correct semantics following the programming specification, the language-based implementations aim to simplify the burden of writing applications that efficiently utilize these features and achieve performance goals for the non-expert programmers. However, the adoption of language-base implementation is much slower than the libraries-based implementation.

1.1 Motivation

For a long time, Fortran is one of the dominant languages in the HPC area. According to The National Energy Research Scientific Computing Center (NERSC), which is the primary scientific computing facility for the Office of Science in the U.S. Department of Energy, over 1/2 the hours on their systems are used by Fortran codes. Fortran 2008 has introduced a set of language features that support PGAS programming model, often referred to as *Coarray Fortran* or *Fortran Coarrays* (CAF). Currently, only a few compilers embrace these new features into their latest release. Although Fortran 2008 has included a set of simple but efficient PGAS features, users demand for advanced Coarray features to express more complicated parallelism in their application. Based on that, the Fortran work group has identified a set of advanced features and plans to introduce them into next language standard[14].

The HPCTools Group in the University of Houston has developed a functional

compiler and runtime implementation to support the Coarray features in Fortran 2008[17]. While processes in global environment always keep its one-to-one mapping of the co-subscription, the *team*, which is a nested construct to represent subset of processes, brings more complexity. Performing communication and synchronization should be respect to *team*s, rather than in the flat global environment. The thesis describes the implementation of the theses features. It also presents two optimizations we applied to reduce the network communication and local memory usage in runtime. We also give a discussion of in which case the optimizations can give better performance.

1.2 Contributions

The contributions of this work include:

- a description of an early implementation of additional parallel processing features, including teams, collectives, and barrier operations, which are complementary to the existing Fortran coarrays model and are being developed for incorporation into the next revision of the Fortran standard
- optimization techniques in the runtime, including locality-aware optimization and distributed mapping table
- evaluation of enhanced coarray features using benchmarks to assess the usefulness of team-based synchronizations and collectives.

1.3 Thesis Organization

This thesis is organized as follows: Chapter 2 will give a brief introduction of background information for this thesis. We discuss the concept of PGAS model. Then we have a tour of Fortran history, so that we can understand what we can do with this specific language in HPC area and why they include the PGAS model into their latest language standard. Also section2.4 will give a short introduction to Coarrays Fortran. Readers can familiar themselves with these syntax. This thesis does not cover the implementation of these features since they are established before this thesis work. We then present a discussion of task decomposition cases in parallel program and the progress of other PGAS libraries and languages to express such decompositions. Finally, section 2.6 will give a brief survey of two on-going Coarray Fortran projects.

Chapter 3 reviews existing compiler and runtime infrastructure that we used for this thesis work. Chapter 4 describes in detail the design of *team* construct, including the `team_type` variable, the functions supporting *team* and the memory management in our runtime. In the following section, we will discuss the effort we made in our runtime to optimize the *team* construct in terms of latency and memory utilization.

Chapter 5 gives an evaluation of our implementation using micro-benchmarks and benchmark from NAS Parallel Benchmarks. Finally, Chapter 6 summarizes this thesis and includes a description of future work.

Chapter 2

Background

In this chapter, we will get to know about the programming model that we follow and language syntax used in this paper. We start with the brief introduction of PGAS programming model then we will see one PGAS implementation in Fortran language. Then we will give a brief introduction of Message Passing Interface, the de facto standard in distributed memory environment. Also we will discuss about two task decomposition patterns and understand why the task decomposition make the subset of processes feature important to have. Finally we give a short survey of two on-going Coarray Fortran projects.

2.1 PGAS model

Programming models are referenced as the style of programming where execution is invoked. In parallel computing, the programming models often expose features of

hardware in order to achieve high performance. In this way, programming models in high performance computing define both the storage of data and the way data are manipulated and the way programs are executed and collaborated. We defined the term of how data is stored and referenced as memory model, and how tasks are organized and executed as execution model.

In last few decades, people defined several parallel programming models to help abstract the parallel computing system interfaces. Given the broad variety of architectures, obviously, many different programming models have been proposed to represent features of underlying parallel machines and memory architectures.

There are several literatures distinguishing existing popular programming models [33][15]. In general, we can classify the most popular parallel programming model into two categories based on the memory architectures:

- Shared Memory Model. Multiple threads execute their own instruction streams, with the access to the same shared memory space. This programming model implies a convenient way for threads to communicate via data in the same memory space. However, the problem behind this model is also obvious. The scalability is greatly determined by the data affinity and reference locality. The shared memory model usually use the fork-join model for execution.

- Distributed Memory Model. Multiple threads execute their own instruction streams, and they can only access to data in their own memory space. So the basic execution unit for distributed memory model is process with distinct memory space. The communication between processes is carried by message

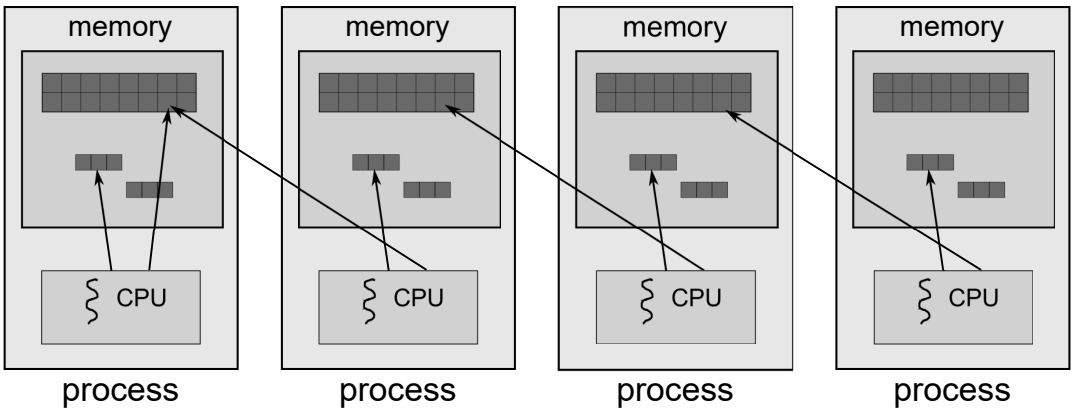


Figure 2.1: PGAS programming model

passing. The distributed memory model usually employ the Single Program Multiple Data execution model for executing the entire program.

In general, different programming models represent different abstraction of physical system. How well the performance we can achieve by employing certain programming model is determined by how well it matches with the underlying system.

The PGAS programming model, as shown in Figure 2.1programming model we follow in this thesis, is considered to suit shared and distributed memory parallel machines[2][44][51]. Ideally this programming model marries the performance and data locality (partitioning) features of distributed memory model with the programmability and data referencing simplicity of a shared-memory (global address space) model.

A system follows the PGAS programming model is expected to have following features:

- A set of programming units, maybe referenced as “threads”, “images” or “Processing element” in different implementations. Each unit has local memory storage attached to it.

- A mechanism for each unit to share at least part of its memory space for other unit to access.

- Every shared memory location has explicit affinity to certain units.

Different from the message passing programming model, the PGAS programming model employs the one-sided manner. In one-sided communication, a process can update or interrogate the memory of another process without any intervention from the destination process. In such way, the one-sided communication de-couples the communication with synchronization. On the other hand, comparing with the shared memory model, the model exposes its data affinity. In other words, the program is aware of where its data objects reside in relation to the one or more processing entities that are executing.

Trend in large scale computers shows the system will be based on distributed memory platform which contains much more nodes compared with today’s computer system. Meanwhile each node is transforming from multi-cores to many-cores. With the growth of core count, both pure share-memory model and distributed memory model will meet their constraints in scalability. Parallel languages and library which use PGAS programming model could potentially offer the scaleable performance both between the nodes and within the node.

2.2 Fortran in HPC

Fortran is a high-level programming language that is used in scientific programmings. The language is a procedural, imperative, compiled language with a syntax well suited to a direct representation of mathematical formulae. Individual procedures may either be grouped into modules or compiled separately, allowing the convenient construction of very large programs and of subroutine libraries. Fortran contains features for array processing, abstract and polymorphic data types, and dynamic data structures. Compilers typically generate very efficient object code, allowing an optimal use of computing resources. It is an old but still dominant base language in High Performance Computing (HPC) area.

The history[40] of Fortran language, which is an acronym derived from FORmula TRANslation, dates back to 1953. In the first version of Fortran, it contains several early form of construct that we can found in every high-level language, including the simple variable, assignment statement, DO-loop and etc. In following two decades, the Fortran language kept evolving until a new revision was published in 1978, becoming known as FORTRAN 77. This standard eventually gave Fortran the position as most commonly used scientific programming language.

Fortran always plays its important role in the area of numerical, scientific, engineering, and technical applications. And the ISO committee ISO/IEC JTC1/SC22 /WG5 (or simply WG5) keeps this language up to date and serve the scientific programmers' needs. They have proposed a new standard which is known as Fortran 90. The main enhancement they have introduced to this version is array operations.

This standard was formed at the age of supercomputer and vectorized code. This version is designed with the consideration of optimizations. The array operations, array assignments, array section and intrinsic procedures for array benefit the users of this language in terms of shorter and cleaner code, reliable executable.

Following the publication of Fortran 90 standard, a minor revision was under construction, which became what we called Fortran 95. Since it only adds several changes to the Fortran 90, the mostly wild used Fortran standard is called Fortran 90/95. At the same time, the High-Performance Fortran Forum (HPFF) was formed.

As the name has implied, the HPFF spent their effort in making extensions to Fortran language to produce portable, single-threaded code working in parallel machines. This work is called High Performance Fortran (HPF). The HPF follows the data parallel model where the data is represented as regular grid spreading over multiple processors. This allows efficient implementation on SIMD and MIMD architectures. Given such syntax nature, they chose the Fortran 90, which has adequate array language, as the base language. HPF features includes:

- New Fortran statement, such as *FORALL* statement for loop-level parallelism
- Directives for distribution of array data, such as

  ```
  !HPF$ DISTRIBUTE A(BLOCK,*)
  !HPF$ ALIGN B(I,J) WITH A(I,J)
  ```

- Additionally intrinsic library routines.

HPF left lot of optimization opportunities for compilers. During this period, lot

of papers have been published about implementation and optimization of HPF in compilers[38][6][24]. Unfortunately, although HPF has engaged some interest and success at the beginning, it eventually faded away. Lesson has been learned from this parallel language experiment[34]. However, the HPF did introduce some pioneer ideas to the HPC field. It inspired many successors including Fortran and its variants, OpenMP and many more HPC languages and it did influent the way in which upcoming Fortran parallel extensions are designed.

There are a large Fortran code demanding for speedup when no parallel feature introduced officially into the standard. Several auxiliary libraries and extensions, besides the HPF mentioned above, have been proposed. The OpenMP and MPI, which are very famous nowadays, formalized their Fortran interfaces. Also, R. W. Numrich and J. K. Reid[42] published a paper describing their proposal of parallel extensions for Fortran called Co-Array Fortran, which has been absorbed to recent Fortran language standard and it is the main topic of this thesis. We will talk about the syntax of Coarray Fortran later in section 2.4.2.

It was no parallel features included into Fortran standard until Fortran 2008 is announced. In the language specification we can see those handful Co-Array syntax and some necessary intrinsic routines that come from the Co-Array Fortran extension. As they are added into the standard, several commercial compilers, including Cray compiler and Intel compiler, are supporting these features. Some academic groups have studied its performance and potential[3][13] in both academic benchmarks and industry prototype codebase. More auxiliary parallel features are under discussion in the draft Technical Specification TS18508 Additional Parallel Features in Fortran[14].

2.3 Message Passing Interface

Message Passing Interface (MPI)[22][21] is the de facto parallel programming standard commonly used in distributed memory environment. It defines the syntax and semantics of a core of library routines to users writing portable message-passing programs, with binding to multiple language,such as Fortran, C and Java, etc. MPI provides many interfaces to users, however, the MPI 1.1 has four key concepts defined and all interfaces in MPI 1.1 serves for them[49].

- Point-to-point communication.
- Collective, operations that all MPI processes in certain communicator should anticipate in.
- Communicator, represents for arbitrary subset of all MPI processes.
- Derived datatype

Later on, these four concepts are widely accepted so that one can find them in almost any SPMD parallel programming implementation. So does in Coarray Fortran. However, communication defined in MPI 1.1 and 2.0 behave in a 'two-sided ' manner, where remote data accesses between two processes requires both sides actively participating in the same communication. This manner couples the synchronization and communication, implying some necessary overhead in every point-to-point communication. And it force the users to assure the sending and receiving pair matching.

MPI-2 introduces one-sided communication operation, such as *put* and *get*. And

MPI-3[47] has introduced more remote memory access (RMA) operations, such as `MPI_RMA_put` and `MPI_RMA_get`. Given the widely adoption of MPI implementation in variety hardware platform and enhanced RMA support in MPI-3, more and more PGAS languages and libraries consider MPI as underlying communication layer for their PGAS implementations[50].

2.4 Coarrays in Fortran 2008

In section 2.2 we have gone through the history of Fortran. In this section, we will have a closer look to the parallel features that are defined in the latest Fortran language specification, the Fortran 2008. It also gives our some flavor about the usage of Coarrays in Fortran program.

2.4.1 Execution unit

In Coarray Fortran program, the execution unit is called *image*. As a SPMD model, Coarray Fortran program is consist a set of *images* which are launching at the beginning of program and running in parallel. One can specify the number of *images* 1) via the environment variable, 2) by specifying to the job launcher. The number of *images* is fixed during the execution of the program. Each image has an identifier referred as *image index* in the range of [1...`num_images`]. Two intrinsic functions `this_image` and `num_images` are provided for user to query about the *image* identification and total number of images running during the program. Intrinsic `image_index` is used to

identify the image index belonging to a co-subscript. Each running image is executing on a separate processor, and it only performs computation on the data belonging to its local memory, or coarrays of its own copy. Part of images' memory are shared between images, which means other images may access to other images' memory space in a one-sided communication manner.

2.4.2 Coarray

As we shown in Figure 2.1, the PGAS program has a chunk of memory out of each processors' memory space that are logically shared among processors. In CAF and OpenSHMEM, we usually refer this chunk of memory as *symmetric memory*. Coarrays are declared and resident in *symmetric memory*. Coarrays are declared with the `codimension` attribute specifier. Codimensions are syntax similar to array dimensions, in that they may be used to describe a multi-dimensional rectangular shape associated with a data entity. Codimensions describes the rectangular *co-shape* of the set of images which each contain the coarray variable in its memory.

Coarrays may be transferred as dummy arguments into a procedure, as long as the associated actual argument is also a coarray. Otherwise, a coarray must be declared with either the `save` or `allocatable` attribute, for static coarrays and dynamic allocatable coarrays respectively. 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, as shown in Figure 2.2. Allocatable coarrays have a *deferred* co-shape; the dimension bounds should specified in the

`allocate` statement with the last codimension specified with an asterisk.

Codimensions associated with a coarray provide a means to uniquely refer to the coarray located on another image, which we called *image selection*. When a coarray variable is referenced with *cosubscript*, similar as subscript to array selection syntax but surrounded within square brackets, the compiler will identify it as a remote memory access to the coarray at the image identified by the cosubscripts. The intrinsic function `image_index` will return the image index of the image containing a specified coarray variable with a specified set of cosubscripts. Unlike the array reference, cosubscripts must uniquely refer to a single image. It cannot use subscript triplets and vector subscripts to refer more than one images at once. In Fortran, a data object is remote 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 corrary.

Remember that in Fortran 2008 there is no collective operations, which means if the user want to have certain collective function, say *broadcast*, he will have to implement the logic by himself. As shown in snippet Figure 2.2.

2.4.3 Image control statement

The language contains several types of image control statements that provide various synchronization facilities to programmers. The most fundamental of these are the `sync memory` statement and `sync all`. The `sync memory` acts as a local memory

barrier and a *fence* for remote memory accesses. A `sync all` statement is a synchronization barrier that any image executing the statement will wait until all images have reached the same `sync all` statement. Also there is a `sync images` statement, accepting a list of image id and do pair-to-pair synchronization. The coarray `lock_type` variables are used with `lock` and `unlock` statement to provide mutually exclusive operation primitive. The `critical` and `end critical` statements are used to define a critical section in the program that must be executed exclusively by one image.

Here we show a code demo in Figure 2.2 of CAF program with the Coarray declaration and reference, as well as a simple synchronization statement.

```
integer, save            :: A[*]
integer, allocatable     :: B(:)[:]

integer :: nimg, me
me = this_image()
nimg    = num_images()
allocate(B(nimg)[*])

A = me
if (me .eq. 1) then
        B(:) = (/ (i, i=1,nimg) /)
end if
sync all
if(me .ne. 1) then
        B(:) = B(:)[1]
end if
```

Figure 2.2: CAF code snippet using only Fortran 2008 features

2.4.4 Termination

A Coarray Fortran program may terminate in one of two modes - *normal termination* or *error termination*. When an image reaches to the end of a program or executes the stop statement, it will terminate properly in three steps: initiation, synchronization

and completion. One image cannot terminated until all images reach the second step of normal completion. Error termination occurs when an image meets an error condition. This could occur when it encounters an error state as defined by the standard, when the program executed an `error stop` statement, or maybe it is notified that some other image is in error termination. When any image initiates error termination, the runtime should attempt to signal all images as soon as possible to shut them down and return an error status code.

2.4.5 Atomic variable and subroutines

The compiler also supports atomic object and operations upon them. Atomic variables are declared with `atomic_integer_kind` or `atomic_logical_kind` attribute. Currently we have supported two subroutine related to it: `atomic_define` and `atomic_ref`. In latest specification, s set of new subroutines for atomic variable are listed and the OpenUH compiler has support them in the latest runtime library release.

2.5 Task decomposition in parallel program

While SPMD has been commonly adopted as the programming model followed by many parallel languages, libraries and interfaces, its restrictiveness does imply some drawbacks. The pure SPMD programming model implies a relatively flat machine model, with no distinction between any execution unit that may resident in close or

far-away physical location. Most PGAS languages and libraries lack the capability to expose this awareness of underlying hierarchy. As consequence, the communication costs that are not transparent for users hurt the performance. Similarly, it make the PGAS model, which should be easily adopted to heterogeneous environment, become difficult to program for heterogeneous machine.

In MPI[22], the *communicator* serves as the representation of subset of processes. It enables programmers to express the potential task parallelism in problem space. There are two possible task parallelism we can discuss.

The first program pattern to consider is parallel divide-and-conquer method. In classical algorithm study, the divide-and-conquer is a common method applied to take down the task size and solve them in reasonable time. In Dr.Hardwick's work[25], he summarized the practical divide and conquer method and proposed the term *team parallelism*. This method is data-oriented, that is, the parallel program can benefit from dividing the data into small chunk and processing them concurrently. The procedures for each *team* or subset of processes are all identical. There are several example applications following this parallel programming pattern, like parallel quick sort and parallel sample sort[27]. In such way, we would like a nested *team* construct to better map to the underlying memory hierarchy.

The second pattern is more general. In parallel programming with task parallelism, it is common for different components of an algorithm to be assigned to different units. For example, a climate simulation may assign a subset of all processes to model the atmosphere, while an other subset to tackle with the ocean model. Each of these components can in turn be decomposed into separate parts

for different purposes or algorithms. For instance, a subset such as one piece that performs a stencil while another piece could do Fourier transform. In such case, the decomposition does not directly depend on the layout of the underlying machine, although we can tell it would be benefit to assign threads nears to each other into one functional team.

Many researchers working in PGAS field realize the need for such method to do decomposition in parallel program. Since 2012, researchers who was working on Titanium project and UPC project published their proposal to add hierarchical additions to the SPMD programming model[31][32]. They proposed the *nested team* construct to express the hierarchy of underlying machines. Compared with MPI, this *nested team* construct is more structural, however, it serves as a better abstraction of hierarchy, other than a general subset of processes. Later the OpenSHMEM work group[48] has proposed their *team* design regarding to the task decomposition. The *nested team* has been identified as a reasonable way to carry the work.

2.6 Survey of coarray fortran implementations

As far as we know, Cray has always been working on the latest Fortran language features and always achieve the most efficient performance. But we can only know little detail about this commercial compiler. Here we will talk about two open-source project, OpenCoarrays project and CAF 2.0 compiler from Rice University.

2.6.1 OpenCoarrays

OpenCoarrays[19] is an open-source software project that produces an application binary interface (ABI) supporting coarray Fortran (CAF) compilers, an application programming interface (API) that supports users of non-CAF compilers, and an associated compiler wrapper and program launcher.

OpenCoarrays is not a module for a particular compiler. It is a portable translation layer that supposed to convert the Coarray syntax to an abstract API that should adapt to any non-CAF compiler later on. Right now it works with GNU Fortran compiler (gfortran). It supports Coarray syntax introduced in Fortran 2008, and some features in Technical Specification.

2.6.2 Rice CAF 2.0

The research group in Rice University started supporting of Coarrays in an early stage[16]. They have implemented a prototype of an open-source, multi-platform CAF compiler that generates code for most common parallel architecture. The cafc compiler translates CAF into Fortran 90 plus calls to one-sided communication primitives, which can further be mapped to certain communication layer library calls. In this implementation, they have supported Coarray descriptor, shared memory object management, translating coarray assignment to communication calls and some intrinsic procedures. Basically, the first version of cafc compiler supported most syntax mentioned in Fortran 2008.

Soon after, the research group found the shortage of functions these syntax can provide to user. In publication[39] after the announcement of Fortran 2008 standard, they summarized what they think are missing in this extension as following.

- There is no support for processor subsets
- The coarray extensions lack any notion of global pointers
- There is no support for collective communication

To address these shortcomings, Rice University is developing a redesign of the Coarray Fortran programming model. Rice's new design for Coarray Fortran, which is call Coarray Fortran 2.0, is a set of coarray-based extensions to Fortran designed to provide a different parallel programming model. Compared to the emerging Fortran 2008, Rice's new coarray-based language extensions include some additional features:

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

Rice's implementations of Coarray Fortran 2.0 has the similar way as we have implemented Fortran 2008. The compiler would translate the Coarray Fortran program to Fortran90/95 program with calls to runtime library primitives.

Although the group claims the Coarray Fortran 2.0 is still a progressing work, they have switched their focus to hierarchical PGAS or APGAS. *Team* also show in context but in a little different manner.

Chapter 3

Infrastructure

3.1 OpenUH compiler

OpenUH[37][11] 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. In Figure 3.1 shows the overall compiler infrastructure for OpenUH. Its modern and complete framework for inter- and intra-procedural state-of-art analyses and optimization is the most prominent part of Open64/OpenUH. OpenUH uses a tree-based IR called WHIRL. It comprises 5 levels, namely Very High(VH), High, Medium, Low and Very Low(VL), to enable a broad range of optimizations in appropriate level.

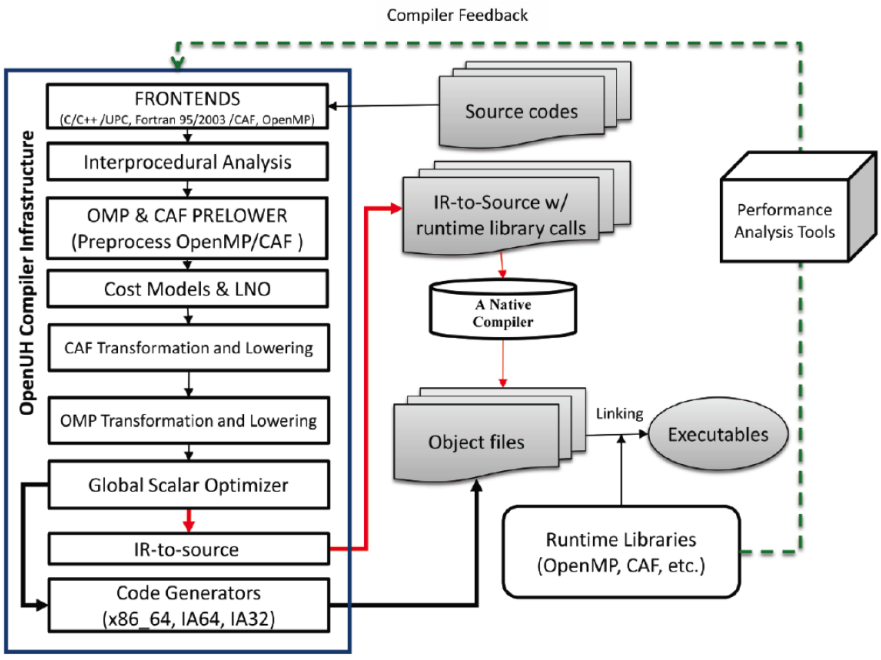


Figure 3.1: OpenUH compiler infrastructure

The major functional parts of the compiler that we may concern is Fortran front-end, the inter-procedural 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 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, etc[29][30].

OpenUH provided a solid base infrastructure for exploring implementation strategies for Coarray Fortran. The Fortran 95 front-end, which is contributed by Cray, was already capable to recognize coarrays and parsing the cosubscript syntactic extension. We took it as the start point for our implementation. The multi-level WHIRL IR, used throughout the middle-end and back-end of the compiler, provides rich support for a wide range of program abstractions. 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 access into bulk communication function calls. 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 port our CAF runtime to be used very easily with different compiler.

3.2 CAF runtime structure

In OpenUH compiler, we separate the runtime part from compiler part. In such way, we can specify different version of runtime library when building the compiler. Also the functionality of runtime can be reused by other program other than the OpenUH compiler only.

The CAF runtime is composed of two layers:

- Compiler interface layer. This layer performs as interface between compiler and runtime routines. During compilation, the compiler transform statement to runtime calls, such as *comm_put* or *comm_get*, as well as allocation, synchronization and inquiry functions.

- Communication layer. This layer conveys communication through the network. Our runtime does not talk with network directly. In stead, we employ GASNet/ARMCI library as our underlying communication layer. For this layer, we have separate implementation using GASNet and ARMCI, respectively. One of two library is linked depending on the layer we have specified. One can compile both layer when compiling the compiler, but one need to choose one of them by passing value to environment parameter when compiling the CAF program.

The Figure 3.2 describes the overall structure of CAF runtime.

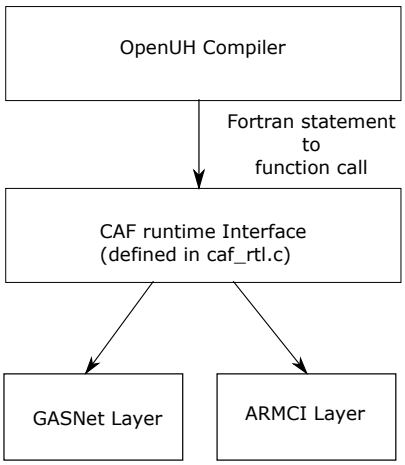


Figure 3.2: CAF runtime structure in OpenUH compiler

3.3 GASNet

In our CAF runtime, we chose two libraries as our communication layer, GASNet and ARMCI. In this thesis, I will only focus on the first one, GASNet, since most of my work is done and verified on top of it. GASNet[5][4], for the abbreviation of Global Address Space Networking library, is a language independent runtime library developed and maintained by a research group at the University of California Berkeley and Lawrence Livermore National Laboratory. GASNet was designed to serve the UPC and Titanium, which are two famous PGAS languages with base language in C and Java. Soon it has been widely used to support a variety of PGAS implementations including the Co-Array Fortran and CAF 2.0 from Rice University. We also found it in Cray's UPC and CAF compiler for the CrayXT series, the Cray Chapel compiler and OpenSHMEM reference implementation, which is developed

and maintained by University of Houston.

GASNet has an API specification which defines interface for compilers or runtime system to use, which we will briefly review in this section. GASNet was designed with portability and performance in mind, and it includes implementations for many popular network APIs covering the common cluster interconnects, as well as specialized interconnects from IBM and Cray.

A running program which uses GASNets, called the client program, consists of a set of operating system processes, or *nodes*. These nodes may furthermore be grouped into super-nodes, 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:

- GASNET_SEQ allows only one single thread on a node the make any GASNet calls
- GASNET_PARSYNC allows only one thread to make GASNet calls at a time
- 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 environment variables in GASNet, and mutexes for multi-threaded client programs. In this thesis, the most significant part of the core API is the *active message* (AM) support. This provides

means to invoke registered AM handlers on a remote GASNet node. The 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, implicit-handle non-blocking *get* and *put*, and explicit-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, we can also find support for non-contiguous remote data transfer in Berkeley's implementation.

3.3.1 Active Message

Active message is a set of communication primitives that performing the processing on its own. Compared with non-blocking MPI, it is a lightweight messaging protocol used to optimize the network communication. The basic mechanism of Active Message in GASNet is shown in Figure 3.3, the mechanism we use exactly in implementing the distributed mapping optimization. The request side launches request with some data payload to the network, with specifying the handler on the receiver side. Once the receiver side receives the message, it will invoke the corresponding request handler, processing the data and may return a reply message with or without data payload attached. When the reply message arrives the request side, the reply call-back function is invoked and finish the whole process. Usually the reply handler

will notify the upper-level function that the completion of transaction. There are two points to keep in mind, 1) every request and reply message must have handler registered at the beginning of program, 2) The reply handler cannot launch further message request.

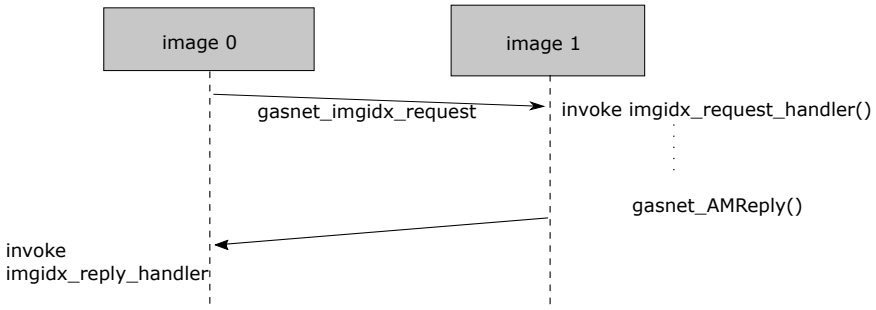


Figure 3.3: Active Message to query the image index

Chapter 4

Implementation

4.1 Additional parallel feature syntax

In this section, I will describe the additional features described in the Technical Specification draft.

Previously, we implemented support for Fortran coarrays in OpenUH in accordance with the Fortran 2008 specification [18] [17]. The OpenUH Coarray Fortran implementation is depicted in Figure 4.1.

We added support into the Fortran front-end of OpenUH for parsing the `form team`, `change team`, `end team` and `sync team` constructs[20]. We added the new type `team_type` to the type system of OpenUH and support for `get_team` and `team_number` intrinsics. We also extended the CAF intrinsics `this_image`, `num_images`, and `image_index` for teams. During the back-end compilation process in OpenUH,

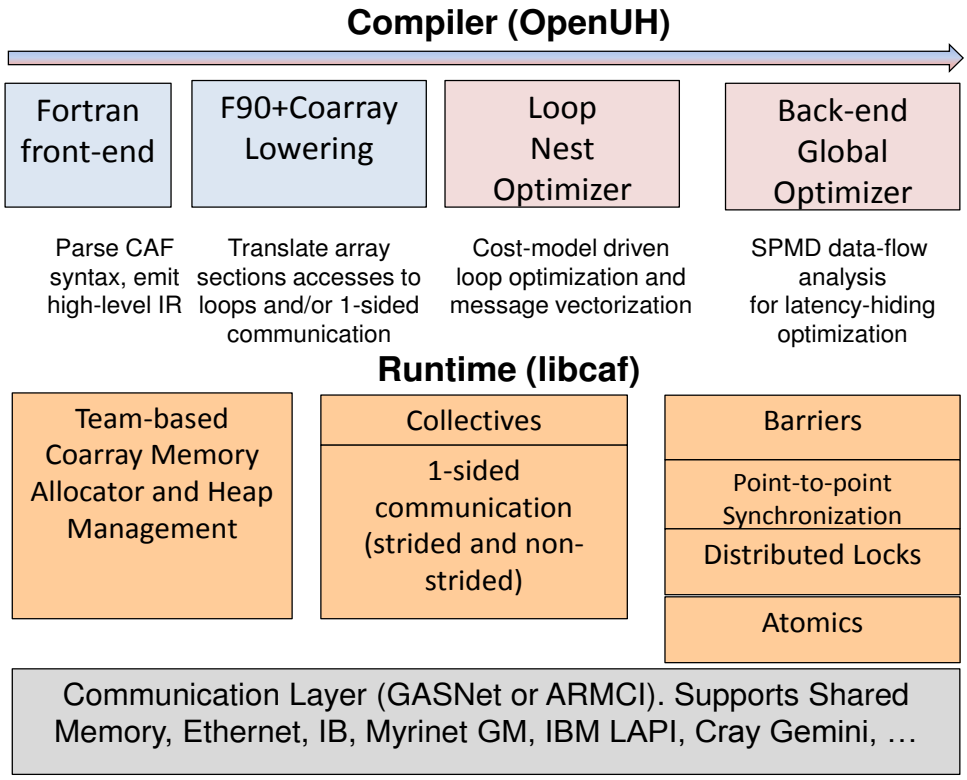


Figure 4.1: OpenUH Coarray Fortran team implementation

team-related constructs are lowered to subroutine calls which constitute the *libcaf* runtime library interface. In the runtime, we added 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. The runtime also provides support for the team-related intrinsics `get_team` and `team_number`.

In following Figure 4.2, we present the basic usage of these new Coarray syntax.

```
type(TEAM_TYPE) :: I,A,B
integer :: id
integer,allocatable :: d1(:)[:], &
                       d2(:)[:], &
                       d3(:)[:]

I = get_team()
allocate(d1(10)[*])
id = (this_image()-1)/2+1
form team(id, A)
change team(A)
  allocate(d2(10)[*])
  if(team_number() .eq. 1) then
    form team(this_image(), B)
    change team(B)
      allocate(d3(10)[*])
    end team !exit team B
 end if
end team
```

Figure 4.2: Code depicting allocation of coarrays inside teams

4.2 Base Team Implemenation

In this section, we will present the idea of how to implement a basic but functional team structure in runtime and what other change we made to the existing runtime library. These implementation details are presented in three aspects: 1) Symmetric memory management, 2)data structure and subroutine for team construct, 3) Changes to make barrier and collectives team-awarable.

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 is no longer necessary. According to the draft of the technical 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 needs to refer to a coarray of another image located in a sibling team, the coarray should be allocated in their common ancestor team.

Team variables are opaque, first-class objects which may be used to query information for a specified team or change to a specified team. In our implementation, a team variable refers to an associated team data structure, depicted in Table 4.1. During the formation of a team, the values for the fields of this data structure are

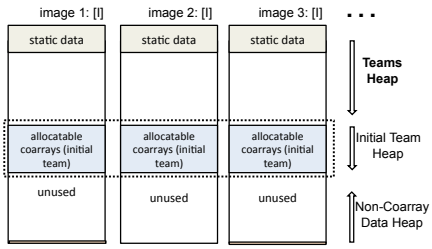
computed and populated, including (1) the list of images that are on the same node, (2) the number of images within the same node, and (3) fields used to facilitate execution of collective operations. This information is used many times in the runtime by different parallel algorithms (for example, collectives and barriers as described in Section 4.2.3).

4.2.1 Memory Management

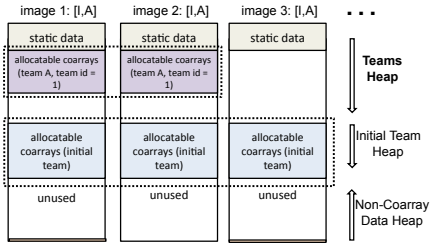
Before incorporating support for teams, we implemented the managed heap as follows. At the beginning of the program, the images collectively allocate a pinned and registered memory segment which may be used for remote memory accesses. Static data which is allocated for the entire lifetime of the program is placed in a reserved space at the top of this segment. The rest of the segment is treated as a managed heap. Allocatable coarrays are symmetrically and synchronously allocated on all images from the top of this heap. We also allow non-symmetric allocations from the bottom of the heap. This serves a few different purposes. We can allocate temporary communication buffers from the bottom and avoid the cost of pinning and registering it. Additionally, even though Fortran 2008 requires all coarrays to be symmetric across all images, the coarrays may indirectly point to non-symmetric data. This is achieved by declaring the coarray to be of a derived data type with a pointer or allocatable component, for which the target data may be allocated independently of other images. This allocated data may then be remotely referenced using the coarray.

Table 4.1: Team data structure

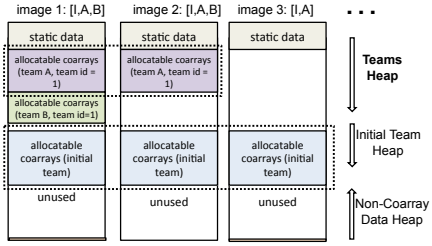
Field	Description
team_num	a team number or id, assigned during `form team` statement
this_image	image index for current image in team
num_images	number of images in team
intranode_set	ordered list of image indices in same compute node
leader_set	ordered list of image indices of node leaders in team
leaders_count	number of node leaders in team
image_index_map	maps image index to image index in inital team
sibling_maps	image index mapping for each sibling team created by same `form team` statement
bar_parity	parity variable for dissemination barrier
bar_sense	sense variable for dissemination barrier
intranode_bar_flags	direct shared pointers to intra-node barrier partners' flags
bar_rounds_info	partner information for inter-node in dissemination barrier rounds
coll_sync_flags	bcast, reduce, and allreduce sync flag
allreduce_bufid	selects between two allreduce buffers
reduce_bufid	selects between two reduce buffers
bcast_bufid	selects between two bcast buffers
allocations	a list of symmetric memory slots allocated for this team
parent	pointer to parent team structure



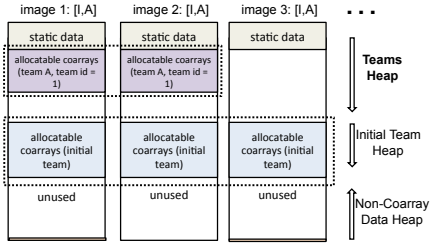
(a) Step 1: Allocate Coarray in initial team



(b) Step 2: Form new team A and allocate Coarray on image 1, 2



(c) Step 3: Form new team B and allocate Coarray on image 1



(d) Step 4: End team, exitting from team B

Figure 4.3: Evolving state of managed heap during team symmetric allocations

In order to support coarray allocation with respect to teams, we considered a few different approaches. The first approach was to reserve a fixed-size memory container for each team, within which any coarray allocations may be made. This could be achieved, for instance, through the use of *mspaces* in dlmalloc [36]. However, such an approach would require foreknowledge of how much space is required for each team, and in general we expected a considerable waste of allocated space using this approach. The second approach which we settled on is to instead reserve a fixed size heap for all teams except the initial team, which we call the *teams heap*. This turns out to be sufficient, since a coarray may only be allocated for a non-initial team when that team is active and none of its descendant teams have currently allocated coarrays. The initial team is an exception, since at any time an image may execute a `change team` statement to change back to the initial team, and hence a separate heap for the initial team is still maintained.

Figure 4.3 illustrates how our managed heap evolves over the course of the example program shown in Figure 4.2. When a coarray is allocated while executing a `change team` block, corresponding allocations should occur on all other images in the *current* team, rather than for all images. Upon exiting the `change team` block, any allocations that had occurred within it are implicitly freed if they were not already freed by a `deallocate` statement. An image may only *change* to a team with the `change team` construct if it was formed by its current team with a `form team` statement or if it is the initial team. The latter scenario requires that the state of symmetric allocations belonging to the initial team should not be affected by allocations (not yet freed) belonging to a non-initial team. To support this, we reserve a

fixed section of memory from the top of our managed heap for symmetric allocations by a non-initial team. We divide the list structure for memory allocations into two lists: one is for symmetric allocations by any non-initial team, and the other is for symmetric allocations by the initial team and all non-symmetric allocations. When changing to a new team, the *allocations* field in the team structure will be set to the current position in the non-initial team allocations list.

4.2.2 Forming and Changing Teams

The `form team` statement forms multiple teams by subdividing the set of images that are members of the current team. Each image in the current team must call the statement, specifying the number of the new team it will join and a team variable which it may use to refer to the newly formed team. A third optional argument may be specified to request a particular image index within the new team. It is otherwise implementation-defined how these image indices are assigned to team members; in our implementation, image indices are assigned to each image in the order of their indices in the current team.

Forming a new team entails a coordinated exchange of information from every image in the current team. Our implementation is currently as shown in Figure 4.4. All the images collectively perform an *allgather* operation to exchange the specified team numbers and (if given) image indices. Once this step completes, each image can determine the members (and their respective image indices) for each team formed. Based on this, each image can fill in the relevant information in the team data

structure which it associates with the newly formed team. If a requested image index was specified with the `form team` statement, this can be directly assigned to the *this_image* field. Otherwise, the new image index is determined by sorting the set of image indices which specified the same team number. The *image_index_mapping* field is a pointer to an array which maps an image's image index in the new team to its image index in the initial team.The *siblings* field, shown in Table 4.1, is a pointer to an array of image maps for every other new team formed. This is useful because an image may also access a coarray belonging to a different team, using an extension to the normal image selector syntax (e.g., $a[i, team\_number = 2]$).

In table 4.1 we included two components related to the super-node information. The leader set contains the image indices for images in the team serving as designated leaders for their respective compute nodes. The intra-node set contains the image indices for all images in the team that share the same compute node. The leader set and intra-node set may be computed based on the runtime's determination of the process-to-node layout for the job.

For barriers, we distinguish flags to be used for synchronization within a node via shared memory (available through *intranode_bar_flags*), versus flags used to synchronize between images on separate nodes (available through *bar_rounds_info*). These flags are also stored in the team data structure associated with each team, shown in Table 4.1. Pointers for accessing a partner image's synchronization flags at each round of the barrier are also precomputed and stored at this stage.

The `change team` statement is used to change the *current team* in which the encountering image is executing to a team referenced by a team variable argument.

When an image executes this statement in our implementation, it will simply change an internal *current_team* pointer to the address of the team structure referenced by the team variable. Next, all images changing to the same team will synchronize via an implicit team barrier (note that the program should generally ensure that all or none of the images in a team reach the statement, though its possible for an image to check for stopped or failed images during its execution). When `end team` is encountered, the runtime will set the internal *current_team* to point to the parent of the current team. If leaving a team which has itself created child teams, then the team structures allocated for each of those child teams may be freed, and the corresponding team variable will be set to a `NIL` value to indicate that it is no longer associated with a team. If the team had allocated coarrays out of the symmetric heap, the associated slots describing these allocations (in the *allocations* field) are freed. Finally, all images in the team it returns to must synchronize via an implicit team barrier. Note that whenever an image switches to a different team, through the `change team` or `end team` statements, it will always synchronize will all images which are members of that team. This ensures that an image will never be executing in a team while other members are executing in a different team. We make use of this fact in the synchronization-avoidance optimizations we implemented for collectives.

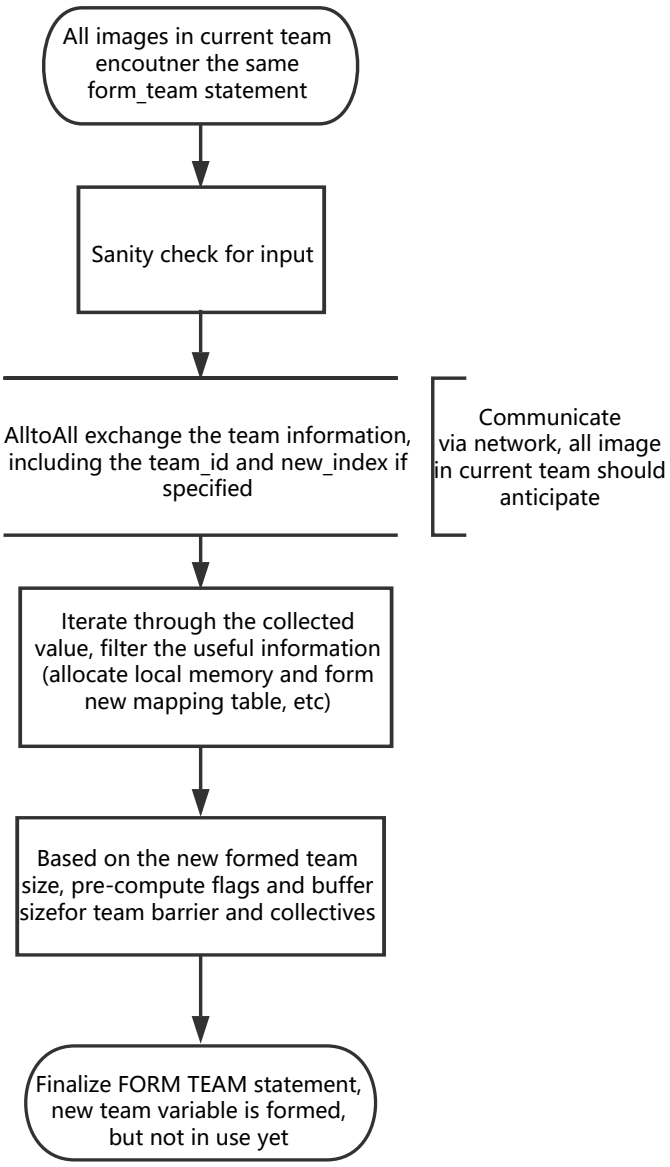


Figure 4.4: Flow chart of FORM_TEAM function in runtime

4.2.3 Synchronization and Collective Operations

Take the team into consideration, we need to modify the synchronization and collective operations, including barrier, synchronization and reductions. 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. The *allgather* support is used exclusively to facilitate formation of teams, and was implemented using Bruck's algorithm [7] 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*. These well-known algorithms, which we implemented using 1-sided communication, are described in [45]. 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 independent writes from both images to the other image in the pair (*allreduce*).

During the team formation step, we allocate various synchronization flags to be used for team-based barriers and collective operations. These procedures are not shown in the skeleton code in Figure 4.4.

Synchronization flags are also allocated and reserved for supported collective operations (specifically, *allreduce*, *reduce*, and *broadcast*) that may be executed by the

new team. This is necessary since we implement collectives using 1-sided communication which is decoupled from synchronization. Since these collectives entail different communication structures, in order to allow for their execution to partially overlap we allocate a distinct set of synchronization flags for each type during team formation.

Our collectives implementation makes use of a *collectives buffer space* – a fixed size, symmetric region which is reserved from the remote access memory segment. By default, the size is 4 MiB per image, but this may be adjusted through an environment variable. If this reserved 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 event that these operations are operating on large arrays which can not be accommodated within the buffer space available (determined by the image heap size), we can break up these arrays into chunks, and perform our algorithm on each of these chunks in sequence.

4.3 Runtime Optimizations

4.3.1 Runtime Data Locality Optimization

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 and fixed across the nodes in a non-uniform manner. We therefore developed 2-level algorithms for these collective operations which exploit the fact that the operations are presumed to be commutative and associative. 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*.

During the initialization stage, the GASNet communication layer provides us a function `gasnet_getNodeInfo()` to query the information of each gasnet node, which is image in this case. The node info structure includes an component called supernode, indicating the id of physical node, comprised by multiple cores that share a physical memory.

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 the 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.

We also enhanced our barrier implementation by taking advantage of node locality information provided by the underlying runtime layer, described in [35]. Within the runtime descriptor for a team there is a list of image indices for images that reside on the same node and a list of image indices for images 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, as described above. Finally, once the dissemination

barrier completes the leaders will notify its non-leaders that all images in the team have encountered the barrier. 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.

4.3.2 Distributed member mapping list

Consider the actual team structure in memory shown in Figure 4.5, the local memory usage is proportional to the number of images in this team. Besides, if we take the *sibling team* into consider and store the image mapping lists into local team structure, the memory usage will be proportional to the total number of images in environment. The program can benefit from this all-in-local manner when the number of images is small because the lookup cost in local memory is O(1). But it is an obvious factor that hurts the scalability.

One reasonable way to resolve it is to distribute the mapping information into different images and we need a mechanism to query each time when we need to send the put/get communication calls. The *team* structure only need to store a few information within it.

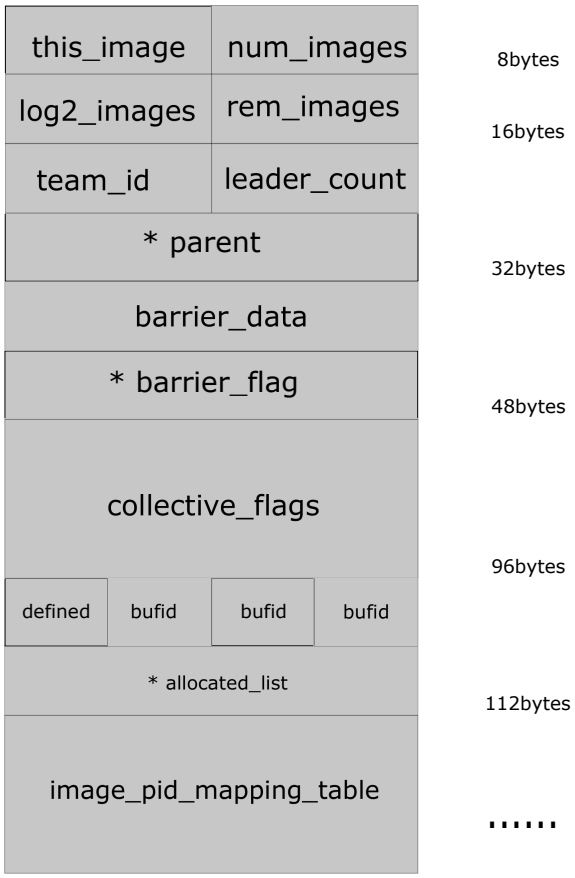


Figure 4.5: OpenUH Coarray Fortran team structure in memory

Here I proposed and implemented the distributed member mapping list in a simple manner. Each image contains the mapping information from its predecessor and successor. So in a global view, the images form a double-linked list. Each time when an image launch a communication call to image that is not its predecessor or successor, it will send an *active message* to look through this list.

We demonstrate the mechanism of *active message* in Figure 3.3 and implementation flowchart in Figure 4.6. Without much prove we can see the delay of the index

query will hurts communication performance severely.

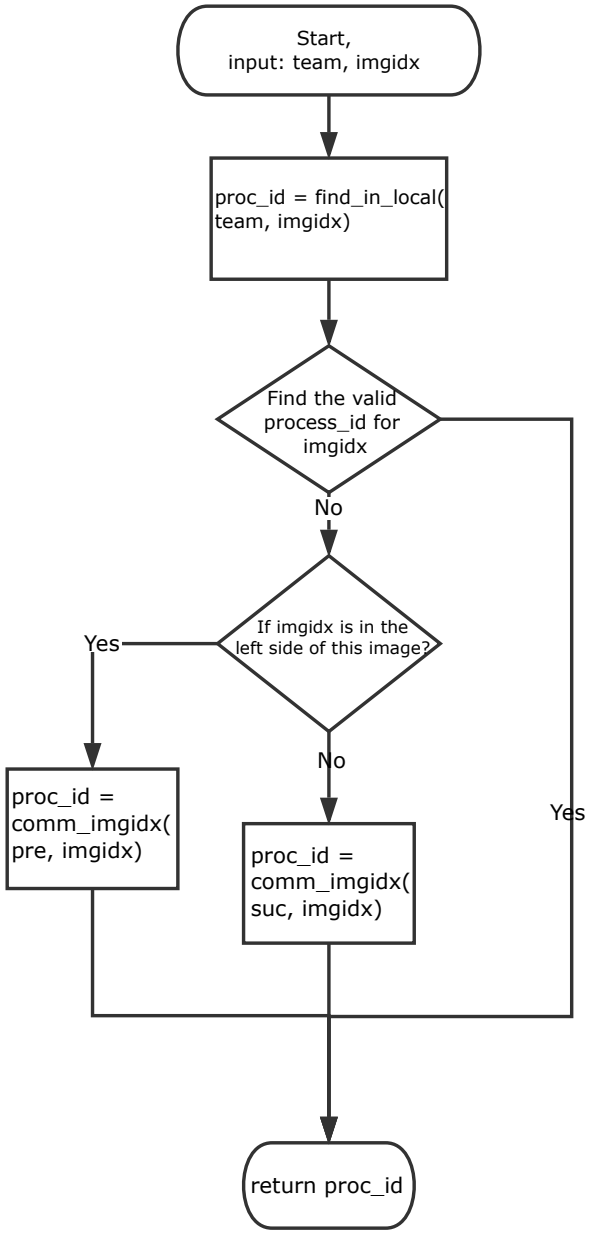


Figure 4.6: Flow chart of mapping image index to process id in distributed mapping table

To reduce the unnecessary network delay, we should try to store as much information as possible in the local memory so that the query function will return at the first step. In addtion to this query function, we also implemented a cache mechanism to store most recently queried result.

As we know there are several cache mechanism with different emphasis property. Here we implemented a simple LRU cache using hash table and double-linked list. We consider the amortized complexity of lookup operation on a hash table is O(1). The lookup and maintainess cost of this LRU is in constant complexity.

4.4 Incomplete parallel features

Besides the team features and collectives routines we described in previous sections, there are a bunch of parallel features need to be complete in OpenUH runtime.

1. *Team selector.* In nested team environment, the *team selector* provides users a shortcut to access coarrays resident in image of *sibling team.* In Figure 4.7 we borrow the code from specification to show a scenario where *team selector* could be useful. By switching between teams, accessing to coarrays across team boundary is possible but it is a less efficient way due to multiple unnecessary synchronizations. In the implementation design we have considered this syntax and leave interface to implement in runtime very quickly. The main obstacle is in front-end, i.e., how to recognize the syntax and pass it to the runtime.

2. *Failed images.* A failed image is one that has ceased participating in program execution but has not initiated termination. A failed image is usually associated with a hardware failure of a cpu, memory system, or interconnection network. The runtime should provide ways to track processes failure and propagates the failure across the network. The specification defines behaviors on reference to failed image, e.g, the lock object acquired by failed image should be unlock automatically. `FAIL IMAGE` statement make the calling image behave like failed. It is one of image control statement. In OpenUH runtime we have supported internally the image failure tracking and propagation. But any failure in current runtime will induce error termination. Behaviors on failure are not supported.

```
USE, INTRINSIC :: ISO_FORTRAN_ENV, ONLY: TEAM_TYPE
TYPE(TEAM_TYPE) :: INITIAL, BLOCK
REAL :: A(0:N+1)[*]
INTEGER :: ME, P2
INITIAL = GET_TEAM()
ME = THIS_IMAGE()
P2 = NUM_IMAGES()/2
FORM TEAM(1+(ME-1)/P2,BLOCK)
CHANGE TEAM(BLOCK,B[*]=>A)
DO
! Iterate within team
:
! Halo exchange across team boundary
        SYNC TEAM(INITIAL)
        IF(ME==P2 ) B(N+1) = A(1)[ME+1,TEAM=INITIAL]
        IF(ME==P2+1) B(0) = A(N)[ME-1,TEAM=INITIAL]
SYNC TEAM(INITIAL)
END DO
END TEAM
```

Figure 4.7: Usage of *team selectors* to access coarrays across team boundary

Chapter 5

Results

In this chapter, I will present the evaluations an evaluation of the implementation and optimizations described in this thesis.

5.1 Experiment Setup

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. We installed OpenUH 3.0.40, Rice CAF 2.0 (r4169), GASNet 1.22.4, and the latest GASNet 1.24.2 on Stampede for evaluations. The MPI implementation we used was MVAPICH2, version 1.9a2.

5.2 Benchmarks

5.2.1 Team Microbenchmark

5.2.1.1 Evaluation of Team Barriers

In Figure 5.1, we show timings for different use cases of team barriers. We arranged 4096 images to show 3 synchronization cases: 1) all participating images synchronize using `sync all`, 2) form teams and have images in each team synchronize using `sync team`, and 3) image subsets synchronize using `sync images`. Logically, the `sync images` statement with an image list consisting of all images in the same logical team" will have the same effect as a `sync team` statement using a team variable representing a team consisting of the same images.

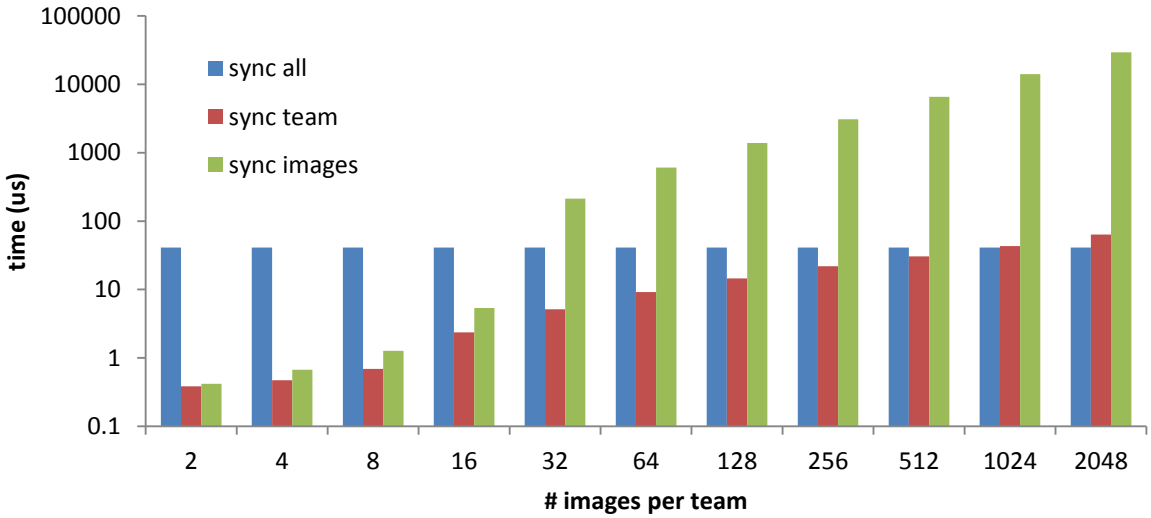


Figure 5.1: Barrier synchronization for groups of images (4096 total images), on Stampede.

The reader may notice that having all participating images execute `sync all`

performs reasonably well until the number of images per team reduces past a certain threshold. This is because we utilized the barrier provided by GASNet to implement `sync all` for the initial team, and it happens to be well tuned for the InfiniBand interconnect used on Stampede. Before `sync team` was proposed, synchronization among a subset of images could be achieved alternatively using the `sync images` statement. The scalability for this statement, however, quickly became an issue as the participating images increase, since the semantics of this statement requires that an image perform a point-to-point synchronization with each image in its specified image list. We observe here that `sync team` is a far more effective approach for synchronizing a subset of images compared to using `sync images`.

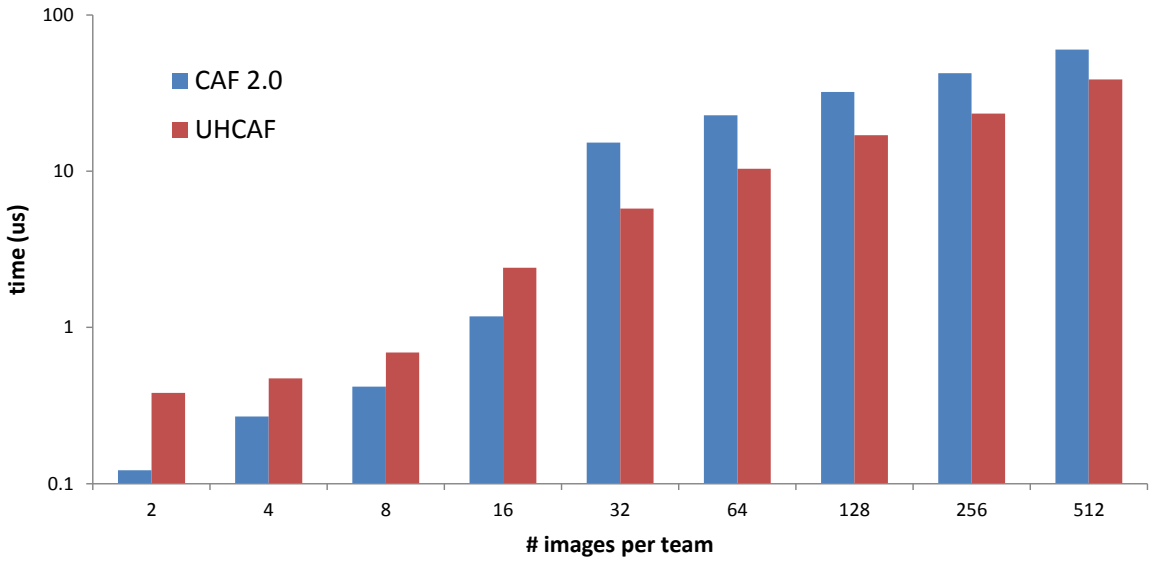


Figure 5.2: Comparison of team barrier between UHCAF and CAF 2.0 (1024 total images), on Stampede.

We also compared the performance of our team barrier implementation with the

equivalent `team_barrier()` routine available in the Rice CAF 2.0 implementation, shown in Figure 5.2. For this comparison, we used the most recent GASNet version, 1.24.2 (all other experiments described in this section used GASnet 1.22.4). The result shows that the CAF 2.0 barrier implementation was more efficient when the team size was less than or equal to 16, where all images in a team reside within the same compute node. On the other hand, our barrier implementation was more efficient when each team spans multiple compute nodes. We attribute this result to the 2-level barrier algorithm we've implemented, while there is evidently some improvements to be made in our intra-node barrier implementation.

5.2.2 Reduction

In the two charts in Figure 5.3, we compare the application of our 2-level optimization on the reduction operation to the original implementation, which uses the recursive doubling algorithm [46]. The two-level implementation uses binomial tree reduction from non-leaders to their node leader; then, a recursive-doubling all-reduce is performed between the leaders; and finally the non-leaders perform parallel local gets from their leader. We also compare it to CAF 2.0, Open MPI and MVAPICH. As expected, the memory hierarchy awareness in our two-level algorithm gives very good results (note the case where we have 8 images per node).

In the case of one image per node, not only is there no additional overhead compared to the original implementation, but we were able to improve the performance

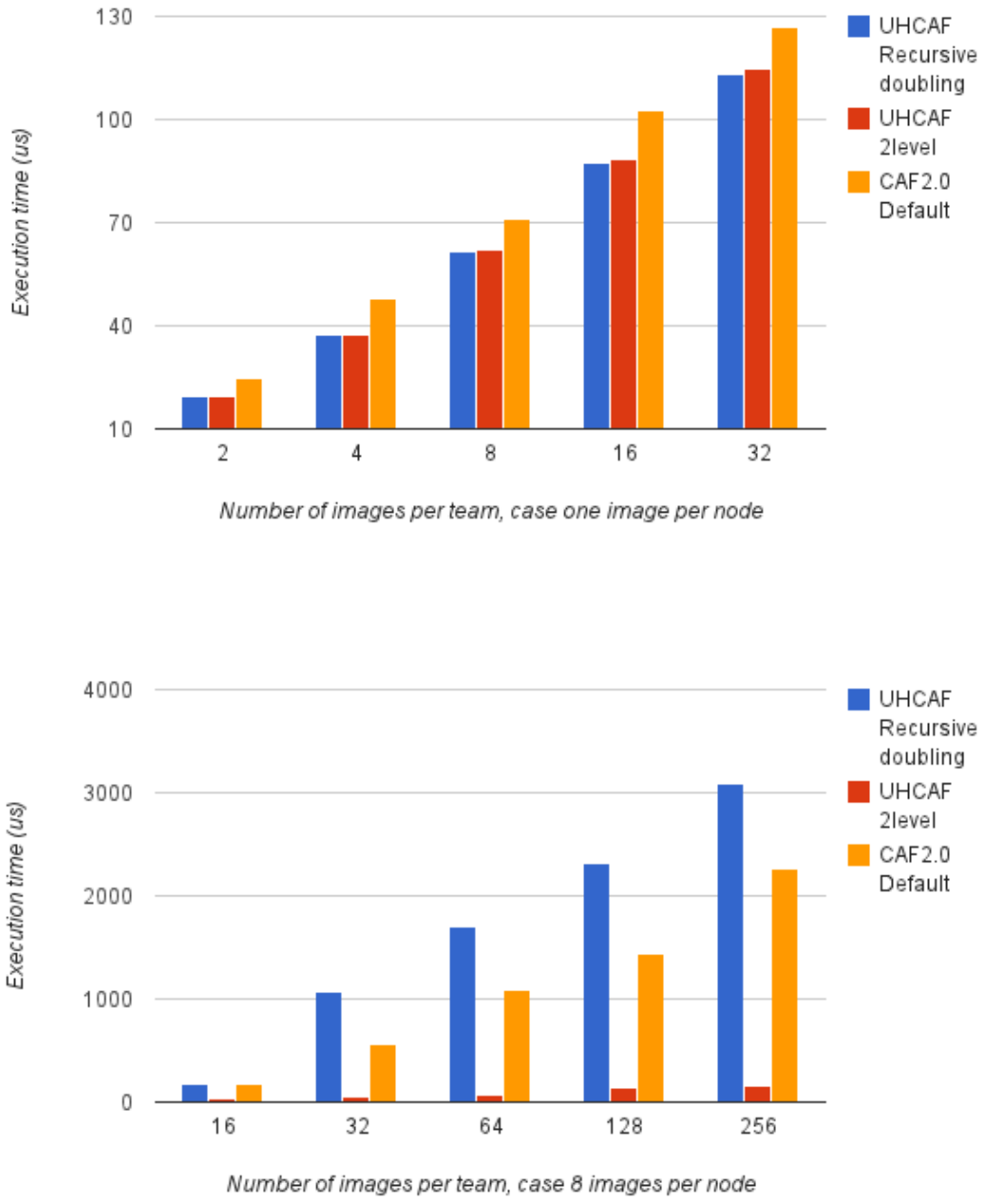


Figure 5.3: Performance evaluations for the 2-level reduction algorithm using the Teams Microbenchmark suite

by applying a further optimization. Using the 2-level approach advocated in this thesis, we can distinguish remote memory operations that access out-of-node memory via the interconnect's RDMA from memory accesses within the node. In the former case, we can employ the canary protocol [28], which entails the target polling on the last byte (or some bytes) as a canary value to check for communication completion (a valid approach because an RDMA write over Infiniband can be assumed to complete in byte order). By using this protocol, which effectively bundles a notification of completion with the data to be sent, we can eliminate sending an additional notification per write in our implementation.

5.2.3 Using Team-based Collectives for CG

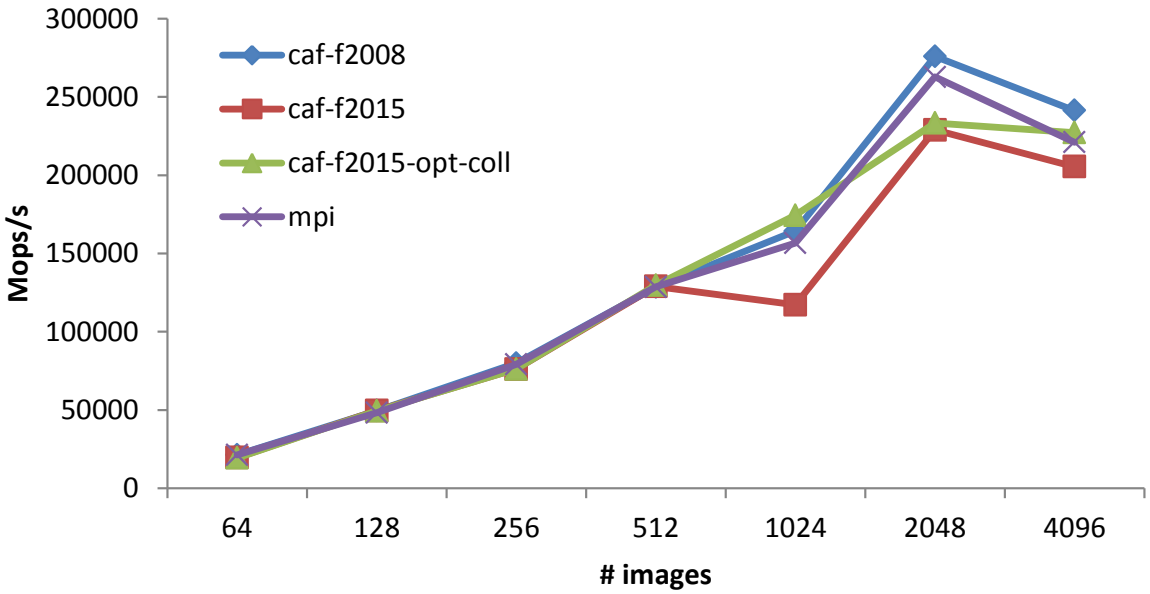


Figure 5.4: CG benchmark (class D) on Stampede, using 16 images per node

To assess the potential benefits of using the teams and collectives features, we updated our CAF implementation of the CG benchmark from the NAS Parallel Benchmarks (NPB) suite (available in [1]). The CG 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. We first ported this benchmark to use Fortran coarrays, adhering to the Fortran 2008 specification. For the extended version, 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. In this way, we were able to assess the utility of both the teams and reduction features that are expected to be included in Fortran 2015.

In Figure 5.4, we compare the results achieved with this new implementation on Stampede using class D problem size with our original Fortran 2008 version of the benchmark. We also show the results executing the original MPI version of CG using MVAPICH2. The baseline collectives implementation resulted in regressed performance relative to the Fortran 2008 version. Through synchronization-hiding and locality-aware optimizations of these collective operations, as described in Section 4.2.3, we were able to improve on the baseline performance. However, we observed scalability issues even when using our optimized reductions when running with 2048 and 4096 images. We believe the issue originates from the need to add the `change team` and `end team` statements before and after calls to `co_sum` for performing the row-based reductions. The `end-team` statement, in particular, entails a barrier synchronization for all images in the initial team. One way around this would be to surround the entire iterative loop executed in *conj_grad* inside a team

block, and utilize the new image selection syntax (e.g., $a(j)[i, team{=}init\_team]$) to perform the necessary communication and synchronization operations across teams (e.g. for the transpose operation). However, we have not yet implemented this image selection feature.

5.2.4 HPL

We implemented a Coarray Fortran version of High Performance Linpack (HPL) [43], which is used to solve systems of linear equation, thus testing temporal and spatial run-time locality. We based our version of HPL on its CAF 2.0 port, described in [23]. HPL makes use of row team *rteam* and column team *cteam* for performing updates of the matrix data. HPL computes LU factorization with row partial pivoting. The recursive factorization of a 1-dimensional panel of columns of processes is performed on *cteam*. The associated swaps and broadcasts of the pivot row are performed on *rteam*.

Figure 5.5 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, CAF 2.0 using OpenUH as backend compiler and Open MPI using GCC compiler. The `-O3` option is passed to the compilers in Figure 5.5. 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 on 256 cores, as compared to 29.48 GFLOPS/s obtained with the CAF 2.0 implementation with the GFortran backend and 80 GFLOPS/s with the OpenUH backend. Note

that we used the classic recursion parameters of the MPI version of HPL without tuning; we got results comparable to UHCAF's.

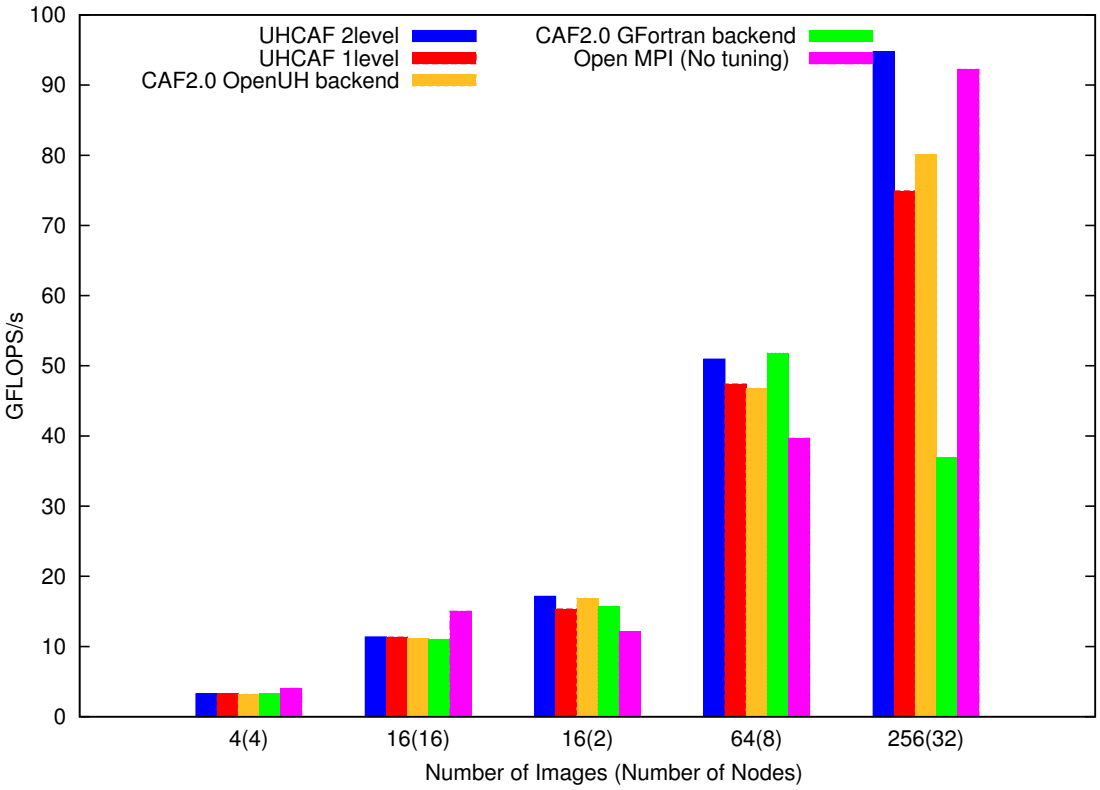


Figure 5.5: Performance results for HPL

Chapter 6

Conclusion

During this thesis, we have described the implementation and optimizations we developed in our runtime and compiler to support the additional parallel features listed in the Fortran language Technical Specification Draft. Our focus is on the *team* construct and collective procedures. The contributions of this work are summarized as following:

- We developed the first implementation of the anticipated team features expected to be added to Fortran 2015. These features include `FORM TEAM`, `CHANGE TEAM`, team managing and finalizing *team*.
- Support the execution of barriers and collectives within a team context.
- We evaluate the language features using the microbenchmark and two benchmarks from NAS Parallel Benchmark suite and High Performance Linpack (HPL) to show the effectiveness of these new features.

6.1 Future Work

In this thesis, we have discussed the basic implementation of *team* construct and collective operation.Taking this implementation as start point, there are several aspects we can explore as future work.

1. Finalize the support of additional features defined in the technical draft. According to the technical specification draft, we still need to implement several more syntax, including *image selector* and *failed images*.

2. Propose new features to current syntax set. We have identify some features are very useful but not in the specification. For example, the *gather* and *scatter* collective operations are two common subroutines found in parallel application but they are nt defined in current specification. Comparing with implementing them in form of additional Fortran module, implementing them in runtime library will let the routine access to internal data structure, which will benefit the subroutine performance.

3. Explore optimization opportunity in runtime and compiler side. As a language based PGAS implementation, one advantage for Coarray Fortran is that the compiler may generate more efficient code or messages by applying static analysis to the program. Given the syntax of Coarray Fortran, it is easier for compiler to recognize optimization opportunity by applying array access analysis and loop optimization techniques.

Furthermore, in[32] and later discussion about hierarchical construct in PGAS language, we have seen some cases where the team analysis can be benefit. We also consider this as part of future work that boost the power of language-based PGAS implementation.

Bibliography

[1] CAF Test Suite. `https://github.com/uhhpctools/caf-testsuite`.

[2] G. Almasi. Pgas (partitioned global address space) languages. In *Encyclopedia of Parallel Computing*, pages 1539–1545. Springer, 2011.

[3] R. Barrett. Co-array fortran experiences with finite differencing methods. In *The 48th Cray User Group meeting, Lugano, Italy*. Citeseer, 2006.

[4] D. Bonachea. Gasnet specification, v1. 1. 2002.

[5] D. Bonachea and J. Jeong. Gasnet: A portable high-performance communication layer for global address-space languages. *CS258 Parallel Computer Architecture Project, Spring*, 2002.

[6] Z. Bozkus, A. Choudhary, G. Fox, T. Haupt, S. Ranka, and M.-Y. Wu. Compiling fortran 90d/hpf for distributed memory mimd computers. *Journal of Parallel and Distributed Computing*, 21(1):15–26, 1994.

[7] J. Bruck, C.-T. Ho, S. Kipnis, E. Upfal, and D. Weathersby. Efficient Algorithms for All-to-All Communications in Multiport Message-Passing Systems. *Parallel and Distributed Systems, IEEE Transactions on*, 8(11):1143–1156, Nov 1997.

[8] W. W. Carlson, J. M. Draper, D. E. Culler, K. Yelick, E. Brooks, and K. Warren. *Introduction to UPC and language specification*. Center for Computing Sciences, Institute for Defense Analyses, 1999.

[9] B. L. Chamberlain, D. Callahan, and H. P. Zima. Parallel programmability and the chapel language. *International Journal of High Performance Computing Applications*, 21(3):291–312, 2007.

[10] B. Chapman, T. Curtis, S. Pophale, S. Poole, J. Kuehn, C. Koelbel, and L. Smith. Introducing openshmem: Shmem for the pgas community. In *Proceedings of the Fourth Conference on Partitioned Global Address Space Programming Model*, page 2. ACM, 2010.

[11] B. Chapman, D. Eachempati, and O. Hernandez. Experiences developing the openuh compiler and runtime infrastructure. *International Journal of Parallel Programming*, 41(6):825–854, 2013.

[12] P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. Von Praun, and V. Sarkar. X10: an object-oriented approach to non-uniform cluster computing. *Acm Sigplan Notices*, 40(10):519–538, 2005.

[13] C. Coarfa, Y. Dotsenko, J. Eckhardt, and J. Mellor-Crummey. Co-array fortran performance and potential: An npb experimental study. In *Languages and Compilers for Parallel Computing*, pages 177–193. Springer, 2003.

[14] F. S. Committee. TS 18508 Additional Parallel Features in Fortran (WG5/N2074). Technical report, August 2015.

[15] J. Diaz, C. Munoz-Caro, and A. Nino. A survey of parallel programming models and tools in the multi and many-core era. *Parallel and Distributed Systems, IEEE Transactions on*, 23(8):1369–1386, 2012.

[16] Y. Dotsenko, C. Coarfa, and J. Mellor-Crummey. A multi-platform co-array fortran compiler. In *Proceedings of the 13th International Conference on Parallel Architectures and Compilation Techniques*, pages 29–40. IEEE Computer Society, 2004.

[17] D. Eachempati, H. J. Jun, and B. Chapman. An open-source compiler and runtime implementation for Coarray Fortran. In *Proceedings of the Fourth Conference on Partitioned Global Address Space Programming Model*, PGAS '10, pages 13:1–13:8, New York, NY, USA, 2010. ACM.

[18] D. Eachempati, A. Richardson, S. Jana, T. Liao, H. Calandra, and B. Chapman. A Coarray Fortran implementation to support data-intensive application development. *Cluster Computing*, 17(2):569–583, 2014.

[19] A. Fanfarillo, T. Burnus, V. Cardellini, S. Filippone, D. Nagle, and D. Rouson. Opencoarrays: open-source transport layers supporting coarray fortran compilers. In *Proceedings of the 8th International Conference on Partitioned Global Address Space Programming Models*, page 4. ACM, 2014.

[20] S. Ge, D. Eachempati, D. Khaldi, and B. Chapman. An evaluation of anticipated extensions for fortran coarrays. In *Partitioned Global Address Space Programming Models (PGAS), 2015 9th International Conference on*, pages 47–58. IEEE, 2015.

[21] W. Gropp, E. Lusk, N. Doss, and A. Skjellum. A high-performance, portable implementation of the mpi message passing interface standard. *Parallel computing*, 22(6):789–828, 1996.

[22] W. Gropp, E. Lusk, and A. Skjellum. *Using MPI: portable parallel programming with the message-passing interface*, volume 1. MIT press, 1999.

[23] J. Guohua, J. Mellor-Crummey, L. Adhianto, W. Scherer, and C. Yang. Implementation and Performance Evaluation of the HPC Challenge Benchmarks in Coarray Fortran 2.0. In *Parallel Distributed Processing Symposium (IPDPS), 2011 IEEE International*, pages 1089–1100, May 2011.

[24] M. Gupta, S. Midkiff, E. Schonberg, V. Seshadri, D. Shields, K.-Y. Wang, W.-M. Ching, and T. Ngo. An hpf compiler for the ibm sp2. In *Supercomputing, 1995. Proceedings of the IEEE/ACM SC95 Conference*, pages 71–71. IEEE, 1995.

[25] J. C. Hardwick. Practical parallel divide-and-conquer algorithms. Technical report, DTIC Document, 1997.

[26] P. N. Hilfinger, D. O. Bonachea, K. Datta, D. Gay, S. L. Graham, B. R. Liblit, G. Pike, J. Z. Su, and K. A. Yelick. Titanium language reference manual, version 2.19. Technical report, UC Berkeley Tech Rep. UCB/EECS-2005-15, 2005.

[27] D. S. Hirschberg. Fast parallel sorting algorithms. *Communications of the ACM*, 21(8):657–661, 1978.

[28] T. Hoefler and T. Schneider. Optimization Principles for Collective Neighborhood Communications. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '12, Los Alamitos, CA, USA, 2012. IEEE Computer Society Press.

[29] L. Huang, H. Jin, L. Yi, and B. Chapman. Enabling locality-aware computations in openmp. *Scientific Programming*, 18(3-4):169–181, 2010.

[30] L. Huang, G. Sethuraman, and B. Chapman. Parallel data flow analysis for openmp programs. In *A Practical Programming Model for the Multi-Core Era*, pages 138–142. Springer, 2007.

[31] A. Kamil and K. Yelick. Hierarchical additions to the spmd programming model. Technical report, Citeseer, 2012.

[32] A. A. Kamil. A Team Analysis Proposal for Recursive Single Program, Multiple Data Programs. Technical Report UCB/EECS-2012-183, EECS Department, University of California, Berkeley, Aug 2012.

[33] H. Kasim, V. March, R. Zhang, and S. See. Survey on parallel programming model. In *Network and Parallel Computing*, pages 266–275. Springer, 2008.

[34] K. Kennedy, C. Koelbel, and H. Zima. The rise and fall of high performance fortran. *Commun. ACM*, 54(11):74–82, Nov. 2011.

[35] D. Khaldi, D. Eachempati, S. Ge, P. Jouvelot, and B. Chapman. A team-based methodology of memory hierarchy-aware runtime support in coarray fortran (accepted). In *IEEE International Conference on Cluster Computing (IEEE CLUSTER)*, Sept 2015.

[36] D. Lea. A memory allocator called doug leas malloc or dlmalloc for short. *Available Online: April*, 18, 2010.

[37] C. Liao, O. Hernandez, B. Chapman, W. Chen, and W. Zheng. Openuh: An optimizing, portable openmp compiler. *Concurrency and Computation: Practice and Experience*, 19(18):2317–2332, 2007.

[38] D. B. Loveman. High performance fortran. *Parallel & Distributed Technology: Systems & Applications, IEEE*, 1(1):25–42, 1993.

[39] J. Mellor-Crummey, L. Adhianto, W. N. Scherer III, and G. Jin. A new vision for coarray fortran. In *Proceedings of the Third Conference on Partitioned Global Address Space Programing Models*, page 5. ACM, 2009.

[40] M. Metcalf. Fortran 90 and its successors. In *Encyclopedia of Parallel Computing*, pages 711–718. Springer, 2011.

[41] J. Nieplocha, R. J. Harrison, and R. J. Littlefield. Global arrays: a portable shared-memory programming model for distributed memory computers. In *Proceedings of the 1994 ACM/IEEE conference on Supercomputing*, pages 340–349. IEEE Computer Society Press, 1994.

[42] R. W. Numrich and J. Reid. Co-array fortran for parallel programming. In *ACM Sigplan Fortran Forum*, volume 17, pages 1–31. ACM, 1998.

[43] A. Petitet, R. C. Whaley, J. Dongarra, and A. Cleary. HPL - A Portable Implementation of the High-Performance Linpack Benchmark for Distributed-Memory Computers.

[44] T. Stitt. *An introduction to the Partitioned Global Address Space (PGAS) programming model*. Connexions, Rice University, 2009.

[45] R. Thakur, R. Rabenseifner, and W. Gropp. Optimization of collective communication operations in mpich. *International Journal of High Performance Computing Applications*, 19(1):49–66, 2005.

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

[47] V. Tipparaju, W. Gropp, H. Ritzdorf, R. Thakur, and J. L. Träff. Investigating high performance rma interfaces for the mpi-3 standard. In *Parallel Processing, 2009. ICPP'09. International Conference on*, pages 293–300. IEEE, 2009.

[48] A. Welch, S. Pophale, P. Shamis, O. Hernandez, S. Poole, and B. Chapman. Extending the OpenSHMEM Memory Model to Support User-Defined Spaces. *PGAS 2014*, oct 2014.

[49] Wikipedia. Message passing interface — Wikipedia, the free encyclopedia, 2004. [Online; accessed 20-March-2016].

[50] C. Yang, W. Bland, J. Mellor-Crummey, and P. Balaji. Portable, mpi-interoperable coarray fortran. In *ACM SIGPLAN Notices*, volume 49, pages 81–92. ACM, 2014.

[51] K. Yelick, D. Bonachea, W.-Y. Chen, P. Colella, K. Datta, J. Duell, S. L. Graham, P. Hargrove, P. Hilfinger, P. Husbands, et al. Productivity and performance using partitioned global address space languages. In *Proceedings of the 2007 international workshop on Parallel symbolic computation*, pages 24–32. ACM, 2007.