RICE UNIVERSITY

Expressiveness, Programmability and Portable High Performance of Global Address Space Languages

by

Yuri Dotsenko

A THESIS SUBMITTED
IN PARTIAL FULFILLMENT OF THE
REQUIREMENTS FOR THE DEGREE

Doctor of Philosophy

APPROVED, THESIS COMMITTEE:

Dr. John Mellor-Crummey, Chair
Associate Professor of Computer Science

Dr. Ken Kennedy
John and Ann Doerr University Professor
of Computational Engineering

Dr. Peter Joseph Varman,
Professor of Electrical & Computer
Engineering

HOUSTON, TEXAS

JANUARY, 2007

Expressiveness, Programmability and Portable High Performance of Global Address Space Languages

Yuri Dotsenko

Abstract

The Message Passing Interface (MPI) is the library-based programming model employed by most scalable parallel applications today; however, it is not easy to use. To simplify program development, Partitioned Global Address Space (PGAS) languages have emerged as promising alternatives to MPI. Co-array Fortran (CAF), Titanium, and Unified Parallel C are explicitly parallel single-program multiple-data languages that provide the abstraction of a global shared memory and enable programmers to use one-sided communication to access remote data. This thesis focuses on evaluating PGAS languages and explores new language features to simplify the development of high performance programs in CAF.

To simplify program development, we explore extending CAF with abstractions for group, Cartesian, and graph communication topologies that we call co-spaces. The combination of co-spaces, textual barriers, and single values enables effective analysis and optimization of CAF programs. We present an algorithm for synchronization strength reduction (SSR), which replaces textual barriers with faster point-to-point synchronization. This optimization is both difficult and error-prone for developers to perform manually. SSR-optimized versions of Jacobi iteration and the NAS MG and CG benchmarks yield performance similar to that of our best hand-optimized variants and demonstrate significant improvement over their barrier-based counterparts.

To simplify the development of codes that rely on producer-consumer communication, we explore extending CAF with multi-version variables (MVVs). MVVs increase programmer productivity by insulating application developers from the details of buffer man-

agement, communication, and synchronization. Sweep3D, NAS BT, and NAS SP codes expressed using MVVs are much simpler than the fastest hand-coded variants, and experiments show that they yield similar performance.

To avoid exposing latency in distributed memory systems, we explore extending CAF with distributed multithreading (DMT) based on the concept of function shipping. Function shipping facilitates co-locating computation with data as well as executing several asynchronous activities in the remote and local memory. DMT uses co-subroutines/co-functions to ship computation with either blocking or non-blocking semantics. A prototype implementation and experiments show that DMT simplifies development of parallel search algorithms and the performance of DMT-based RandomAccess exceeds that of the reference MPI implementation.

Acknowledgments

I would like to thank my adviser, John Mellor-Crummey, for his inspiration, technical direction, and material support. I want to thank my other committee members, Ken Kennedy and Peter Varman, for their insightful comments and discussions. I am indebted to Tim Harvey for his help with data-flow analysis. I am grateful to Luay Nakhleh, Keith Cooper, and Daniel Chavarría-Miranda who provided guidance and advice.

I want to thank my colleague, Cristian Coarfa, for years of productive collaboration. I would like to thank Fengmei Zhao, Nathan Tallent, and Jason Eckhardt for their work on the Open64 infrastructure. Jarek Nieplocha and Vinod Tipparaju provided invaluable help with the ARMCI library. Kathy Yelick, Dan Bonachea, Parry Husbands, Christian Bell, Wei Chen, and Costin Iancu provided assistance with the GASNet library. Craig Rasmussen helped to reverse-engineer the dope vector format of several Fortran 95 vendor compilers. Collaboration with Tarek El-Ghazawi, Francois Cantonnet, Ashrujit Mohanti, and Yiyi Yao resulted in a successful joint publication.

I would like to thank many people at Rice who helped me during my stay. Tim Harvey, Bill Scherer, and Charles Koelbel helped in revising and improving the manuscript. Daniel Chavarría-Miranda and Tim Harvey provided invaluable assistance with the qualification examination. I want to thank Robert Fowler, Yuan Zhao, Apan Qasem, Alex Grosul, Zoran Budimlic, Nathan Froyd, Arun Chauhan, Anirban Mandal, Guohua Jin, Cheryl McCosh, Rajarshi Bandyopadhyay, Anshuman Das Gupta, Todd Waterman, Mackale Joyner, Ajay Gulati, Rui Zhang, and John Garvin.

This dissertation is dedicated to my family, my dear wife Sofia, my son Nikolai, my sister, and my parents for their never-ending love, infinite patience, and support. Sofia also contributed to this work by drawing some of the illustrations.

Contents

Illustrations

Tables

Chapter 1

Introduction

Modern scientific progress heavily relies on computer simulations that are becoming more complex, memory demanding, and computation hungry. However, the computation power of individual processor cores is limited by clock frequency. To satisfy the demand, computer manufacturers have shifted their focus towards more parallel hardware. Not only high-end super computers but also personal desktops and laptops are becoming multiprocessor and multi-core. Unfortunately, software development practices and tools for parallel and concurrent computing are lagging behind. For parallel programming to flourish, it will require a programming model that is *ubiquitous*, *expressive*, and *easy to use* while also providing transparent *performance portability*.

Today, the *de facto* standard for programming scalable parallel systems is the Message Passing Interface (MPI) [62]. MPI is a low-level library-based parallel programming model based on two-sided communication that is implemented on almost every parallel platform. In parallel programs based on MPI, application developers have full control over performance critical decisions such as data decomposition, computation partitioning, and communication placement. While MPI is a powerful instrument in the hands of an experienced programmer, most developers have found that it is difficult and error-prone to write parallel programs using the MPI model. Due to the library-based nature of MPI communication, MPI programs are not well-suited to compiler-based improvement, which leaves application developers solely responsible for choreographing communication and computation to achieve high performance.

There has been significant interest in trying to improve the productivity of parallel programmers by either using automatic parallelization techniques, such as those found in the

Polaris compiler [14], or language-based parallel programming models that abstract away most of the complex details of library-based high performance communication. The two parallel programming models that have received attention from the scientific community are OpenMP [42] and High Performance Fortran (HPF) [77]. However, both of these models have significant shortcomings that reduce their utility for writing portable, scalable, high performance parallel programs. OpenMP programmers have little control over data layout; as a result, OpenMP programs are difficult to map efficiently to distributed memory platforms. In contrast, HPF enables programmers to explicitly control the mapping of data to processors; however, to date, commercial HPF compilers have failed to deliver high performance for a broad range of programs. Experience with early HPF compilers has shown that in the absence of very capable parallelizing compilers, it is crucial to provide programmers with sufficient control to enable them to employ sophisticated parallelizations by hand.

The family of partitioned global address space (PGAS) languages, including Co-array Fortran (CAF) [86], Unified Parallel C (UPC) [121], and Titanium [66], has attracted interest as a promising alternative to MPI because it offers the illusion of shared memory. CAF, UPC, and Titanium employ the single-program-multiple-data (SPMD) model for parallel programming and are simple extensions to widely-used languages, Fortran 95, C, and Java, respectively. The global address space abstraction of these languages naturally supports a one-sided communication style, considered easier and more convenient to use than MPI's two-sided message passing. With communication and synchronization as part of the language, programs written in these languages are more amenable to compiler-directed communication optimization than MPI's library-based communication; however, in PGAS languages, programmers retain full control over critical decisions necessary to achieve high performance.

The goal of this thesis is to evaluate the PGAS programming model to identify limitations and find ways to address them by programming model improvement, compiler optimization, and sophisticated run-time engineering to equip application developers with an easier to use, more expressive and ubiquitously available programming model that de-

livers performance comparable to that of hand-tuned codes for a variety of applications on a broad range of modern parallel architectures.

1.1 Thesis overview

This research mainly explores Co-array Fortran, a member of the PGAS family of languages. CAF is based on a small set of extensions to Fortran 95. CAF is of special interest for the scientific community because many legacy codes are written in Fortran and many parallel high performance codes are developed using Fortran and MPI. To inspire the community to incrementally port these codes into CAF and to develop new CAF applications, Co-array Fortran must be easy to use and deliver high performance on a range of parallel platforms.

CAF was designed by Cray for tightly-coupled architectures with globally addressable memory featuring low communication latency and high communication bandwidth. The performance results for a CAF version of the NAS MG benchmark [12] were promising on the Cray T3E [22]. However, it was not clear whether efficient CAF implementations could be engineered for a range of architectures including shared-memory, cluster, and hybrid platforms to deliver high performance for a broad spectrum of applications.

The research described in this dissertation was performed in two parts. The first part was joint work with Cristian Coarfa on the design and implementation of `cafc`, a research CAF compiler for distributed- and shared-memory systems, and several evaluation studies [30, 47, 48, 31, 32, 33] to investigate the quality of the CAF programming model and its ability to deliver high performance. Using numerous parallel applications and benchmarks, we showed that the performance of `cafc`-compiled codes matches that of their hand-tuned MPI counterparts on a range of parallel architectures. However, developing high performance programs using classical CAF [87, 86] today is as difficult and error-prone as writing the equivalent MPI codes. The second part was independent research focused on exploring enhancements the CAF programming model to simplify the development of high performance codes in CAF.

My thesis is that *extending CAF with language-level communication topologies, multi-version variables, and distributed multithreading will increase programmers' productivity by simplifying the development of high performance codes*. In particular,

- Extending CAF with communication topologies will equip programmers with commonly used abstractions for organizing program processes, facilitate compiler communication analysis, and support more efficient collective communication.

- Using communication topologies (in the form of co-spaces) and single-valued expressions enables a CAF compiler to perform conversion of textual barriers into faster point-to-point synchronization; this relieves the programmer of the burden of orchestrating complex synchronization while delivering the level of performance comparable to that of hand-optimized codes.

- Enhancing CAF with multi-version variables simplifies the development of wavefront and other producer-consumer applications by insulating the programmer from the details of buffer management and synchronization.

- Adding distributed multithreading to CAF enables computation to be co-located with data to avoid exposing communication latency, simplifies access to remote complex data structures, and enables several asynchronous activities in the remote and local memory.

1.2 Contributions of joint work

This section briefly summarizes the contributions of joint work with Cristian Coarfa.

First, we provide a brief overview of CAF language constructs. An executing CAF program consists of a fixed number of asynchronous process images. The images use co-arrays to access distributed data. For example, `integer::a(n,m)[*]` declares a shared co-array `a` with `n` $\times$ `m` integers local to each process image. The dimensions inside brackets are called co-dimensions. Co-arrays may be declared for user-defined or primitive types. A

local section of a co-array may be a singleton instance of a type rather than an array of type instances. Remote sections of a co-array can be accessed by using the bracket notation. For example, process `p` can access the first column of co-array `a` from process `p+1` by referencing `a(:,1)[p+1]`. A remote co-array access induces one-sided communication in the sense that only one image knows about the access; the target image is not aware of the communication. A remote co-array assignment translates into a one-sided remote write (PUT). A remote co-array reference translates into a one-sided remote read (GET).

We designed and implemented a Co-array Fortran compiler, `cafc`, that supports most of the original CAF language specification [87]. It is the first multiplatform open-source CAF compiler. It is a source-to-source translator, based on the OPEN64/SL [101] infrastructure, that transforms a CAF program into a Fortran 95 program augmented with communication and synchronization code tailored to the target architecture.

We ported several parallel benchmarks and applications into CAF and used `cafc` to compile them. We performed extensive evaluation [30, 47, 48, 31, 32] of these codes to identify the underlying causes of inefficiencies and performance bottlenecks on a range of modern parallel architectures. An important result is demonstration that a broad variety of CAF codes *can* match, and sometimes exceed, the performance of equivalent hand-tuned MPI variants on a range of shared-memory and cluster platforms.

We performed a thorough comparison of the CAF and UPC programming models for several benchmarks that perform computations on multi-dimensional arrays [33]. It revealed that it is harder to match the performance of MPI codes with UPC. The main reason is that UPC uses C as the target sequential language, which lacks language support for multi-dimensional arrays.

We identified performance bottlenecks that kept CAF programs from matching the MPI's performance, fixed some of them, and suggested coding recipes to alleviate the others until `cafc`'s infrastructure matures.

The `cafc` compiler uses source-to-source translation to leverage each target platform's best Fortran 95 compiler to optimize sequential Fortran 95 program. The translation pro-

cess must not inhibit the ability of the target Fortran 95 compiler to generate high performance code. We investigated different co-array representations for local and remote co-array accesses across a range of architectures and back-end compilers. The result of our study [48] is that it is acceptable to represent co-arrays as Fortran 95 pointers. However, the translation process might not convey the shapes and lack of aliasing for static co-arrays to the target platform's Fortran 95 compiler, resulting in suboptimal scalar performance. We devised a technique, called procedure splitting [47], that represents static co-arrays as procedure parameters conveying static co-array shapes and lack of aliasing to the target platform's Fortran 95 compiler. This enabled `cafc`-translated sequential program to have scalar performance similar to that of an equivalent Fortran 95 program.

Lack of efficient communication in parallel programs hinders performance and scalability. On cluster architectures, communication vectorization and communication aggregation are essential to increase the granularity of communication. An advantage of CAF over other languages is that communication vectorization can be conveniently expressed in the source code using Fortran 95 triplet notations, *e.g.*, `a(1,:)[p]`. However, array sections not contiguous in memory lead to strided communication that is not supported efficiently by the existing communication libraries. Programmers should use contiguous temporary buffers and packing/unpacking of communicated data to yield top performance [47]. We also investigated the possibility of enabling non-blocking communication using hints [47].

In CAF, several process images can access the same shared data. Programmers must use explicit synchronization to ensure the correct order of such accesses. CAF provides global barrier and team synchronization. We observed that using barrier-based synchronization is simpler, but results in suboptimal performance and poor scalability. As expected, point-to-point synchronization between a pair of images is faster and yields higher performance and scalability [47]; however, this performance comes at the cost of greater programming complexity. Using several co-arrays for communication together with point-to-point synchronization might reduce the number of synchronization messages and/or remove them from the critical path [31, 32].

Chapter 4 provides additional details and results of our joint studies.

1.3 Research contributions

The second part of this dissertation explores extensions to the CAF programming model that simplify the development of high performance parallel applications by either providing missing language features or enabling better compiler analysis. We explore extending CAF with language-level communication topologies called co-spaces, multi-version variables, and distributed multithreading. The combination of co-spaces, textual co-space barriers, and co-space single-valued expressions enables synchronization strength reduction, which helps a CAF compiler to replace textual barriers with faster point-to-point synchronization.

1.3.1 CAF communication topologies – co-spaces

CAF differs from the other PGAS languages in that programmers explicitly specify the target image of a remote co-array access. For instance, `a[p]` references the portion of co-array data located in image `p`'s memory. A co-array can have several co-dimensions and its declaration defines a co-shape, *e.g.*, `integer a[2,*]` arranges process images into a 2×3 Cartesian grid without periodic boundaries when the number of process images is 6. However, co-shape has several disadvantages described in detail in Section 5.1. The topology can be incompletely filled if the number of images is not divisible by the product of co-shape dimensions. Co-shape provides only one type of communication topology for all process images forcing programmers to re-implement topologies of other types that are more suitable for the application needs. In turn, this not only imposes a burden on the programmers but also complicates compiler analysis.

We explore replacing the notion of co-shape in CAF with support for *group*, *Cartesian*, and *graph* communication topologies, based on the ideas of MPI communicators and process topologies [62, 112]. In CAF, an instance of a communication topology is called a *co-space*. Co-spaces provide reusable abstractions for organizing application's images into groups with Cartesian or graph communication topologies and specifying the target of a

remote co-array access. Each co-space has a set of interface functions that enable programmers to specify communication targets in a systematic way. In addition, a CAF compiler can use these functions to symbolically analyze a number of common communication patterns, which enables several powerful communication and synchronization optimizations.

1.3.2 Synchronization strength reduction

We present an algorithm for one such optimization, called synchronization strength reduction (SSR). Where legal, SSR tries to replace barriers with faster point-to-point synchronization. To do so successfully, it requires *textual co-space barriers* and *co-space single values* described in detail in Chapter 6, which are based on similar concepts from Titanium [66, 75]. A textual barrier guarantees that images execute the same barrier statement. A single-valued expression evaluates to the same value on a group of images. Single-valued expressions, used in control statements such as `IF-THEN-ELSE`, enable the compiler to reason about the control flow of a group of images. We explore extending CAF with textual co-space barriers and co-space single-value hints to enable SSR. We focus on optimizing common communication patterns. For instance, when every image accesses a co-array on its left neighbor in a Cartesian topology. If such a statement is executed by images of a co-space and the target image is specified via co-space interface functions with co-space single-valued arguments, a CAF compiler would be able to analyze the communication pattern. Such analysis enables every co-space image to determine the origin(s) of one-sided communication locally, without the need to contact other images. Using this knowledge, in certain cases a CAF compiler might be able to replace textual barriers and one-sided communication with two-sided communication.

SSR works within procedure scopes that use only structured control flow in the form of `IF-THEN-ELSE` statements and `DO` loops. Without inter-procedural communication analysis or automatic procedure inlining, its applicability is limited. We explore a set of SSR compiler directives to overcome this limitation until inter-procedural analysis is available. We implemented prototype support for synchronization analysis and SSR in `cafc`.

Several SSR-optimized CAF codes annotated with the SSR directives show performance comparable to that of hand-coded versions that use point-to-point synchronization and are noticeably faster than their barrier-based counterparts. The details of CAF program analysis and SSR can be found in Chapters 6 and 7.

1.3.3 Multi-version variables

PGAS languages are concurrent in that several threads of execution can access the same shared variable simultaneously either locally or using one-sided communication. The programmer is responsible for synchronization of these threads. A typical communication pattern found in wave-front and producer-consumer applications is sending a stream of values from one processor to another. The processor that generates and sends values is called the producer; the processor that receives the values is called the consumer. CAF implementation of such a producer-consumer pattern that yields high performance is not trivial for a distributed memory machine [31, 32]. Images transmit data using co-arrays as communication buffers. If a consumer uses only one communication buffer to accept data, the producer has to wait for the consumer to finish using the buffer before sending a new value. This results in an algorithm that is not asynchrony-tolerant. To obtain high performance, the programmer has to manage several communication buffers and use point-to-point synchronization [31, 32]. In fact, developing such codes using MPI's two-sided buffered communication is easier.

We explore extending CAF with *multi-version* variables (MVVs) to simplify development of wave-front and other producer-consumer codes. An MVV can store more than one value. A sequence of values is managed with the semantics of a stream: the producer commits new values and the consumer retrieves them in the same order. MVVs increase programmer productivity by insulating the programmer from the details of buffer management and synchronization. Producer-consumer codes expressed via MVVs are cleaner and simpler than their hand-optimized counterparts, and experiments show that they deliver similar performance.

1.3.4 Distributed multithreading

CAF provides co-arrays to efficiently access remote array and scalar data. However compared to the other PGAS languages, it fails to provide efficient language constructs and semantics for accessing complex data structures such as lists and trees located in *remote* memory. We explore extending CAF with *distributed multithreading* (DMT) to avoid exposing communication latency. DMT is based on the concept of function shipping, which facilitates co-locating computation with data as well as enables several asynchronous activities in the remote and local memory. DMT uses *co-subroutines* and *co-functions*, by analogy with co-arrays, to spawn a new thread of computation locally or remotely. We explore the impact of DMT on the CAF execution model and semantics of co-subroutine parameter passing. A prototype implementation and experiments showed that DMT simplifies development of parallel search algorithms without a dedicated master; and DMT-based RandomAccess [1] code, which performs asynchronous random updates of a huge distributed table, exceeds that of the standard MPI version for a medium-size cluster.

1.4 Thesis outline

Chapter 2 reviews several approaches to parallel programming and, where applicable, compares and contrasts them with our solutions. Chapter 3 provides background information necessary for understanding our approaches and methodologies in Chapters 4–9. Chapter 4 describes interesting details of work with Cristian Coarfa on `cafc` engineering and performance evaluation studies. Chapters 5, 6 and 7 present co-spaces, program analysis and SSR. Chapters 8 and 9 focus on multi-version variables and distributed multithreading. Finally, Chapter 10 summarizes the results of our research and discusses promising future research directions.

Chapter 2

Related Work

This chapter starts with an overview of existing Co-array Fortran compilers. Next, it summarizes several programming models and parallel programming paradigms with the emphasis on features and optimization techniques related to those addressed in this dissertation. Where appropriate, our ideas are compared and contrasted with the existing approaches.

2.1 CAF compilers

As of this writing, there exist only two CAF compiler implementations: one is available on Cray X1E [37], X1 [36], and T3E [109] architectures, the other is a multi-platform CAF compiler developed at Rice University [30, 47]. The Cray CAF compiler is available only for Cray architectures that provide globally accessible memory, where each processor can access memory of other processors through a high-bandwidth, low-latency memory subsystem. These architectures are perfect for CAF. In fact, the original CAF specification was influenced by the assumption of "good" hardware. To the best of our knowledge, there is no publication about the design, engineering, and optimizations of the Cray CAF compiler. However, vector registers on X1 & X1E and E-registers on T3E enable access to remote co-array data directly, without the need to allocate a temporary in local memory to store off-processor data for the duration of a computation. This enables streaming remote data directly into a local computation. This approach requires minor modifications to the existing Fortran compiler, but it cannot be employed for architectures that lack globally accessible memory. Several studies [22, 91] showed that the performance of several CAF codes (NAS MG and LBMHD) compiled with the Cray CAF compiler exceeded that of

the equivalent MPI versions on these architectures for two reasons. First, PUT/GET can be translated into lightweight hardware operations, which is cheaper than performing a procedure call for MPI send/receive. Second, MPI may use extra memory copies to perform two-sided communication, increasing cache pressure.

The other CAF compiler is `cafc` developed at Rice University. `cafc` is a *multi-platform*, open-source compiler that, in contrast to the Cray CAF compiler, generates code for a range of cluster and shared-memory architectures. `cafc` is a source-to-source translator. It transforms a CAF program into an equivalent Fortran 95 program augmented with communication and synchronization code. To accommodate cluster architectures, `cafc` uses intermediate temporaries to hold off-processor data for the duration of a computation and utilizes the ARMCI [83] or GASNet [17] library to perform PUT/GET. We showed that the performance of a broad variety of CAF codes compiled with `cafc` can match that of their MPI counterparts. We do not have an implementation of `cafc` for Cray architectures and did not perform a comparison study. We present a detailed description of `cafc` design choices and engineering effort in Chapter 4.

2.2 Data-parallel and task-parallel languages

In this section, we provide a detailed overview of several data-parallel and task-parallel models such as High-Performance Fortran (HPF) [52, 77], OpenMP [42], UC [11], and SUIF system [3] for automatic parallelization. We first discuss how the analysis and optimization of Partitioned Global Address Space languages (PGAS), which include Co-array Fortran (CAF), Unified Parallel C (UPC), and Titanium, are different from those of data-parallel and task-parallel languages.

The intent of data-parallel and task-parallel programming languages is to simplify parallel programming by providing high-level abstractions for expressing parallelism. However, mapping these abstractions onto parallel architectures efficiently is a difficult task. In these models, programmers do not have full control over all performance critical parallelization decisions such as data decomposition, computation partitioning, data movement,

and synchronization. They must rely on compilers to deliver high performance.

In general, compiler implementations of task- and data-parallel languages have not been able to deliver high performance for a variety of applications on a wide range of parallel architectures, especially for distributed-memory platforms. As a consequence, these programming models have not received general acceptance. However, reliance on the compiler to deliver performance has spurred development of many compiler analysis techniques and communication/synchronization optimizations.

Unfortunately, not all technology for analysis and optimization of data- and task-parallel languages can readily be adopted for analysis and optimization of PGAS languages. CAF, UPC, and Titanium are *explicitly-parallel* single-program multiple-data (SPMD) languages. They can benefit from traditional analysis and optimization techniques, such as scalar optimizations and communication vectorization/aggregation, that rely on control flow and values of a *single* process of an SPMD parallel program. However, new compiler technology is necessary to relate control-flow and values of several SPMD processes.

In data- and task-parallel languages, the compiler deals with *structured* parallelism expressed via sequential program statements, data-distribution directives, or special parallel execution statements/directives (*e.g.*, SPMD regions or parallel loops). The programming model exposes the structure of parallelism to the compiler that often can analyze and "understands" this structure. The analysis of explicitly-parallel languages such as CAF is different. A programmer creates an arbitrary parallel program and parallelism is defined by the program's semantics. In this respect, the parallelism is "unstructured" and the compiler must *infer* its structure to analyze and optimize communication and synchronization. One contribution of this work is a technique for imposing computation structure on CAF programs that both simplifies program development and enables analysis and optimization of communication/synchronization.

We now describe several data-parallel and task-parallel programming models and approaches to their analysis and optimizations in more detail.

2.2.1 High-Performance Fortran

High Performance Fortran (HPF) [52, 77] is a data-parallel language. To use HPF, a programmer annotates a sequential Fortran program with data-distribution directives. For distributed-memory systems, an HPF compiler transforms this program into a parallel SPMD program, in which the data is distributed across the nodes of the system. HPF compilers use mathematical representation, expressed as functions or sets, for data elements owned by each processor. These sets are used to determine computation partitioning guided by the *owner-computes rule* [8] — the owner of the left-hand side of each assignment must compute the right-hand side expression. Analysis of subscripted references is used to determine off-processor data necessary for computation. The analysis starts with a sequential program and the compiler can leverage traditional analysis and optimization technology. Since the compiler is solely responsible for transforming the sequential program into an SPMD program, it "understands" the computation structure, *e.g.*, global control flow, and can generate efficient two-sided communication.

There are several implementations of HPF compilers that are able to deliver good performance primarily for regular, dense scientific codes on several architectures. Chavarría's thesis [24] and his joint work with Mellor-Crummey [26, 25] showed that, using the dHPF compiler, it is possible to match the performance of MPI for regular, dense scientific codes on several architectures. A notable feature of dHPF is support for generalized multipartitioning [27, 43]. Gupta *et al.* [64] discuss the design, implementation, and evaluation of the pHPF compiler done at IBM Research; they show good speedups for several regular benchmarks. PGHPF [115, 19] is a commercial HPF compiler from PGI. Both pHPF and PGHPF have limited support for communication optimization of loops with carried dependence along distributed dimensions. Sakagami *et al.* [105] showed that IMPACT-3D plasma simulation code compiled with HPF/ES [120] achieved 45% of the peak performance when running on 512 nodes on the Earth Simulator [51].

While HPF improves programmability and can deliver high performance for regular scientific applications, it has two disadvantages that prevented HPF from achieving

widespread acceptance. First, programmers have little control over the parallelization process; *i.e.*, if an HPF compiler makes a wrong parallelization decision, it is very hard for the programmer to intervene and undo the "harm", and any intervention requires immense knowledge about the compiler internals. As a consequence, there is not enough evidence that HPF can deliver performance for a broad class of applications. Second, a good HPF compiler implementation requires heroic effort, which makes HPF less appealing as a pragmatic programming model. In contrast, CAF sacrifices programmability, but allows programmers to retain much more control over performance critical decisions to obtain the same level of performance as that of hand-optimized MPI codes. In addition, the effort to implement a CAF compiler is modest; *e.g.*, refer to `cafc` engineering details in Chapter 4.

2.2.2 OpenMP

OpenMP [42] is a task-parallel directive-based programming model that offers a fork-join model for parallelism with a focus on loop parallelization. In comparison to the CAF, UPC, and Titanium languages, OpenMP provides no means to the programmer for controlling the distribution of data among processors. As a result, it is very hard to map efficiently onto a distributed-memory architecture. There are two major approaches to optimize OpenMP for a cluster architecture. The first approach is to use data-distribution directives. For example, Chapman *et al.* [23] proposes a set of data-distribution directives, based on similar features of HPF, to enable programmers to control data locality in OpenMP. However, this complicates OpenMP as a programming model and requires similar compiler technology as for HPF to deliver high performance, which is hard for a broad class of applications. As of this writing, the OpenMP specification [16] does not have data-distribution directives. The second approach is based on clever engineering of the runtime layer to exploit data locality, perhaps, with the help of the compiler. Nikolopoulos *et al.* [85] describes and evaluates a mechanism for data-distribution and redistribution in OpenMP without programmer intervention. The approach is effective for fixing poor initial page placement on a coherent-cache non-uniform memory access architecture (ccNUMA). Hu *et al.* [68] de-

scribes and evaluates an implementation of OpenMP that uses the TreadMarks software distributed shared-memory (SDSM) system [76]. Their experiments show that speedups of multithreaded TreadMarks programs are within 7–30% of the MPI versions for several benchmarks on 16 processors (an IBM SP2 cluster of four four-processor SMPs). Cluster OpenMP [70, 67] is a commercial implementation of OpenMP for clusters based on SDSM. Hoeflinger [67] compares speedups of several applications run on an Itanium2-based cluster using Cluster OpenMP and on an Itanium2-based hardware shared-memory machine using OpenMP. The results show that it is possible to achieve a good percentage of the performance of a hardware shared-memory machine on a cluster by using Cluster OpenMP.

While some codes parallelized using OpenMP can achieve good performance on small-scale SMPs and even clusters, in our study [48], we observed that for other codes OpenMP does not scale well even for non-uniform memory access (NUMA) shared-memory architectures such as SGI Altix [111]. In general, it is hard to efficiently parallelize codes that use multi-dimensional arrays [126]. Using OpenMP might be a good parallelization strategy for applications that have high data locality (*e.g.*, primarily stride one accesses) and little fine-grain synchronization. The recent shift towards multi-core multiprocessor architectures might increase the significance of OpenMP as a programming model to achieve performance on multi-core multiprocessor nodes. However, OpenMP (especially, without data-distribution directives) is unlikely to deliver good performance on large-scale cluster architectures. OpenMP can be used together with CAF to exploit parallelism available within a multi-core multiprocessor node. In this combination, CAF provides data locality and inter-node parallelism, while OpenMP allows to parallelize code accessing only local data to exploit intra-node parallelism.

2.2.3 UC

UC [11] uses the *index-set* data-type and `par` keyword to explicitly specify parallel execution of statements, which is more suited to shared-memory machines. `par` specifies

a well-defined parallel region "understandable" by the compiler. The compiler performs data mapping according to built-in heuristics and can permute data dynamically, if necessary [95]. The programmer can also provide a hint how the compiler should map the data. The compiler computes a synchronization graph for each `par` region based on the sequential data dependence graph. Renaming (extra storage) and alignment are used to reduce the number of interprocessor dependencies. Other dependencies must be preserved by using synchronization.

Prakash *et al.* [95] devised a set of techniques to reduce the number of barriers and/or to replace barriers with cheaper clustered synchronization in data parallel languages. They demonstrate them for UC [11], which has the `par` construct to specify a parallel region. They use a greedy algorithm to minimize the number of barriers necessary to preserve the dependencies in the `par` region. They eliminate barriers by breaking each analyzable data dependency (subscripts can be inverted at compile time) with send/receive communication and a temporary to store the result of the send. In addition, they use run-time techniques such as fuzzy barriers [65] and non-blocking send/receive to further optimize the program. Our SSR algorithm relies on the analysis of explicitly-parallel SPMD CAF programs to detect communication patterns via the interpretation of subscripts used to reference off-processor data.

2.2.4 Compiler-based parallelization

The SUIF [3] parallelizing compiler performs automatic parallelization of a sequential source program. Parallelism is created by the compiler and structured as a collection of fork-join SPMD regions synchronized with barriers. These barriers may lead to oversynchronized code and cause unnecessary overhead. Tseng [119] presents an algorithm for eliminating barriers or replacing them with counters by employing communication analysis developed for distributed memory machines [118]. Communication analysis determines how data flows between processors. If processors access only on-processor data in two adjacent SPMD regions, they do not communicate and no synchronization is required. Thus,

if communication analysis detects that the producers and consumers of all data shared between two SPMD regions are identical (the same processor), the barrier between these two regions can be eliminated. If inter-processor data movement is necessary, it may be possible to replace a barrier with counter-based point-to-point synchronization. Since the excessive use of counters is not efficient, the compiler uses them, one counter per a pair of processors, only for the cases of simple communication patterns such as nearest-neighbor, one-to-many, and many-to-one. These patterns are identified based on the system of linear inequalities corresponding to data movement between processors.

SUIF parallelizes a sequential (implicitly-parallel) program. CAF is explicitly-parallel with explicit data movement and synchronization. The novelty of our communication analysis is to use a combination of co-spaces, textual co-space barriers, and co-space single-valued expressions to infer communication patterns in a CAF program. If profitable, our synchronization strength reduction algorithm replaces barriers with more efficient point-to-point synchronization for the inferred patterns. We use notify and wait, which are similar to CAF's `notify`/`wait`, unidirectional point-to-point synchronization primitives per co-space processor group so that notify/wait of different co-spaces do not interfere. The implementation of co-space notify and wait conceptually uses pairwise counters between each pair of co-space processors. However, we suggest allocating a counter state on demand at runtime; *i.e.*, to create a real counter for a pair of processors iff point-to-point synchronization between them happens. This is necessary to reduce the memory overhead of having a counter for each pair of processors on a large-scale parallel machines such as Blue Gene/L [56].

2.3 PGAS programming models

There are three Partitioned Global Address Space (PGAS) parallel programming languages: Co-array Fortran, Unified Parallel C (UPC) and Titanium. They are based on Fortran, C, and Java, respectively. Each PGAS language extends its base language with a set of constructs to enable explicit SPMD parallel programming. There is no "best" PGAS language.

Each one offers some advantages and has some disadvantages. It is likely that the choice of the language would be determined by its base sequential language. In this respect, CAF has an advantage, because many high performance scientific codes that require parallelization are implemented in Fortran. In the following, we compare CAF with UPC and Titanium as well as review compiler analysis technology for Titanium.

2.3.1 Unified Parallel C

UPC [121] is an explicitly-parallel extension of ISO C that supports a global address space programming model for writing SPMD parallel programs. In the UPC model, SPMD threads share a part of their address space. The shared space is logically "flat". Physically, it is partitioned into fragments, each with a special association (affinity) to a given thread. UPC declarations give programmers control over the distribution of data across the threads; they enable a programmer to associate data with the thread primarily manipulating it. A thread and its associated data are typically mapped by the system into the same physical node. Being able to associate shared data with a thread makes it possible to exploit locality. In addition to shared data, UPC threads can have private data always co-located with its thread. UPC supports dynamic shared memory allocation. UPC provides the `upc_forall` work-sharing construct that distributes loop iterations according to the loop affinity expression that indicates which iterations to run on each thread. UPC adds several keywords to C that enable it to express a rich set of private and shared pointers. UPC has a memory model with relaxed and strict variables. Accesses to relaxed variables can be reordered for performance, while strict variables can be used for language-level synchronization, *e.g.*, point-to-point synchronization [33]. The language offers a range of synchronization constructs. Among the most interesting synchronization concepts in UPC is the split-phase fuzzy barrier [65], which enables overlapping local computation and inter-thread synchronization. Bonachea [18] proposes a set of UPC extensions that enable strided data transfers and overlap of communication and computation.

CAF is different from UPC in that it does not provide the abstraction of a "flat" shared

address space. CAF has a simple two-level memory model with local and remote memory. In CAF, the programmer explicitly specifies the target of each communication or synchronization. Thus, UPC is more convenient to use for irregular fine-grain codes. On the other hand, CAF enables a compiler to distinguish between local and remote accesses at compile time for free: co-array accesses with brackets are usually remote, co-array accesses without brackets are always local. This enables CAF compilers to optimize local references well, while in UPC, the programmer needs to cast shared pointers to local C pointers to increase efficiency of local accesses [33]. Such casting eliminates the run-time overhead associated with each shared-pointer dereferencing. We found that for dense scientific codes, it is essential to use C99 `restrict` local pointers to indicate lack of aliasing to C compilers to achieve better scalar performance [33].

CAF is based on Fortran 95 and inherits *multi-dimensional* arrays. The lack of multi-dimensional arrays in C and UPC can be an obstacle for achieving high performance due to less precise dependence analysis [33]; because without multi-dimensional arrays, the compiler must analyze linearized polynomial array subscripts, which is a much harder task than analysis of multi-dimensional vector subscripts for multi-dimensional arrays. CAF provides array and co-array sections enabling programmers to conveniently express communication vectorization in the source code. In UPC, programmers must use library-based primitives [18] to express bulk and strided communication, a clear disadvantage compared to CAF. In CAF, co-arrays are equivalent to UPC relaxed variables. UPC's strict variables can be used to implement custom synchronization primitives such as unidirectional point-to-point synchronization [33]. It is possible to implement language-level synchronization in CAF as well; to ensure ordering of co-array accesses, programmers can use the memory fence and CAF's synchronization primitives.

2.3.2 Titanium

Titanium [66] is an explicitly-parallel SPMD language based on Java. It has a few advantages over both CAF and UPC, mainly in what Java can offer C and Fortran 95 developers.

It is an object-oriented, strongly-typed language. It has garbage collected memory as well as zone-based managed memory for performance. It supports multi-dimensional arrays. Remote memory is accessed using global pointers; the `local` type qualifier is used to indicate that a pointer points to an object residing in the local demesne (memory), thus compiler optimizations are possible for local references. The strong type system guarantees compile-time deadlock prevention for programs that use only textual barriers for synchronization; however, the current version of Titanium allows only global textual barriers. Since every process of a parallel program must participate in a global textual barrier, applications such as CCSM [50] that operate in independent, interacting groups of processes cannot be readily expressed in Titanium, using only global textual barriers for synchronization, without major re-engineering. The focus of Titanium language design and implementation is on providing sequential memory consistency without sacrificing performance [74]. It was shown that Titanium can match the performance of Fortran+MPI for the NAS MG, CG, and FT benchmarks on several architectures [45]. Titanium's cross-language application support can alleviate sequential code performance issues by calling optimized computation kernels implemented in Fortran 95 or C.

Compiler analysis for Titanium

Aiken *et al.* [6] use barrier inference to verify that an SPMD program is structurally correct; *i.e.*, the program executes the same number of barriers. They pioneered the notion of *single-valued* expressions that evaluate to the same value on every process. They used the `single` type qualifier to mark single-valued variables and developed a set of type inference rules to statically verify that an SPMD program is structurally correct. They proved their ideas on a simple procedural language, *L*, and adapted them for Split-C [40] and Titanium. However, their analysis is limited only to global textual barriers.

Kamil and Yelick [75] use *textually aligned barriers* (referred to as textual barriers hereafter) as well as single-valued expressions to further improve the analysis of Titanium and to statically verify that a program that uses only textual barriers for synchronization is

deadlock-free. Textual barriers enforce all processes to execute the same barrier statement. Thus, the control flow graph (CFG) [35] can be partitioned to improve the precision of the concurrency analysis [74], which determines the set of all statements that may run concurrently. This reduces the number of memory fences necessary to provide sequential consistency in Titanium [74]. Their analysis is also limited to global textual barriers.

Global textual barriers pose a severe limitation when implementing loosely-coupled applications such as CCSM [50] that execute in independent, interacting groups of processes. Our analysis in Chapter 6 uses textual co-space (or group) barriers and co-space single-valued expressions as compiler hints rather than elements of a type system for two reasons. First, type inference for group textual barriers and group single values is hard, if not impossible, in the case of several groups, and no inference algorithms exist to date. Second, CAF cannot be made a strongly-typed language. Aiken *et al.* [6] use the CFG and single static assignment form (SSA) [41] to derive constraints for single-valued expressions. Solving the system of these constraints yields the maximal set of single values. This approach can be adopted for the inference of single values for a co-space C in a scope where synchronization is done only via textual co-space barriers of the same co-space C. We devised a simpler forward propagation inference algorithm presented in Chapter 6. While our solution is less general and limited to structured control flow, it is sufficient for the synchronization strength reduction (SSR) optimization, presented in Chapter 7, that works only for structured control flow.

2.3.3 Barrier synchronization analysis and optimization

Jeremiassen and Eggers [72] use the presence of barriers to perform non-concurrency analysis of explicitly-parallel programs. Their algorithm, based on barrier synchronization graph and live variable analysis, partitions the program into a set of non-concurrent phases that are delimited by barriers. A phase is a set of statements that may execute concurrently between two global barriers. They apply their analysis to reduce false sharing. It is not clear whether SSR can benefit from non-concurrency analysis. Communication analysis

for SSR uses textual co-space barriers to rely on the fact that all co-space processes execute the same program statement.

Darte and Schreiber [44] present a linear-time algorithm for minimizing the number of global barriers in an SPMD program. The authors acknowledge that minimizing the number of barriers might not yield best performance because optimized barrier placement may introduce load imbalance. SSR replaces textual co-space barriers with point-to-point synchronization and does not introduce load imbalance.

2.4 Message-passing and RPC-based programming models

PGAS languages use one-sided communication to access off-process data. Since several threads of execution can access the same shared data, these languages are concurrent. Programmers must use *explicit* synchronization to ensure the proper ordering of accesses to shared data. In contrast, programming models based on two-sided communication do not use explicit synchronization. In two-sided communication, both communication partners participate in a communication event, which synchronizes them *implicitly*. For some communication patterns, *e.g.*, producer-consumer, two-sided communication is more natural and simpler to use. In addition, an implementation of a two-sided mechanism can use extra storage to buffer communicated data for better asynchrony tolerance between the producer and consumer. In the one-sided programming model, programmers have to manage all details of buffering and pipelined synchronization explicitly to get high performance [31, 32].

In this dissertation, we explore multi-version variables as a practical and efficient solution to simplify program development of high-performance codes with producer-consumer communication in CAF. We first describe two-sided communication in MPI. Then we provide an overview of several programming languages that encapsulate two-sided communication via the abstraction of a stream/link/channel/pipe.

2.4.1 Message Passing Interface

MPI [62] is a library-based programming model that is the *de facto* standard for parallel programming today. Writing parallel programs using MPI is hard, because programmers are responsible for managing all details of parallelization: data decomposition, computation partitioning, communication, and synchronization. In return, MPI programs can achieve high performance and good scalability for a variety of codes. The ability to deliver performance and availability on almost every platform have made MPI the programming model of choice for parallel computing today.

The strength of MPI is that it can be used with almost any programming language. Program developers do not need to learn another programming language to parallelize an application. However, codes written using MPI are harder to optimize because MPI is a library, which limits opportunities for compiler optimization. Communication in MPI programs is expressed in a detailed form, which makes it hard to analyze.

Ogawa *et al.* [89] developed the Optimizing MPI (OMPI) system to reduce software overhead of MPI calls especially for applications with finer-grained communication. OMPI removes much of the excess overhead of MPI function calls by employing partial evaluation techniques, which exploit static information of MPI calls. It also utilizes pre-optimized template functions for further optimization. OMPI work dates from a decade ago; communication latency is now much more significant than CPU overhead due to MPI function calls. However, reducing the overhead of library function calls to perform communication and synchronization is likely to improve the performance of codes with a lot of fine-grain communication, especially pipelined wavefront applications, in any parallel programming model. In this respect, CAF provides better performance portability. For example, a CAF compiler could generate code to use load/store (*e.g.*, via F90 pointers) to perform fine-grain accesses on a shared-memory architecture, eliminating the overhead of function calls altogether [48].

Compared to MPI, CAF offers programmers more convenient syntax for communication based on Fortran 95 array sections as well as type/shape checking for co-array ac-

cesses. A CAF compiler has more opportunities for efficient tailoring of generated code to the target architecture and run-time layer without the need to modify the source program. For example, it can generate code to use CPU load/store or vector instructions for remote co-array accesses on shared-memory architectures. A CAF compiler can also vectorize and/or aggregate remote co-array accesses on cluster architectures and generate non-blocking communication. Our evaluation studies [30, 47, 48, 31, 32, 33] showed that even without compiler support for optimizations, it is possible to achieve the same level of performance in CAF as with MPI. However, achieving high performance without optimizations is as hard as for MPI. A part of this thesis explores extending CAF with abstractions that simplify development of high-performance codes in CAF, *e.g.*, the multi-version variables to compensate for the lack of two-sided communication in CAF.

MPI uses send and receive library primitives to express two-sided message passing. An implementation usually supports two communication modes: eager and rendezvous communication protocols. The eager protocol is used for small messages. The send operation does not require a matching receive to send data; instead, the data is copied into an auxiliary buffer on the sender and then communicated to the receiver, or is communicated into an auxiliary buffer on the receiver and then copied to the destination, when the receiver participates in communication. The rendezvous protocol is used for large messages; the sender does not start data transmission and is blocked in send until a matching receive is executed by the receiver. The eager protocol enables better asynchrony tolerance, but uses more memory for buffering and exhibits extra memory copies. Some interconnects such as Myrinet require that data being communicated resides in registered memory. Because MPI can transfer arbitrary user variables and some of them may reside in unregistered memory, MPI might incur extra memory registration/copying/unregistration overhead. For in-core scientific applications, a CAF compiler can avoid this overhead by allocating co-arrays in registered memory without the programmer's intervention.

CAF's multi-version variables (MVVs) are a language construct, not a library primitive. Thus, they are more amenable to compiler-based optimizations. Communication via

MVVs resembles the MPI eager protocol. The sender knows the address of the destination memory for each data transfer, which has two advantages. The memory can be allocated from a special memory pool, *e.g.*, registered memory and extra memory copies can be avoided at the source (with proper compiler analysis or hints) and destination by adjusting MVV's Fortran 90 array descriptor (see Chapter 8). To summarize, CAF's MVVs offer clean and simple semantics of two-sided communication and can deliver comparable or better performance than that of MPI's send/receive.

2.4.2 Message passing in languages

Several languages encapsulate two-sided communication via the abstraction of a stream (or link/channel/pipe). We believe that these abstractions are too general, better-suited for distributed programming rather than SPMD high performance programming. MVVs provide less generality, but, in our opinion, are more convenient to use in a broad class of scientific applications. They can also be optimized to avoid extra memory copies by communicating data in-place; this is hard to do for the more general stream abstraction. In addition, some of message-passing languages provide limited capabilities for executing code in remote process. Our distributed multithreading discussed in Chapter 9 is more general and flexible.

Lynx. Scott presents Lynx [108] with the abstraction of the *link*, a two directional communication channel for type-checked message passing. Links are first-class objects in Lynx and can be passed to other processes, supporting dynamic topology changes. Lynx is well-suited for the programming of distributed systems and client-server applications. Because links are a rather general abstraction, enabling even inter-program communication, it is not clear whether it is possible to optimize them to deliver the best performance.

Lynx's links provide a form of cooperative multithreading in each communicating program. Lynx allows only one active thread per program, and each consumer executes code with run-until-block semantics when it handles a message. As we discuss in Chapter 9, only one hardware thread of execution per program is not enough to exploit the parallelism

available within modern cluster nodes with parallel execution context, *e.g.*, multi-core multiprocessors; also, run-until-block semantics are not appropriate for multithreaded high performance computing. In Lynx, consumers can use the "early reply" to unblock producers; this concept inspired the "remote return" (`reply`) concept in distributed multithreading (see Section 9.3).

Fortran M. Foster *et al.* propose Fortran M [54] that uses *channels* to plug together Fortran processes. Channels are used for passing messages between tasks. Fortran M allows variable message sizes, dynamic topology changes, and many-to-one communication. Compared to MVVs, channels require explicit connection and are harder to optimize.

Strand and PCN. Strand and PCN [53] are compositional programming languages designed by Foster. Strand is commonly used as a coordination language to control the concurrent execution of sequential modules. It has only single-assignment (definitional) variables used to communicate values from producers to consumers and to synchronize them. A stream of values, accumulated in a list, is used to communicate data; these lists need to be garbage collected. While Strand is a powerful symbolic and distributed programming language, it is ill-suited for numeric codes. Program Composition Notation (PCN) improves on Strand in combining declarative and imperative programming. Both Strand and PCN are hard to optimize because of single-assignment variables and garbage collection. It is not clear whether they can deliver performance of hand-optimizes MPI or MVV-based codes.

Teleport Messaging. Thies *et al.* propose *Teleport Messaging* [117] to solve the problem of precise handling of events across parallel system. Control messages that change the state are treated as special data messages. When a receiver receives a control message, it invokes the associated handler that changes the corresponding state. Because control messages flow with the data in the stream, data dependencies enforce the precise timing (with respect to the data stream) of executing the action carried by a control message. This approach enables optimizing the signal processing applications, modeled in a Cyclo-Static Dataflow language, *e.g.*, StreamIt [116], by exposing the true data dependence to the com-

piler. Teleport Messaging provides limited capabilities for execution of code in a remote process; however, it is too restrictive for multithreading in many scientific applications.

Space-Time Memory. Ramachandran *et al.* developed the *Space-Time Memory* abstraction [96] — a dynamic concurrent distributed data structure for holding time-sequenced data. STM is designed for interactive multimedia applications to simplify complex buffer management, intertask synchronization, and meeting soft real-time constraints. STM has globally known *channels* where threads can PUT a data item with a timestamp and GET a data item with a timestamp. The semantics of PUT and GET are copy-in and copy-out. Unused memory is globally garbage collected. If used directly in the program, STM requires programmers to establish connections, which is not necessary with MVVs, and pack/unpack strided transmitted data, which is usually not necessary with MVVs. We could use STM as a vehicle to implement MVVs; however, as of this writing, an STM implementation is available only for a cluster of Alpha SMPs running Digital UNIX. Also, an STM-based implementation of MVVs is likely to have more overhead than a lighter-weight implementation based on Active Messages (AM).

2.5 Concurrency in imperative languages

Another approach to simplify the development of concurrent programs is to enable variables that can hold infinitely many values managed with the semantics of a stream. The simplicity comes from removing the anti- and output dependencies due to variable memory reuse.

2.5.1 Single-Assignment C

Grelck and Scholz present Single Assignment C (SAC) [59], a purely functional array processing language. Programming in SAC can be thought of as programming in Static Single Assignment form (SSA) [41]. Each assignment is done into a new memory location, thus, there are no anti- and output data dependencies. While this simplifies many compiler optimizations and enables detection of parallelism, the compiler is fully respon-

sible for code optimization. SAC is a functional programming language and, in practice, does not offer a "natural" programming style for imperative-language programmers. Several studies [59, 58, 60] reported reasonable performance results for a few benchmarks, including NAS MG and FT, on a 12-processor SUN Ultra Enterprise 4000 shared-memory multiprocessor; however, no comparison was done with explicitly-parallel models such as Fortran+MPI, CAF, or UPC. As of this writing, there is no implementation of SAC for a cluster. It would be interesting to see whether a functional language without the notion of data locality can be optimized to deliver high performance on large-scale cluster architectures for a broad class of scientific applications.

2.5.2 Data-flow and stream-based languages

Streams and Iteration in a Single Assignment Language — Sisal [2] — is a general-purpose, single assignment, functional programming language with strict semantics, automatic parallelization, and efficient array handling. The strong point of Sisal is that programs are deterministic, regardless of platform or environment. Several Sisal compiler implementations for distributed- and shared-memory platforms were reported [20, 55, 107, 57]. Good performance was demonstrated for small-scale parallel machines. It would be interesting to see whether Sisal, which does not offer programmers any means to control data distribution, can be optimized to deliver high performance on large-scale distributed-memory machines such as Blue Gene/L.

Many data-flow and stream-based domain specific languages, *e.g.*, YAPI [46], used for signal processing systems are based on Kahn process networks (KPNs) [123], a distributed model of computation where a group of processes is connected by communication channels. Processes communicate via unbounded first-in-first-out (FIFO) data channels. Processes read and write atomic data elements or tokens from and to channels. Writing to a channel is non-blocking; *i.e.* it always succeeds and does not stall the process, while reading from a channel is blocking; *i.e.*, a process that reads from an empty channel will stall and can only continue when the channel contains sufficient data items. Given a spe-

cific input (token) history for a process, the process must be deterministic in that it always produces the same outputs (tokens). Timing or execution order of processes must not affect the result and, therefore, processes are not allowed to test an input channel for existence of tokens without consuming them. The KPN computation model is too restrictive for general-purpose scientific applications. It is also unlikely that the abstraction of channels that can hold infinitely many tokens in-flight would be optimized in CAF to deliver the best performance. We limit the number of unconsumed versions that can be buffered by a multi-version variable and block the producer that tries to commit another value until one of the buffered values is consumed.

2.5.3 Clocked final model

Saraswat *et al.* proposed the *clocked final* (CF) model [106] to address the difficulty of concurrency in imperative languages and express concurrent applications in a natural way similar to sequential code. CF guaranties determinacy and deadlock freedom. Under CF, each mutable location, *e.g.*, shared scalar or an array element, is associated with a *clocked stream* of immutable (final) values. Writers write the location at different stream indices and readers consume the locations at a particular index. If there is no value to read at the index, the reader blocks until the value is available. The stream is conceptually infinite. In practice, the number of items buffered for a stream should be bounded, and analysis is required to determine the right buffer size. While the concept looks appealing, experimental evidence is required to prove that the CF model can be optimized to deliver high performance for a variety of codes on a range of parallel platforms. In our approach, programmers explicitly specify which variables are multi-version and, thus, will be used for data streaming. The number of versions that an MVV can hold is finite and can be specified explicitly by the programmer. We chose to support only “PUT-style” (push strategy) multi-version variables (MVVs), since they deliver the best performance for distributed-memory machines; we could also extend MVVs to support “GET-style” (pull strategy) retrieves, but this inherently exposes communication latency. There can be multiple producers that com-

mit values into an MVV located on process image p, but there is only one consumer process image — p — that retrieves vales locally. While MVVs are less general than the CF model, they can simplify development of many parallel codes and deliver high performance today.

2.6 Function shipping

The distributed multithreading for CAF evaluated in this work is based on the concept of function shipping. While function shipping has been used in many contexts, the novelty of this research is the first design and evaluation of function shipping for CAF.

2.6.1 Remote procedure calls

The idea of performing a computation by the remote processor is quite old. The *Remote Procedure Calls (RPC)* [13] provide the ability to invoke a procedure remotely; input parameters can be passed by-value and the result returned back to the caller. This is a library-based approach and the programmer is responsible for parameter marshaling/unmarshaling and thread management.

2.6.2 Active Messages

Eicken *et al.* [122] propose *Active Messages (AM)*. An AM header contains the address of a user-level handler that is executed on message arrival with the message body as argument. Thus, arbitrary user code can be executed in the remote address space. The active message handler must execute quickly and to completion and not block the network hardware from receiving other messages. The AM concept is implemented in both ARMCI [83] (called Global Procedure Calls) and GASNet[17]. However, it is too restrictive to be used as is in the language. Our implementation of CAF's distributed multithreading (DMT), described in Chapter 9, uses AM to support language-level remote activities. However, DMT allows arbitrarily long, potentially blocking computations spawned locally and remotely.

2.6.3 Multilisp

Halstead introduced the concept of the *future* in Multilisp [103]. It allows one to spawn a concurrent computation and returns an undetermined value without blocking the caller. If the return value is needed for computation in one of the following statements, the statement blocks until the future executes and returns the real (determined) value. Futures enable lazy evaluation of parallel work and allow transparent concurrency in functional languages on shared-memory architectures.

2.6.4 Cilk

Cilk [63] is a language that enables concurrent execution of several tasks on a shared-memory multiprocessor. It introduces the concept of provably good "work-stealing" scheduler [15] that maintains load balancing among the processors transparently to the programmer. Cilk has the limitation that all tasks spawned within a function must complete before the function returns. An implementation of Cilk for a distributed-memory architecture has been reported [97]. It is based on principles of software shared-memory, and the scheduler's heuristic is biased towards stealing "local" work. However, the performance results were inconclusive and it is unlikely that a programming model without the notion of data locality can perform well on large-scale distributed-memory machines for a broad range of codes.

2.6.5 Java Remote Method Invocation

Java [113] offers the *Remote Method Invocation* mechanism that enables calling a method of a remote object. The object must be registered with a global repository. The parameter passing is done via a well-defined Java serialization/deserialization mechanism, though the overhead is high. CAF is not an object-oriented language, and it is necessary to design robust semantics of how parameters are passed to a co-subroutine and how the values can be returned. We discuss these issues in Chapter 9.

Chapter 3

Background

We discuss the CAF programming model and run-time layer support for an efficient CAF implementation. Then, we describe experimental platforms and parallel codes that we used for our evaluation studies. Finally, we summarize the basics of data-flow analysis.

3.1 Co-array Fortran

CAF is a Single Program Multiple Data (SPMD) Partitioned Global Address Space (PGAS) programming model based on a small set of extensions to Fortran 95. It has a two-level memory model with local and remote data and provides the abstraction of globally accessible memory both for cluster-based and for shared-memory architectures. Similar to MPI, CAF is an explicitly-parallel programming model. CAF programmers partition data and computation and use explicit communication and synchronization. Access to remote data is done via one-sided read (GET) or write (PUT) communication.

An executing CAF program consists of a static collection of asynchronous process images (or images, for short). The number of images can be retrieved at run time by invoking the intrinsic function `num_images()`. Each image has a unique index from one to `num_images()`, which can be retrieved via the intrinsic function `this_image()`. The programmer controls the execution sequence in each image through explicit use of Fortran 95 control constructs and through explicit use of synchronization intrinsics.

Below we describe CAF features related to this work. A more complete description of the CAF language can be found elsewhere [88].

3.1.1 Co-arrays

CAF supports symmetric distributed data using a natural extension to Fortran 95 syntax. For example, the declaration `integer::a(n,m)[*]` declares a shared co-array `a` with `n` $\times$ `m` integers local to each process image. The dimensions inside brackets are called co-dimensions and define the co-shape; their number is called co-rank. Co-shape can be thought of as an arrangement of all program images into a Cartesian topology without periodic boundaries. For example, the declaration

```
integer::a(n,m)[2,4,*]
```

specifies a co-shape that represents a Cartesian topology with dimensions $2 \times 4 \times 8$ when the total number of images is 64. We discuss the limitations of CAF co-shapes in Chapter 5 and present the concept of co-spaces as a more general and flexible specification of communication topologies for CAF.

Co-arrays may be static data specified by `COMMON` or `SAVE` or they can be dynamic data specified by `ALLOCATABLE`. They can also be passed as procedure arguments. Co-arrays may be declared for primitive types as well as user-defined types. A local section of a co-array may be a singleton instance of a type rather than an array of type instances.

3.1.2 Accessing co-arrays

Instead of explicitly coding message exchanges to access data residing in other process' memories, CAF programmers can directly reference non-local values using an extension to Fortran 95 syntax for subscripted references. For instance, process `p` can read the first column of co-array `a` from process `p+1` by referencing `a(:,1)[p+1]`. If the square brackets are omitted, the reference is to the local co-array data. Remote co-array references naturally induce a one-sided communication style in which all transfer parameters are supplied by the image executing the communication and the target process image might not be aware of the communication.

3.1.3 Allocatable and pointer co-array components

Allocatable and pointer co-array components can be used for asymmetric shared data structures. They can be of intrinsic or user-defined types. For example, a co-array `a` may have a pointer component `ptr(:)`. `a[p]%ptr(i:j)` references the array section of `a%ptr` located in image `p`. An allocatable component must be allocated prior to use. A pointer component must point to a co-array or Fortran 95 variable located in the local memory. Brackets are allowed only for the first level of a co-array component access, *e.g.*, `b(i,j)[p]%ptr1(x)%ptr2(:)`; the other levels are relative to the remote image `p` and cannot have brackets. The rules for evaluating the implicit bounds, *e.g.*, for a `a[p]%ptr(:)` reference, are not precisely stated in the current CAF standard. We assume that the implicit bound values are those on the target image `p`.

3.1.4 Procedure calls

Co-arrays are allowed to be procedure arguments and can be reshaped at a procedure call. An explicit interface is required for co-array parameters. There are no local co-array variables since the procedure activation frame might not exist on every image. The original CAF specification required implicit memory fences before and after each procedure call. In [30], we argued that such memory fences make it impossible to overlap communication with a procedure's computation. The requirement was removed in the updated language specification [86].

3.1.5 Synchronization

CAF has a `sync_memory` memory fence to explicitly complete all outstanding communication operations issued by the invoking process image; this is a local operation. `sync_all` implements a synchronous barrier among all images. `sync_team` is used for synchronization among teams of two or more processes.

In [30], we considered augmenting CAF with unidirectional, point-to-point synchronization primitives: `sync_notify` and `sync_wait`. `sync_notify(q)` sends a non-

blocking notification message to process image `q`; this notification is guaranteed to be seen by image `q` only after all communication events previously issued by the notifier to image `q` have been committed into `q`'s memory. `sync_wait(p)` blocks its caller until it receives a matching notification message from the process image `p`. The updated CAF specification [86] allows using `notify_team` and `wait_team` for unidirectional point-to-point synchronization. The names of the unidirectional point-to-point synchronization primitives in the CAF specification may change in the future; therefore, we refer to these primitives as `notify(p)` and `wait(q)` in the rest of the dissertation.

3.1.6 CAF memory consistency model

A memory consistency model defines legal orderings of how the results of write operations can be observed via read operations. Stricter memory consistency helps in developing and debugging programs; but limiting the scope of legal code reorderings can result in lower performance.

As of this writing, CAF's memory consistency model is still being defined. For the purposes of this work, we assume a *weak* memory consistency model with the following rules:

- Each image's own data dependencies for accessing local data must be preserved.
- The ordering of shared accesses is guaranteed only by synchronization primitives such as the memory fence `sync_memory`, global barrier `sync_all`, team synchronization `sync_team`, and `notify`/`wait` unidirectional point-to-point primitives.

The first rule enables leveraging existing scalar compiler technology for optimizing local accesses in between synchronization points. It is essential for source-to-source translation and makes it legal to use a Fortran 95 compiler of the target architecture to optimize translated code.

The second rule enables programmers and the compiler to make assumptions about the memory state and completed PUT/GET operations of images participating in a syn-

chronization event. The memory fence `sync_memory` completes all outstanding memory operations issued by the image before execution of the fence. Completion means that local accesses have been flushed to memory, GETs have finished reading the remote memory, and PUTs have been written to the remote memory. Global barrier `sync_all` implies that every image executes an implicit memory fence right before participating in the barrier synchronization. In other words, each image participating in a barrier can assume that after the return from the barrier, all memory operations issued by any participant prior to the barrier have been completed.

There is still ongoing debate over what memory guarantees should be for point-to-point (or team) synchronization. Two alternatives are being considered. First, `notify(p)` may have an implicit memory fence, which completes all memory operations issued by the invoking image before `p` receives the notification. Second, `notify(p)` may have weaker semantics explored by us for `sync_notify` in [30]: a delivery of `notify(p)` from `q` to `p` guarantees only that all PUTs/GETs issued by `q` to `p` have completed. The following example illustrates the difference.

```
if (this_image()==q)
   a[r] = ...
   b[p] = ...
   call notify(p)
end if
if (this_image()==p)
   call wait(q)
end if
```

With the first proposal, the programmer can assume that both `a[r]` and `b[p]` PUTs have completed when `p` receives the notification. The semantics are intuitive; however, no compiler optimization technology yet exists that can optimize this code to deliver best performance. Most likely, an implementation would have to complete both PUTs before executing the `notify(p)`; this exposes the communication and notification latency to `p`.

In the second proposal, `notify(p)` received by `p` guarantees only that `b[p]` has completed, but does not guarantee that `a[r]` has completed ($p \neq r$). These semantics might be less intuitive to the programmer; however, we have not yet observed CAF codes

that rely on the assumptions that `a[r]` must also complete. The compiler and run-time can optimize this code to hide all communication and synchronization latency. During our experimental studies, we noticed that the weaker notify semantics enabled to deliver noticeably higher performance and better scalability. For the rest of this work, we assume the weaker form of `notify(p)`.

We also assume similar ("closed group") weak semantics for a group barrier. Execution of a group barrier guarantees that all PUTs/GETs executed by all images of the group before the barrier and destined to any member image of the group have completed. No guarantees are provided for a PUT/GET executed by any image of the group and destined to an image that is a not a member of the group.

We believe that these weak semantics are intuitive enough and would not restrict the compiler and run-time to deliver top performance. Alternatively, both forms of `notify` and group barrier can co-exist and can be distinguished by an optional extra parameter to `notify` that specifies what memory guarantees to provide.

3.2 Communication support for PGAS languages

Message Passing Interface (MPI) [62, 112], the *de facto* standard for parallel programming, uses two-sided (send and receive) message passing to transfer data between processes. With two-sided communication, both the sender and the receiver explicitly participate in a communication event and supply communication parameters. As a consequence, both sender and receiver temporarily set aside their computation to communicate data, which also has heavy impact on the programming style. Having two processes complete a send/receive communication explicitly synchronizes them.

PGAS languages use one-sided communication to access remote data: GET is used for a remote read and PUT is used for a remote write. In one-sided communication, only the process, called the origin of communication, executing the remote access specifies the target process and all other communication parameters. No synchronization between the origin and target processes takes place. From the programmer's perspective, the target

image is not aware of the communication. Thus, the one-sided model cleanly separates data movement from synchronization, which can be particularly useful for development of irregular applications. To achieve high performance with PGAS languages for a broad class of applications, one-sided communication must be efficient. Co-array sections may reference remote memory that is strided. Therefore, communication must be efficient not only for co-array sections that reference contiguous memory (contiguous transfers) but also for those that reference strided memory (strided transfers). Implementing an efficient one-sided communication layer is not a trivial task due to interconnect hardware differences.

On shared-memory platforms, such as the SGI Altix 3000, one-sided communication can be performed for globally addressable shared-memory by the CPU, using load/store instructions. As our recent study [48] demonstrated, on shared-memory architectures, fine-grain one-sided communication is fastest with compiler generated load/store instructions, and large contiguous transfers are done more efficiently by using a memory copy library function optimized for the target platform.

On loosely-coupled architectures, a one-sided communication layer can take advantage of Remote Direct Memory Access (RDMA) capabilities of modern networks, such as Myrinet [9] and Quadrics [94]. During an RDMA data transfer, the Network Interface Controller (NIC) controls data movement without interrupting the remote host CPU. This enables the remote CPU to compute while communication is in progress.

Several specifications for one-sided communication were designed to encapsulate hardware differences and to simplify PGAS compiler development [83, 17]. We describe Aggregate Remote Memory Copy Interface (ARMCI) and GASNet, which we used to implement `cafc`'s run-time layer.

3.2.1 ARMCI

ARMCI [83] is a multi-platform library for high performance one-sided communication. ARMCI provides both blocking and non-blocking primitives for one-sided data movement as well as primitives for efficient unidirectional point-to-point synchronization. On

some platforms, using split-phase primitives enables communication to be overlapped with computation. ARMCI provides an excellent implementation substrate for global address space languages because it achieves high performance on a variety of networks (including Myrinet, Quadrics, and IBM's switch fabric for its SP systems) as well as on shared-memory platforms (Cray X1, SGI Altix3000, SGI Origin2000), while insulating its clients from platform-specific implementation issues such as shared memory, threads, and DMA engines. A notable feature of ARMCI is its support for efficient non-contiguous data transfers [84], essential for delivering high performance with CAF. ARMCI provides support for Global Procedure Calls (GPCs) on Myrinet, Quadrics, and InfiniBand interconnects. GPCs enable execution of procedures in remote process. For the rest of the discussion, we refer to ARMCI's GPCs as Active Messages (AMs) [122].

3.2.2 GASNet

GASNet [17], standing for "**G**lobal-**A**ddress **S**pace **Net**working", is another one-sided communication library. The GASNet library is optimized for a variety of cluster and shared-memory architectures and provides support for efficient communication by applying communication optimizations such as message coalescing and aggregation as well as optimizing accesses to local shared data. As of this writing, GASNet has only a reference implementation for strided communication and shows lower performance for strided transfers compared to ARMCI for some interconnects.

The design of GASNet is partitioned into two layers to make porting easier without sacrificing performance. The lower level provides a core subset of functionality called the GASNet core API. It is based on Active Messages [122], and is implemented directly on top of each individual network architecture. The upper level is a more expressive interface, called the GASNet extended API. It provides high-level operations to access remote memory and various collective operations.

3.3 Experimental platforms

We used several cluster and non-uniform memory access (NUMA) shared-memory architectures to perform our experiments.

3.3.1 Itanium2+Myrinet2000 cluster (RTC)

The Rice Terascale Cluster (RTC) [102] is a cluster of 92 HP zx6000 workstations interconnected with Myrinet 2000. Each workstation node contains two 900MHz Intel Itanium 2 processors with 32KB/256KB/1.5MB of L1/L2/L3 cache, 4-8GB of RAM, and the HP zx1 chipset. Each node is running the Linux operating system. We used the Intel Fortran compiler versions 8.x-9.x for Itanium as our Fortran 95 back-end compiler.

3.3.2 Itanium2+Quadrics cluster (MPP2)

MPP2 consists of 2000 HP Long's Peak dual-CPU workstations at the Pacific Northwest National Laboratory (PNNL). The nodes are connected with Quadrics QSNet II (Elan 4). Each node contains two 1.5GHz Itanium2 processors with 32KB/256KB/6MB L1/L2/L3 cache and 4GB of RAM. The operating system is Red Hat Linux. The back-end compiler is the Intel Fortran compiler version 8.0.

3.3.3 Alpha+Quadrics cluster (Lemieux)

Lemieux is a cluster at the Pittsburgh Supercomputing Center (PSC). Each node is an SMP with four 1GHz Alpha EV68 processors and 4GB of memory. The operating system is OSF1 Tru64 v5.1A. The cluster nodes are connected with Quadrics QSNet (Elan3). The back-end Fortran compiler used was Compaq Fortran V5.5.

3.3.4 Altix 3000 (Altix1)

The SGI Altix 3000 machine (Altix1) at Pacific Northwest National Laboratory (PNNL) is a NUMA shared-memory multiprocessor that has 128 Itanium2 1.5GHz processors each

with 32KB/256KB/6MB L1/L2/L3 cache, and 128 GB RAM, running the Linux64 OS and the Intel Fortran compiler version 8.1.

3.3.5 SGI Origin 2000 (MAPY)

The SGI Origin 2000 machine at Rice University has 16 MIPS R12000 processors with 8MB L2 cache and 10 GB RAM. It runs IRIX 6.5 and the MIPSpro Compilers version 7.3.1.3m

3.4 Parallel benchmarks and applications

Throughout our studies, we use several parallel codes to extensively evaluate the CAF language, the quality of code generated by our CAF compiler, and performance of the run-time communication library. Each code contains a regular or irregular computation that represents the kernel of a realistic scientific application. These codes are widely regarded as useful for evaluating the quality of parallel compilers and used in this thesis to evaluate the effects of different optimization techniques.

For most of our experiments, we compare the parallel efficiency of different CAF versions to that of MPI version used as the baseline for comparison. We compute parallel efficiency as follows. For each parallelization ρ, the efficiency metric is computed as $\frac{t_s}{P \times t_p(P,\rho)}$. In this equation, t_s is the execution time of the sequential version; P is the number of processors; $t_p(P, \rho)$ is the time for the parallel execution on P processors using parallelization ρ. Using this metric, perfect speedup would yield efficiency of 1.0 for each processor configuration. We use efficiency rather than speedup or execution time as our comparison metric because it enables us to accurately gauge the relative performance of multiple benchmark implementations across the *entire* range of processor counts and even across different architectures.

3.4.1 NAS Parallel Benchmarks

The NAS Parallel Benchmarks (NPB) [12, 73] are implemented by the NASA Advanced Supercomputing (NAS) Division group at the NASA Ames Research Laboratory. They are designed to help evaluate the performance of parallel supercomputers. We used MPI, OpenMP, and serial flavors of official NPB 2.3, NPB 3.0, and NPB 2.3-serial releases respectively [12, 73]. We implemented corresponding CAF versions by modifying communication and synchronization in the MPI benchmarks, without changing the original algorithms.

NAS MG. The MG multigrid kernel calculates an approximate solution to the discrete Poisson problem using four iterations of the V-cycle multigrid algorithm on a $\mathtt{n} \times \mathtt{n} \times \mathtt{n}$ grid with periodic boundary conditions [12]. MG's communication is highly structured and repeats a fixed sequence of regular patterns.

In the NAS MG benchmark, for each level of the grid, there are periodic updates of the border region of a three-dimensional rectangular data volume from neighboring processors in each of six spatial directions. Four buffers are used: two as receive buffers and two as send buffers. For each of the three spatial axes, two messages (except for the corner cases) are sent using basic MPI send to update the border regions on the left and right neighbors. Therefore, two buffers are used for each direction, one buffer to store data to be sent and the other to receive the data from the corresponding neighbor. Because two-sided communication is used, there is implicit two-way point-to-point synchronization between each pair of neighbors.

NAS CG. The CG benchmark uses a conjugate gradient method to compute an approximation to the smallest eigenvalue of a large, sparse, symmetric positive definite matrix [12]. This kernel is typical of unstructured grid computations in that it tests irregular long distance communication and employs sparse matrix vector multiplication. The irregular communication employed by this benchmark is a challenge for cluster architectures.

NAS BT and SP. The NAS BT and SP benchmarks are two simulated computational fluid dynamics (CFD) applications that solve systems of equations resulting from an ap-

proximately factored implicit finite difference discretization of three-dimensional Navier-Stokes equations [12]. The principal difference between the codes is that BT solves block-tridiagonal systems of 5x5 blocks, whereas SP solves scalar penta-diagonal systems resulting from full diagonalization of the approximately factored scheme [12]. Both consist of an initialization phase followed by iterative computations over time steps. In each time step, boundary conditions are first calculated. Then the right hand sides of the equations are calculated. Next, banded systems are solved in three computationally intensive bi-directional sweeps along each of the x, y, and z directions. Finally, flow variables are updated. During each time-step, loosely-synchronous communication is required before the boundary computation, and tightly-coupled communication is required during the forward and backward line sweeps along each dimension.

Because of the line sweeps along each of the spatial dimensions, traditional block distributions in one or more dimensions would not yield good parallelism. For this reason, SP and BT use a skewed-cyclic block distribution known as multipartitioning [12, 81]. With multi-partitioning, each processor handles several disjoint blocks in the data domain. Blocks are assigned to the processors so that there is an even distribution of work for each directional sweep, and that each processor has a block on which it can compute in each step of every sweep. Using multipartitioning yields full parallelism with even load balance while requiring only coarse-grain communication.

3.4.2 Sweep3D

The benchmark code Sweep3D [4] represents the heart of a real Accelerated Strategic Computing Initiative (ASCI) application. It solves a one-group time-independent discrete ordinates (Sn) 3D Cartesian (XYZ) geometry neutron transport problem. The XYZ geometry is represented by an IJK logically rectangular grid of cells. The angular dependence is handled by discrete angles with a spherical harmonics treatment for the scattering source. The solution involves two steps: the streaming operator is solved by sweeps for each angle and the scattering operator is solved iteratively.

```
do iq = 1, 8                ! octants
  do mo = 1, mmo            ! angle pipelining loop
    do kk = 1, kb           ! k-plane pipelining loop

      receive from east/west into Phiib    ! recv block I-inflows
      receive from north/south into Phijb  ! recv block J-inflows

      ...
      ! computation that uses and updates Phiib and Phijb
      ...

     send Phiib to east/west               ! send block I-outflows
     send Phijb to north/south             ! send block J-outflows

    enddo
  enddo
enddo
```

Figure 3.1 : Sweep3D kernel pseudocode.

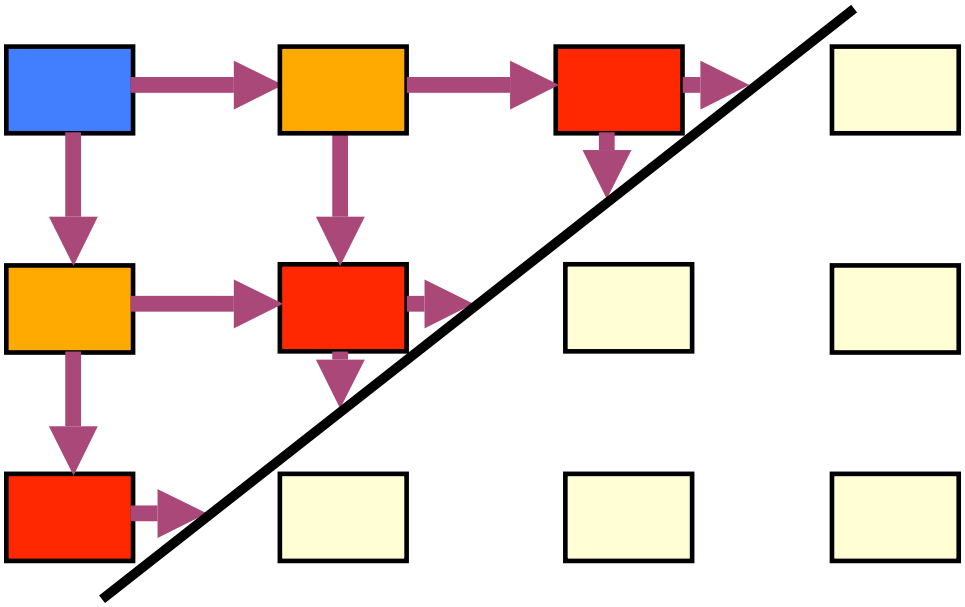


Figure 3.2 : Wavefront communication in Sweep3D.

Sweep3D exploits wavefront parallelism. It uses a 2D spatial domain decomposition of the I- and J-dimensions onto a 2D logical grid of processors. For efficient parallelization, Sweep3D is coded to pipeline blocks of MK k-planes and MMI angles through this 2D processor array. Thus, the wavefront exploits parallelism in both I- and J-directions simul-

taneously. Figure 3.1 shows a piece of pseudocode representing a high-level view of the Sweep3D kernel. Figure 3.2 provides a visualization of a Sweep3D sweep on a 2D logical processor grid: the boxes represent the processors, the sweep starts in the top left corner, the arrows show the direction of PUT communication, and the thick solid line shows the front of computation propagation in 2D distributed pipeline. A more complete description of Sweep3D can be found in [4].

3.4.3 RandomAccess

The RandomAccess benchmark measures the rate of random memory updates. It is available in serial and MPI versions as a part of the HPC Challenge benchmark suite [1].

The serial version of RandomAccess[1] declares a large array `Table` of 64-bit words and a small substitution table `stable` to randomize values in the large array. The array `Table` has the size of `TableSize` = 2^n words. After the `Table` has been initialized, the code performs a number of random updates on `Table` locations. The kernel of the serial benchmark is shown in Figure 3.3 (a).

```
do i = 0, 4*TableSize
  pos = <random number in
        [0,TableSize-1]>
  pos2 = <pos shifted to index
        inside stable>
  Table(pos) = Table(pos) xor
        stable(pos2)
end do
```

(a) Sequential RandomAccess

```
do i = 0, 4*TableSize
  gpos = <random number in
       [0, GlobalTableSize-1]>
  img  = gpos div TableSize
  pos  = gpos mod TableSize
  pos2 = <pos shifted to index
       inside stable>
  Table(pos)[img] = Table(pos)[img] xor
       stable(pos2)
end do
```

(b) CAF RandomAccess

Figure 3.3 : RandomAccess Benchmark.

We implemented a fine-grain CAF version of the benchmark representing the global table as a co-array. The aggregate size of the `Table` is `GlobalTableSize=TableSize` $\times$ `num_images()`, where `TableSize` is the number of words residing in each image.

[1]We used the Table Toy Benchmark (08/01/97 version).

Every image has a private copy of the substitution table. All images concurrently generate random global indices and perform the update of the corresponding `Table` locations. No synchronization is used for concurrent updates (errors on up to 1% of the locations due to race conditions are acceptable). The kernel for CAF variants of RandomAccess is shown in Figure 3.3 (b).

A parallel MPI version [1] of RandomAccess uses buckets to locally cache updates destined to remote memories. Each image has the number of buckets equal to the number of processes; one bucket per destination processes. When a bucket becomes full, the code executes MPI_Alltoall symmetric collective communication to exchange updates among processes. Each process receives updates destined to it from all processes and, then, applies the updates to its local portion of the `Table`. Compared to the fine-grain CAF version, the bucketed version improves locality and increases communication granularity. However, it is more difficult to code and caching becomes problematic for large scale parallel machines. We also implemented and experimented with several CAF bucketed versions. These results can be found in Section 9.6.3.

3.4.4 Data-flow analysis

We briefly describe several concepts of data-flow analysis that we use in Chapters 6 and 7.

Control flow graph

A control flow graph (CFG) [35] represents flow of control in the program. It is a directed graph, $G = (N, E)$. Each node $n \in N$ corresponds to a basic block (BB), a sequence of operations that always execute together. Each edge $e = (n_i, n_j) \in E$ corresponds to a potential transfer of execution control from BB n_i to BB n_j. A CFG has a unique entry node and a unique exit node.

Static single assignment form

Single static assignment form (SSA) [41] is an intermediate representation of the program in which each variable is assigned exactly once. A program is in SSA form when it satisfies two conditions: (1) each definition (assignment) has a distinct name and (2) each use refers to exactly one definition. To satisfy these properties, a compiler must insert ϕ-functions at control flow path merge points and then rename the variables to satisfy the single assignment property. SSA provides an intuitive and efficient namespace that incorporates the information about the relation between definitions and uses, simplifying many compiler analysis and optimization algorithms.

Dominance and postdominance

In a control flow graph, a node a dominates a node b if every path from the CFG entry node to b must pass through a. a strictly dominates b if a dominates b and $a \neq b$. A node a is the *immediate dominator* of a node b if a dominates b and every other dominator of b dominates a [78]. A *dominator tree* is a tree in which the children of each node are those it immediately dominates. The *dominance frontier* of a CFG node a is the set of all nodes b such that a dominates a predecessor of b, but does not strictly dominate b [41].

A node a postdominates a node b if every path from b to the CFG exit node passes through a. The concepts of immediate postdominator and postdominator tree are similar to those of dominators. The *reverse dominance frontier* of a node a can be computed as the dominance frontier of a on the reverse CFG; a reverse CFG can be obtained by reversing each edge of the original CFG.

Chapter 4

Co-array Fortran for Distributed Memory Platforms

We discuss important design and implementation details of `cafc`, a multi-platform open-source Co-array Fortran compiler. Then, we present selected results of our evaluation studies [30, 47, 48, 31, 32] that we performed to determine what level of performance can be delivered with CAF.

4.1 Rice Co-array Fortran compiler — `cafc`

We designed the `cafc` compiler for Co-array Fortran with the major goals of being portable and delivering high performance on a multitude of platforms. To support multiple platforms efficiently, `cafc` performs source-to-source transformation of CAF code into Fortran 95 code augmented with communication operations. By employing source-to-source translation, `cafc` aims to leverage the best back-end compiler available on the target platform to optimize local computation. For communication, `cafc` typically generates calls to a one-sided communication library; however, it can also generate code that uses load and store instructions to access remote data on shared-memory systems.

`cafc` uses OPEN64/SL [101] as the front-end. The OPEN64/SL project at Rice produced a nearly production-quality, multi-platform CAF/Fortran 95 front-end suitable for source-to-source transformations. We have ported it to machines running Linux, Irix, Tru64, and Solaris.

The engineering effort to port `cafc` to a new architecture is minor. First, `cafc` requires OPEN64/SL that can be compiled with the GNU compilers available on most architectures. Second, it requires a one-sided communication library such as ARMCI [83] and GASNet [17] that is (or will be) available on most parallel machines. Finally, it must

be able to manipulate the Fortran 90 array descriptor (dope vector) of the target back-end Fortran 95 compiler to manage and access co-array memory outside of Fortran 95 run-time system to deliver best performance.

As of this writing, `cafc` supports COMMON, SAVE, ALLOCATABLE, and formal parameter co-arrays of primitive and user-defined types, co-arrays with multiple co-dimensions, co-array communication using array sections, CAF synchronization primitives, and most of CAF intrinsic functions defined in the CAF specification [88]. Parallel I/O is not supported. Support of allocatable co-array components was implemented using ARMCI's Global Procedure Calls (GPC). `cafc` compiles natively and runs on the following architectures: Pentium clusters with Ethernet interconnect, Itanium2 clusters with Myrinet or Quadrics interconnect, Alpha clusters with Quadrics interconnect, SGI Origin 2000, and SGI Altix 3000.

4.1.1 Memory management

To support efficient access to remote co-array data on the broadest range of platforms, memory for co-arrays is managed by the communication substrate separately from memory managed conventionally by a Fortran 95 compiler's language run-time system. Having the communication substrate control allocation of co-array memory enables our generated code to use the most appropriate allocation strategy for that platform. For instance, on a Myrinet 2000-based cluster, `cafc` generates code that allocates data for co-arrays in pinned physical memory; this enables the communication library to perform data transfers on the memory directly, using the Myrinet adapter's DMA engine.

4.1.2 Co-array descriptors and local co-array accesses

For CAF programs to perform well, access to local co-array data must be efficient. Since co-arrays are not supported in Fortran 95, `cafc` needs to translate references to the local portion of a co-array into valid Fortran 95 syntax. For performance, generated code must be amenable to back-end compiler optimization. In an earlier study [48], we explored several

alternative representations for co-arrays. Our current strategy is to use a Fortran 95 pointer to access local co-array data.

In `cafc`'s generated code, co-arrays are represented using *co-array descriptors*, which reside in the local memory of each process image. A co-array descriptor structure contains two components. One component is a Fortran 90 pointer (a deferred shape array) used to directly access the local portion of the co-array's data. The second component is an opaque handle (an integer of sufficient length to store a pointer) that represents any underlying state for a co-array maintained by the communication substrate. For example, to represent a three-dimensional SAVE or COMMON co-array of real numbers, `cafc` generates a descriptor such as the one shown here:

```
Type CoArrayDescriptor_Real8_3
   integer(ptrkind) :: handle
   real(kind=8), pointer:: ptr(:,:,:)
End Type CoArrayDescriptor_Real8_3
```

A co-array's `handle` refers to a run-time representation that contains the the size of co-array and additional information to locally compute the base virtual address of the co-array on each process image to use RDMA capabilities without the need to contact a remote image. Co-array shape and co-shape are not represented explicitly in the run-time layer.

Since co-array data is allocated outside the Fortran 95 run-time system, `cafc` needs to initialize and manipulate compiler-dependent Fortran 95 array descriptors (dope vectors) on a variety of target platforms. For historical reasons, we use our own multi-compiler library for this purpose. Alternatively, it is possible to employ the CHASM library [98] from Los Alamos National Laboratory. CHASM is a tool to improve C++ and Fortran 90 interoperability. CHASM supplies a C++ array descriptor class which provides an interface between C and Fortran 95 arrays. This allows arrays to be created in one language and then passed to and used by the other language.

4.1.3 Co-array parameters

CAF allows programmers to pass co-arrays as arguments to procedures. According to the CAF specification [87, 88], there are two types of co-array argument passing: by-value and by-co-array.

To use by-value parameter passing of a co-array, one wraps a co-array actual parameter in an additional set of parentheses, *e.g.*, `call foo((ca(1:n,k)[p]))`. In this case, the CAF compiler first allocates a local temporary to hold the value of the remote co-array section (`ca(1:n,k)` from processor `p`) for the duration of the call. Next, it fetches the remote section from processor `p`. Then, it invokes the procedure. After the procedure returns, the temporary is freed.

The pass by-co-array convention, e.g, `call foo(ca(i,k))`, has semantics similar to Fortran's by-reference parameter passing convention: only the local address of `ca(i,k)` is passed down to the subroutine. Each co-array dummy argument to a procedure is declared as an explicit-shape co-array within the procedure. It is illegal to pass a remote co-array element by-co-array, *e.g.*, `call foo(ca(i,k)[p])`. It is also illegal to pass a co-array section to a subroutine since this might require copy-in-copy-out semantics; this would interfere with memory consistency across procedure calls. `cafc` converts each dummy argument $\mathcal{P}$ passed by-co-array into two parameters: $\mathcal{L}$ — local portion of the co-array — and $\mathcal{H}$ — the co-array handle. As part of the translation, all local references to the dummy argument $\mathcal{P}$ within the procedure are replaced by references to $\mathcal{L}$, while remote references through dummy argument $\mathcal{P}$ use $\mathcal{H}$ to communicate data.

`cafc` also supports a pass by-reference convention with an explicit interface, in which the callee receives the local part of a co-array as an array argument and treats it as a regular array. This allows the programmer to reuse subroutines that compute over arrays for processing local parts of co-arrays. Fortran 90 interfaces are used to differentiate what type of calling convention should be used. An example is shown in Figure 4.1.

When declaring a procedure interface that receives a co-array by-reference, the dummy argument's shape (and co-shape) information may be omitted. A callee receiving a co-

```
interface
  subroutine foo(a)
    double precision a[*]
  end subroutine foo
  subroutine bar(b)
    double precision b
  end subroutine bar
end interface
double precision x(10,10)[5,*]

call foo(x(i,j))  ! pass by-co-array
call bar(x(i,j))  ! pass by-reference
```

Figure 4.1 : Using Fortran 90 interfaces to specify by-co-array and by-reference argument passing styles.

array argument declares fresh shape and co-shape information; this can be used to reshape a co-array in the callee if desired.

4.1.4 COMMON and SAVE co-arrays

CAF explicitly supports sequence association between local parts of co-arrays in COMMON blocks. Using Fortran EQUIVALENCE statements to associate co-array and non-co-array memory is prohibited. Because of this restriction, `cafc` is able to split a COMMON block containing both co-array and local variables into two separate COMMON blocks: one containing only local variables and the other containing only co-array variables. The latter co-array COMMON block is handled as described below.

Managing co-array memory outside of the Fortran 95 run-time subsystem requires special mechanisms for allocating and initializing SAVE and COMMON co-array variables. During translation, `cafc` replaces declarations of static co-arrays with descriptors for the separately allocated co-array storage. When `cafc`-generated code begins execution, it performs a two-step initialization process. First, it allocates storage for co-arrays. Second, it initializes procedure-level views of SAVE and COMMON co-arrays by associating co-array descriptors with the allocated memory and the communication substrate's run-time state.

For each procedure containing SAVE co-arrays, `cafc` generates an initialization routine that allocates memory for each SAVE co-array and sets up a descriptor for the co-array.

`cafc` generates an allocator procedure for each co-array COMMON block. An allocator for a co-array COMMON block reserves a contiguous chunk of storage for the COMMON block's set of co-arrays at program launch. Since different procedures may declare different layouts for the same COMMON block, which we call *views*, `cafc` synthesizes one view initializer per procedure per COMMON block. Each view initializer is invoked once at program launch after storage allocation to fill in a procedure-private copy of a co-array descriptor for each co-array in the procedure's view of the common block.

When linking a CAF program, `cafc` first examines the object files to collect the names of all storage allocators and co-array descriptor and view initializers. Next, `cafc` synthesizes a global initializer that calls each allocator and initializer. The global initializer is called once at program launch before any user-written code executes.

4.1.5 Procedure splitting

SAVE and COMMON co-arrays are static and their properties are known to a CAF compiler at compile-time. These properties include: co-array bounds, the fact that memory occupied by a co-array is contiguous, and the lack of aliasing among such co-arrays. After `cafc` translates a CAF program, a SAVE or COMMON co-array `a` is represented with a co-array descriptor; so local co-array accesses are done via Fortran 90 pointer in the generated code. Such accesses are difficult to optimize by the back-end compiler because the information about bounds, contiguity, and lack of aliasing is not readily available for Fortran 90 pointers. These properties were lost in translation and the back-end compiler must rediscover them to produce fast code.

Consider the following lines that are common to both the CAF and Fortran+MPI versions of the `compute_rhs` subroutine of the NAS BT benchmark.

```
rhs(1,i,j,k,c) = rhs(1,i,j,k,c) + dx1tx1 *  &
   (u(1,i+1,j,k,c) - 2.0d0*u(1,i,j,k,c) +   &
   u(1,i-1,j,k,c)) -                        &
   tx2 * (u(2,i+1,j,k,c) - u(2,i-1,j,k,c))
```

`u` and `rhs` reside in a single COMMON block in both sources. The CAF and Fortran+MPI versions of the program declare identical data dimensions for these variables, except that the CAF code adds a single co-dimension to `u` and `rhs` by appending a "`[*]`" to the end of their declarations. After translation, `cafc` rewrites the declarations of the `u` and `rhs` co-arrays with co-array descriptors that use a deferred-shape representation for co-array data. References to `u` and `rhs` are rewritten to use Fortran 95 pointer notation as shown here:

```
rhs%ptr(1,i,j,k,c) = rhs%ptr(1,i,j,k,c) + dx1tx1 * &
   (u%ptr(1,i+1,j,k,c) - 2.0d0*u%ptr(1,i,j,k,c) +  &
   u%ptr(1,i-1,j,k,c)) -                           &
   tx2 * (u%ptr(2,i+1,j,k,c) - u%ptr(2,i-1,j,k,c))
```

Our experimentation with several back-end Fortran 90 compilers showed that performance of CAF codes with Fortran 90 pointers is up to 30% inferior to that of equivalent MPI codes that use SAVE or COMMON variables. The main reason is that Fortran 90 pointer-based representation complicates the alias analysis in the back-end compiler. In turn, this precludes important loop optimizations. Some Fortran 95 compilers accept a compile-time flag indicating the lack of aliasing among Fortran 95 pointers, but despite using the flag, `cafc`-translated codes showed slower node performance than their MPI+Fortran counterparts.

We addressed this problem by automatically converting Fortran 90 pointers of SAVE and COMMON co-array representation into explicit-shape procedure arguments, which are contiguous and do not alias. Bounds of SAVE and COMMON co-arrays are constant, and, thus, the argument bounds can be redeclared in a procedure. As the result, the properties of SAVE and COMMON co-arrays are conveyed to the back-end Fortran 95 compiler.

We named the transformation *procedure splitting*. `cafc` splits each procedure that references SAVE and COMMON co-arrays into two subroutines: an inner procedure and an outer procedure. The transformation is applied prior to any other transformation involv-

```
subroutine f(a,b)
real a(10)[*], b(100), c(200)[*]
save c
... = c(50) ...
end subroutine f
```

(a) Original procedure

```
subroutine f(a,b)
real a(10)[*], b(100), c(200)[*]
save c
interface
  subroutine f_inner(a,b,c_arg)
    real a[*], b, c_arg[*]
  end subroutine f_inner
end interface
call f_inner(a,b,c)
end subroutine f

subroutine f_inner(a,b,c_arg)
real a(10)[*], b(100), c_arg(200)[*]
... = c_arg(50) ...
end subroutine f_inner
```

(b) Outer and inner procedures after splitting.

Figure 4.2 : Procedure splitting transformation.

ing co-arrays. Pseudo-code in Figure 4.2 illustrates the effect of the procedure-splitting transformation.

The outer procedure retains the same interface as the original one. It does not perform any computation of the original procedure. Its purpose is to declare original parameters and the inner-procedure interface and to call the inner procedure, passing the arguments. The inner procedure performs the computation of the original one and is created by applying three changes to the original procedure. First, its argument list is extended with parameter co-arrays corresponding to the SAVE and COMMON co-arrays referenced by the original procedure. Second, explicit-shape co-array declarations are added for each additional co-array argument. Third, each reference to any SAVE or COMMON co-array is rewritten with the reference to the corresponding co-array parameter. Figure 4.2 shows the effect of

rewriting the reference to `c(50)` in `f` with a reference to `c_arg(50)` in `f_inner`.

After the translation process, parameter co-arrays become explicit-shape Fortran 95 dummy parameters, which do not alias according to Fortran 95 specification. The final result is that SAVE and COMMON co-arrays within the inner procedure are now dummy arguments represented using explicit-shape arrays rather than deferred-shape arrays. Therefore, the back-end compiler is conveyed the lack of aliasing among SAVE and COMMON co-arrays, their bounds, and their contiguity. Better aliasing information leads to more precise dependence analysis and more aggressive loop optimizations. Knowing bounds at compile-time may reduce register pressure. Knowing that referenced memory is contiguous might improve software prefetching. While the procedure-splitting transformation introduces extra procedure calls and slightly increases the code size, we have not observed that it decreases performance. The procedure-splitting transformation allows many codes, especially the ones with complex dependence patterns, to achieve the same level of scalar performance as that of their MPI+Fortran counterparts [47, 48].

4.1.6 Multiple co-dimensions

The CAF programming model does not limit the programmer to using a flat co-shape. Instead, the user can specify a multi-dimensional co-shape, with the same column-major convention as regular Fortran code. This feature is of most use when the processor topology of a problem is logically mapped onto a Cartesian processor grid without periodic boundaries. The programmer has the ability to mold the co-shape to fit the logical processor grid. Indexing of a multi-dimensional organization of remote images is then straightforward using this feature.

Let us consider a general co-shape definition, $[\mathtt{lb}_1 : \mathtt{ub}_1, \mathtt{lb}_2 : \mathtt{ub}_2, \ldots, \mathtt{lb}_n : \mathtt{ub}_n, \mathtt{lb}_{n+1} : *]$. For SAVE and COMMON co-arrays the co-shape must be specified using exclusively constants. A remote reference to $[\mathtt{i}_1, \mathtt{i}_2, \ldots, \mathtt{i}_n, \mathtt{i}_{n+1}]$ corresponds to processor image $\sum_{j=1}^{n+1} (\mathtt{i}_j - \mathtt{lb}_j) * \mathtt{m}_j$, where

$$\mathtt{m}_1 = 1 \tag{4.1}$$

$$\mathtt{m}_\mathtt{j} = \prod_{\mathtt{k}=1}^{\mathtt{j}-1} \left(\mathtt{ub}_\mathtt{k} - \mathtt{lb}_\mathtt{k} + 1\right),\ 2 \leq \mathtt{j} \leq \mathtt{n}+1 \tag{4.2}$$

In order to support co-arrays with multiple co-dimensions, we augment the co-array metadata used in `cafc`-generated code with several co-shape variables. For a co-array `a` with the co-shape definition $[\mathtt{lb}_1 : \mathtt{ub}_1, \mathtt{lb}_2 : \mathtt{ub}_2, \ldots, \mathtt{lb}_\mathtt{n} : \mathtt{ub}_\mathtt{n}, \mathtt{lb}_{\mathtt{n}+1} : *]$, we add the following variables:

- `a_coLB_i`, for $1 \leq \mathtt{i} \leq \mathtt{n}+1$
- `a_coUB_i`, for $1 \leq \mathtt{i} \leq \mathtt{n}$
- `a_ThisImage_i`, for $1 \leq \mathtt{i} \leq \mathtt{n}+1$
- `a_CoIndexMultiplier_i`, for $1 \leq \mathtt{i} \leq \mathtt{n}+1$
- `a_ThisImageVector`

`a_coLB_i`, `a_coUB_i` and `a_CoIndexMultiplier_i` correspond directly to $\mathtt{lb}_\mathtt{i}$, $\mathtt{ub}_\mathtt{i}$ and $\mathtt{m}_\mathtt{i}$. `a_ThisImage_i` and `a_ThisImageVector` are used to precompute the values returned by the CAF intrinsic function `this_image`. According to the CAF specification, `this_image(a,i)` returns the i-th co-space coordinate for `a` on the corresponding process image. This value is precomputed in `a_ThisImage_i`. `this_image(a)` returns a vector containing the values of `this_image(a,i)` for all the co-dimensions of co-array `a`. This vector is thus precomputed in `a_ThisImageVector`.

We extend the initialization routines mentioned above to set up the co-shape metadata variables. `a_coLB_i`, `a_coUB_i` are trivially assigned using the co-array definition. The variables `a_CoIndexMultiplier_i` are computed iteratively using Formulas 4.1 and 4.2 for `m_i`. To compute `a_ThisImage_i` we use the process image index returned by `this_image` as follows:

$\texttt{a\_ThisImage\_i} = \texttt{mod}(\texttt{div}(\texttt{this\_image}() - 1, \texttt{m}_\texttt{i}), (\texttt{ub}_\texttt{i} - \texttt{lb}_\texttt{i} + 1)) + \texttt{lb}_\texttt{i}$, for $\texttt{i} = 1..\texttt{n}$

$\texttt{a\_ThisImage\_i} = \texttt{div}(\texttt{this\_image}() - 1, \texttt{m}_\texttt{i}) + \texttt{lb}_\texttt{i}$, for $\texttt{i} = \texttt{n} + 1$

Note that a dead-code eliminator would remove unused co-shape variables generated for dummy co-arrays. When generating code that computes the remote image number, the CAF compiler replaces the multipliers by constants whenever possible.

One immediate consequence of the above scheme is that we can support co-shape reshaping during argument passing. `cafc` allows co-shapes of dummy co-array arguments to be declared using specification expressions rather than only constants. The co-lower and co-upper bounds variables are initialized by the corresponding specification expressions; the rest of the computation to determine the "coIndexMultiplier" variables, the components of "this image" variables, and the "this image vector" is performed as above. This extension enables programmers to express processing on co-array arguments with variable co-spaces, leading to more general code.

4.1.7 Intrinsic functions

`cafc` supports the CAF intrinsic functions: `log2_images()`, `this_image()`, `num_images()`, and `rem_images()`. To implement them efficiently, we precompute their values at program launch and store them into scalars. At compile time, calls to these functions are replaced by references to the corresponding scalars. A more complicated strategy is employed to support `this_image(a)` evaluated for a co-array `a`. We compute the components of `this_image` once at program initialization for SAVE and COMMON co-arrays, and once per procedure invocation for dummy co-arrays. We replace calls to `this_image(a)` with a reference to `a_ThisImageVector`. We replace calls to `this_image(a,i)` with a scalar variable if `i` is a compile-time constant, and if `i` is a variable, we use an array reference into `a_ThisImageVector`.

4.1.8 Communication code generation

Communication events expressed with CAF's bracket notation must be converted into Fortran 95; however, this is not straightforward because the remote memory may be in a different address space. Although CAF provides shared-memory semantics, the target architecture may not; a CAF compiler must perform transformations to bridge this gap.

Shared-memory machines

On a hardware shared-memory platform, the transformation is relatively straightforward, since references to remote memory in CAF can be expressed as loads and stores to shared locations. With proper initialization, Fortran 90 pointers can be used to directly address non-local co-array data. The CAF run-time library provides the virtual address of a co-array on remote images; this is used to set up a Fortran 90 pointer for referencing the remote co-array. An example of this strategy is presented in Figure 4.3 (a) for the following code.

```
DO J=1, N
   C(J)=A(J)[p]
END DO
```

The generated code accesses remote data by dereferencing a Fortran 90 pointer, for which Fortran 95 compilers generate loads/stores. In Figure 4.3 (a), the procedure `CafSetPtr` sets up the pointer and is called for every access; this adds significant overhead. Hoisting pointer initialization outside the loop as shown in Figure 4.3 (b) can substantially improve performance. To perform this optimization automatically, `cafc` needs to determine that the process image index for a non-local co-array reference is loop invariant.

```
DO J=1,N
  ptrA=>A(J)
  call CafSetPtr(ptrA,p,A_h)
  C(J)=ptrA
END DO
```

(a) Fortran 90 pointer to remote data

```
ptrA=>A(1:N)
call CafSetPtr(ptrA,p,A_h)
DO J=1,N
  C(J)=ptrA(J)
END DO
```

(b) Hoisted Fortran 90 pointer initialization

Figure 4.3 : Fortran 90 pointer access to remote data.

In comparison to general library-based communication for cluster architectures, load/store communication avoids unnecessary overhead due to library calls. This is especially beneficial for applications with fine-grain communication. Our study [48] contains a detailed exploration of the alternatives for performing communication on hardware shared-memory systems. However, the load/store strategy cannot be used for cluster-based systems with distributed memory.

Cluster architectures

To perform data movement on clusters, `cafc` must generate calls to a communication library to access data on a remote node. Moreover, `cafc` must manage storage to temporarily hold remote data needed for a computation. For example, in the case of a read reference of a co-array on another image, `arr(:)=coarr(:)[p]+...`, a temporary, `temp`, is allocated just prior to the statement to hold the value of the `coarr(:)` array section from image p. Then, a call to get data from image p is issued to the run-time library. The statement is rewritten as `arr(:)=temp(:)+...`. The temporary is deallocated immediately after the statement. For a write to a remote image, such as `coarr(:)[p1,p2]=...`, a temporary `temp` is allocated prior to the remote write statement; the result of the evaluation of the right-hand side is stored in the temporary; a call to a communication library is issued to perform the write; and finally, the temporary is deallocated. When possible, the generated code avoids using unneeded temporary buffers. For example, for an assignment performing a co-array to co-array copy, `cafc` generates code to move the data directly from the source into the destination.

Currently, `cafc` generates blocking communication operations. In our study [47], we introduced non-blocking communication hints that enable `cafc` to exploit non-blocking PUTs.

Hints for non-blocking communication

Overlapping communication and computation is an important technique for hiding interconnect latency as well as a means for tolerating asynchrony between communication partners. However, as CAF was originally described [88], all communication must complete before each procedure call in a CAF program. In a study of our initial implementation of `cafc`, we found that obeying this constraint and failing to overlap communication with independent computation hurt performance [30].

Ideally, a CAF compiler could always determine when it is safe to overlap communication and computation and to generate code automatically that does so. However, it is not always possible to determine at compile time whether a communication and a computation may legally be overlapped. For instance, if the computation and/or the communication use indexed subscripts, making a conservative assumption about the values of indexed subscripts may unnecessarily eliminate the possibility of communication/computation overlap. In the presence of separate compilation, a CAF compiler cannot determine whether it is legal to overlap communication with a called procedure without whole-program analysis.

To address this issue, we believe it is useful to provide a mechanism to enable knowledgeable CAF programmers to provide hints as to when communication may be overlapped with computation. Such a mechanism serves two purposes: it enables overlap when conservative analysis would not, and it enables overlap in `cafc`-generated code today before `cafc` supports static analysis of potential communication/computation overlap. While exposing the complexity of non-blocking communication to users is not ideal, we believe it is pragmatic to offer a mechanism to avoid performance bottlenecks rather than forcing users to settle for lower performance.

To support communication/computation overlap in code generated by `cafc`, we implemented support for three intrinsic procedures that enable programmers to demarcate the initiation and signal the completion of non-blocking PUTs. We use a pair of intrinsic calls to instruct the `cafc` run-time system to treat all PUT operations initiated between them as non-blocking. We show this schematically below.

```
region_id =  open_nb_put_region()
...
Put_Stmt_1
...
Put_Stmt_N
...
call close_nb_put_region(region_id)
```

Only one non-blocking region may be open at any particular point in a process image's execution. Each PUT operation that executes when a non-blocking region is open is associated with the `region_id` of the open non-blocking region. It is a run-time error to close any region other than the one currently open. Eventually, each non-blocking region that was initiated must be completed with the call shown below.

```
call complete_nb_put_region(region_id)
```

The completion intrinsic causes a process image to wait at this point until the completion of all non-blocking PUT operations associated with `region_id` that the process image initiated. It is a run-time error to complete a non-blocking region that is not currently pending completion.

Using these hints, the `cafc` run-time system can readily exploit non-blocking communication for PUTs and overlap communication with computation. Overlapping GET communication associated with reads of non-local co-array data with computation would also be useful. We are currently exploring how one might sensibly implement support for overlapping GET communication with computation, either by initiating GETs early or delaying computation that depends upon them.

Alternative code generation strategy for PUTs

It would be interesting to consider another code generation strategy that enables non-blocking PUTs and does not require the programmer to provide hints. The strategy is similar to the one described in Section 4.1.8 for distributed-memory machines. `cafc` can always allocate a temporary to store the result of the right-hand side (RHS) of a remote co-array assignment statement. The data movement can be initiated as a non-blocking PUT

operation. It is safe to proceed with the execution of the next statement right after the PUT because the RHS value resides in a temporary that is not accessible by the user code. Deallocation of this temporary must be delayed until the PUT completes. The run-time layer can keep track of all such PUTs "in flight".

PUTs can be completed safely in several ways. First, the underlying communication layer can invoke a callback indicating PUT completion, if such a mechanism is supported. This callback can deallocate the temporary. Second, all PUTs must be completed before the program executes a synchronization statement. Temporaries can be deallocated at this time. Third, the run-time layer can limit the number of non-completed PUTs and complete them in first-in-first-out order, deallocating corresponding temporaries in addition. Fourth, the run-time layer can complete some outstanding PUTs and deallocate corresponding temporaries if the system is low on memory.

The only case when using a temporary may be avoided on a distributed memory architecture is for a co-array assignment in which the right hand side is a variable, *e.g.*, `a(:)[p] = b(:)`. If `b` is not modified by local computation on all CFG paths from the assignment statement to a synchronization statement (barrier or `notify(p)`), the PUT can communicate data in-place and be non-blocking.

4.1.9 Allocatable and pointer co-array components

CAF provides allocatable and pointer co-array components to support asymmetric data structures. For example, an allocatable component `a%ptr(:)` might have different sizes on different images or might not be allocated at all on some images. The current version of `cafc` provides allocatable co-array components only for the ARMCI substrate with support for Global Procedure Calls (GPCs) (see Section 3.2). We use GPCs as Active Messages (AMs) [122] and refer to them as such in the following discussion.

Memory management for co-array components

The allocation and deallocation of `a%ptr` are local operations. For the most efficient remote data accesses, it is necessary to allocate components in special memory managed by the underlying communication library. `cafc`'s run-time layer uses `ARMCI_Malloc_local` for this purpose. `cafc` implements deallocation using `ARMCI_Free_local`.

Accesses to remote co-array components

`a[p]%ptr1(i)%ptr2(:)` is an example of an access to a co-array component. The first reference of the chain (referred to as the head) is the co-array reference — `a[p]`. It determines the process image `p` relative to which the rest of the chain (referred to as the tail) is dereferenced. The brackets are allowed only for the head reference. A local reference does not have brackets.

`cafc` uses the same code generation strategy for local co-array accesses and for local co-array component references. For example, a reference `a(i)%ptr1(:)` is rewritten as `a_desc%ptr(i)%ptr1(:)`, where `a_desc` is the co-array `a` descriptor.

However, remote accesses to co-array components are different from those to co-arrays. A co-array access, *e.g.*, `b(i,:)[p]`, refers to a strided memory section S, local or remote. The start address and shape of S can always be computed on the process image `q` executing the access. `cafc` can use strided PUT/GET to access S without the need to contact `p`. On the contrary, the start address and shape of the memory section corresponding to a co-array component reference, *e.g.*, `a[p]%ptr1(i)%ptr2(:)`, may not be known locally on `q`. There are two feasible approaches to support remote co-array component accesses.

The first uses GET to dereference the chain level by level. For example, `q` can GET the remote dope vector `a[p]%ptr1`, determine the remote start address of `a[p]%ptr1(i)`, and GET the next level dope vector `a[p]%ptr1(i)%ptr2`. `a[p]%ptr1(i)%ptr2` determines the start address and shape of the remote co-array component reference `a[p]%ptr1(i)%ptr2(:)`; it is a strided memory section W. Only after that, the refer-

ence data W can be updated via PUT or fetched via GET. The disadvantage of this approach is that it exposes communication latency for each level of dereferencing; however, it may be the only feasible option for architectures without AM support.

An alternative is to use an AM to obtain the start address and shape of W in one step. An AM executed on `p` has access to `p`'s memory and can dereference the chain as a local reference `a%ptr1(am_i)%ptr2(:)`, where `am_i` is the value of subscript `i` on q. This approach is more efficient than the first one because it needs only one network message to obtain the parameters of W; however, it requires AM support.

`cafc` implements the AM-based strategy using ARMCI with GPC support. It synthesizes an accessor Fortran subroutine for each remote co-array component reference R. The accessor is called inside the AM handler on `p` and is given a vector of subscripts `sv`. `sv` contains the values of non-constant and non-implicit subscripts of R; *e.g.*, for `a(i,j+1)[p]%ptr1(5)%ptr2(k1:k2,7:)`, `sv` contains, in order, the values of `i`, `j+1`, `k1`, and `k2` evaluated on `q`. The accessor subroutine computes the start address and the extents of the `a(sv(1),sv(2))%ptr1(5)%ptr2(sv(3):sv(4),7:)` reference in the process image `p` address space; it uses the Fortran `SIZE` intrinsic[1]. It also computes the extents of the reference dope vector in the chain — `a(sv(1),sv(2)+1)%ptr1(5)%ptr2`. A relevant code fragment is shown below:

```
commShape => a(sv(1),sv(2))%ptr1(5)%ptr2(sv(3):sv(4),7:)
addr = loc(commShape(1,1))  ! the start address of the section
shp(1) = SIZE(a(sv(1),sv(2))%ptr1(5)%ptr2, 1)
shp(2) = SIZE(commShape, 1)
shp(3) = SIZE(a(sv(1),sv(2))%ptr1(5)%ptr2, 2)
shp(4) = SIZE(commShape, 2)
```

The AM returns the start address `addr` and the vector `shp`, containing extents, to process image `q`. Using the address and extents, `q` can compute the strided memory section W similarly to how a Fortran 95 compiler computes a strided section for a pointer `ptr=>a(...)` operation. Knowing the remote memory parameters, `cafc` (1) allocates

[1] `SIZE(a,i)` returns the extent of an array section `a` for dimension `i`.

the temporary of proper size to hold off-processor data, (2) instructs ARMCI to PUT or GET data (in the same way as done for a co-array access).

The described AM-based approach uses two communication operations: one to obtain the remote memory section parameters, the other to transfer data via PUT/GET. It might be possible to combine these two messages into one. For a PUT, the data and access parameters can be sent together in one AM, usually if the data size is not very large; inside the accessor subroutine, the data must be copied into the remote memory section. For a GET, if the size of transmitted data can be inferred locally, the temporary to hold off-processor data can be allocated without the first AM. The address of this temporary and access parameters can be sent in a request AM. A reply AM can return the requested data in its payload and copy the data into the temporary. The current implementation of `cafc` does not support this optimization.

4.2 Experimental evaluation

We ported numerous parallel benchmarks and real applications into CAF and performed extensive evaluation studies [30, 47, 48, 31, 32] to identify the sources of inefficiencies and performance bottlenecks on a range of modern parallel architectures. Among the ported applications (see Section 3.4) are NAS MG, CG, SP, BT, LU and EP Parallel Benchmarks [12], ASCI Sweep3D [4], RandomAccess and STREAM HPC Challenge Benchmarks [1], Spark98 [90], LBMHD [91], Parallel Ocean Program (POP) [114] and Jacobi iteration. These codes are widely recognized as useful for evaluation of parallel programming models.

Our studies [30, 47, 48, 31, 33] have shown that even without automatic communication optimizations CAF codes compiled with `cafc` can match the performance and scalability of their MPI counterparts on a range of cluster and hardware shared-memory systems. Among these codes are regular and irregular parallel benchmarks and applications such as the NAS benchmarks, Sweep3D, Spark98, and STREAM. We now briefly overview the main conclusions of these studies and present selected results.

4.2.1 Co-array representation and local accesses

We investigated different co-array representations for local and remote accesses across a range of architectures and back-end compilers. The result of this study [48] is that it is acceptable to represent co-arrays as Fortran 90 pointers and use procedure splitting for local accesses; however, `cafc` should use different communication strategies for cluster and shared-memory architectures.

4.2.2 Communication efficiency

Cluster architectures offer only one option to communicate data — to use a one-sided communication library. On shared-memory architectures, it is also possible to access remote data using hardware load and store instructions via Fortran 90 pointer dereferencing. Load/store communication has two advantages over the library-based communication. First, it avoids temporaries to hold off-processor data necessary for computation and enables utilizing both local memory and interconnect bandwidth. Second, for benchmarks requiring fine-grain communication, such as RandomAccess, the load/store code generation strategy avoids expensive function calls and provides better performance. In contrast, coarse-grain communication is more efficient when implemented using an architecture-tuned memory copy routine for bulk data movement rather than direct load/store.

Communication generation for generic parallel architectures

To access data residing on a remote node, `cafc` generates ARMCI (or GASNet) calls. Unless the statement causing communication is a simple copy, temporary storage is allocated to hold non-local data.

Consider the statement `a(:)=b(:)[p]+...`, which reads co-array data for `b` from another process image. First, `cafc` allocates a temporary, `b_temp`, just prior to the statement to hold the value of `b(:)` from image `p`. `cafc` adds a GET operation to retrieve the data from image `p`, rewrites the statement as `a(:)=b_temp(:)+...` and inserts code to deallocate `b_temp` after the statement. For a statement containing a co-array write to a

```
DO J=1, N
   C(J)=A(J)[p]
END DO
```

(a) Remote element access

```
DO J=1,N
  call CafGetScalar(A_h, A(J), p, tmp)
  C(J)=tmp
END DO
```

(b) General communication code

Figure 4.4 : General communication code generation.

remote image, such as `c(:)[p]=...`, `cafc` inserts allocation of a temporary `c_temp` prior to the statement. Then, `cafc` rewrites the statement to store its result in `c_temp`, adds a PUT operation after the statement to perform the non-local write, and inserts code to deallocate `c_temp`. An example of this translation strategy is shown in Figure 4.4.

Communication generation for shared-memory architectures

Library-based communication adds unnecessary overhead for fine-grain communication on shared-memory architectures. Loads and stores can be used to directly access remote data more efficiently. Here we describe several representations for fine-grain load/store access to remote co-array data.

Fortran 90 pointers. With proper initialization, Fortran 90 pointers can be used to directly address non-local co-array data. The CAF run-time library provides the virtual address of a co-array on remote images; this is used to set up a Fortran 90 pointer for referencing the remote co-array. An example of this strategy is presented in Figure 4.5 (a). The generated code accesses remote data by dereferencing a Fortran 90 pointer, for which Fortran 90 compilers generate direct loads and stores. In Figure 4.5 (a), the procedure `CafSetPtr` is called for every access; this adds significant overhead. Hoisting pointer initialization outside the loop as shown in Figure 4.5 (b) can substantially improve performance. To perform this optimization automatically, `cafc` needs to determine that the process image index for a non-local co-array reference is loop invariant.

```
DO J=1,N
  ptrA=>A(J)
  call CafSetPtr(ptrA,p, A_h)
  C(J)=ptrA
END DO
```

(a) Fortran 90 pointer to remote data

```
ptrA=>A(1:N)
call CafSetPtr(ptrA,p,A_h)
DO J=1,N
  C(J)=ptrA(J)
END DO
```

(b) Hoisted Fortran 90 pointer initialization

Figure 4.5 : Fortran 90 pointer access to remote data.

```
POINTER(ptr, ptrA)
...
DO J=1,N
  ptr = shmem_ptr(A(J), p)
  C(J)=ptrA
END DO
```

(a) Cray pointer to remote data

```
POINTER(ptr, ptrA)
...
ptr = shmem_ptr(A(1), p)
DO J=1,N
  C(J)=ptrA(J)
END DO
```

(b) Hoisted Cray-pointer initialization

Figure 4.6 : Cray pointer access to remote data.

Vector of Fortran 90 pointers. An alternate representation that doesn't require pointer hoisting for good performance is to precompute a vector of remote pointers for all the process images per co-array. This strategy should work well for parallel systems of modest size. Currently, all shared-memory architectures meet this requirement. In this case, the remote reference in the code example from Figure 4.4 (a) would become:
`C(J) = ptrArrayA(p)%ptrA(J)`.

Cray pointers. We also explored a class of shared-memory code generation strategies based on the SHMEM library. After allocating shared memory with `shmalloc`, one can use `shmem_ptr` to initialize a Cray pointer to the remote data. This pointer can then be used to access the remote data. Figure 4.6 (a) presents a translation of the code in Figure 4.4 using `shmem_ptr`. Without hoisting the pointer initialization as shown in Figure 4.6 (b), this code incurs a performance penalty similar to the code shown in Figure 4.5 (a).

Fine-grain applications on shared-memory architectures

In our study [48], we evaluated the quality of source-to-source translation for applications where fine-grain accesses are preferred due to the nature of the application. Previous studies have shown the difficulty of improving the granularity of fine-grain shared-memory applications [125]. We use the RandomAccess benchmark, described in Section 3.4, as an analog of a complex fine-grain application.

The results of RandomAccess with different co-array representations and code generation strategies are presented in Table 4.1 for the SGI Origin 2000 architecture and in Table 4.2 for the SGI Altix 3000 architecture. The results are reported in MUPs, 10^6 updates per second, per processor for two main table sizes: 1MB and 256MB per image, simulating an application with modest memory requirements and an application with high memory requirements. All experiments were done on a power-of-two number of processors, so that we can replace `div`s and `mod`s with fast bit operations.

Version	*size per proc = 1MB*					*size per proc = 256 MB*				
# procs.	1	2	4	8	16	1	2	4	8	16
CAF vect. of F90 ptrs.	10.06	1.04	0.52	0.25	0.11	1.12	0.81	0.57	0.39	0.2
CAF F90 pointer	0.31	0.25	0.2	0.16	0.15	0.24	0.23	0.21	0.18	0.12
CAF Cray pointer	12.16	1.11	0.53	0.25	0.11	1.11	0.88	0.58	0.4	0.21
CAF `shmem`	2.36	0.77	0.44	0.25	0.11	0.86	0.65	0.53	0.36	0.19
CAF general comm.	0.41	0.31	0.25	0.2	0.09	0.33	0.3	0.28	0.23	0.14
OpenMP	18.93	1.18	0.52	0.32	0.17	1.1	0.81	0.62	0.45	0.23
MPI bucket 2048	15.83	4.1	3.25	2.49	0.1	1.15	0.85	0.69	0.66	0.1

Table 4.1 : RandomAccess performance on the Origin 2000 in MUPs per processor.

Each table presents results in MUPs per processor for seven variants of RandomAccess. *CAF vector of F90 ptrs.* uses a vector of Fortran 90 pointers to represent co-array data. *CAF F90 pointer* uses Fortran 90 pointers to directly access co-array data. *CAF Cray pointer* uses a vector of integers to store the addresses of co-array data. A Cray pointer is initialized in place to point to remote data and then used to perform an update. *CAF*

Version	*size per proc = 1MB*						*size per proc = 256 MB*					
# procs.	1	2	4	8	16	32	1	2	4	8	16	32
CAF vect. of F90 ptrs.	47.66	14.85	3.33	1.73	1.12	0.73	5.02	4.19	2.88	1.56	1.17	0.76
CAF F90 pointer	1.6	1.5	1.14	0.88	0.73	0.55	1.28	1.27	1.1	0.92	0.74	0.59
CAF Cray pointer	56.38	15.60	3.32	1.73	1.13	0.75	5.14	4.23	2.91	1.81	1.34	0.76
CAF `shmem`	4.43	3.66	2.03	1.32	0.96	0.67	2.57	2.44	1.91	1.39	1.11	0.69
CAF general comm.	1.83	1.66	1.13	0.81	0.63	0.47	1.37	1.34	1.11	0.81	0.73	0.52
OpenMP	58.91	15.47	3.15	1.37	0.91	0.73	5.18	4.28	2.96	1.55	1.17	—
MPI bucket 2048	59.81	21.08	16.40	10.52	5.42	1.96	5.21	3.85	3.66	3.36	3.16	2.88

Table 4.2 : RandomAccess performance on the Altix 3000 in MUPs per processor.

shmem uses `shmem_put` and `shmem_get` functions called directly from Fortran. *CAF general comm.* uses the ARMCI functions to access co-array data. *MPI bucket 2048* is bucketed MPI version with a bucket size of 2048 words. *OpenMP* uses the same fine-grained algorithm as the CAF versions; it uses a private substitution table and performs first-touch initialization of the global table to improve memory locality.

The best representations for fine-grain co-array accesses are the Cray pointer and the vector of Fortran 90 pointers. The other representations, which require a function call for each fine-grain access, yield inferior performance. The *MPI bucket 2048* row is presented for reference and shows that an algorithm with better locality properties and coarser-grain communication clearly achieves better performance. It is worth mentioning that the bucketed MPI implementation is much harder to code compared to a fine-grain CAF version. The OpenMP version of the benchmark performs as well as the best CAF version, due to similar fine-grained access patterns.

Coarse-grain applications on shared-memory architectures

To evaluate our code generation strategy for codes with coarse-grain communication on hardware shared-memory platforms, we selected two benchmarks, MG and SP, from the NAS Parallel Benchmarks [12, 73], widely used for evaluating parallel systems.

We compare four versions of the benchmarks: the standard 2.3 MPI implementation, two compiler-generated CAF versions based on the 2.3 distribution, and the official 3.0 OpenMP [73] versions of SP and MG. *CAF-cluster* uses the Fortran 90 pointer co-array representation and the ARMCI functions that rely on an architecture-optimized memory copy subroutine supplied by the vendor to perform data movement. *CAF-shm* uses the Fortran 90 pointer co-array representation, but uses Fortran 90 pointers to access remote data. The OpenMP version of SP incorporates structural changes made to the 3.0 serial version to improve cache performance on uniprocessor machines, such as fusing loops and reducing the storage size for temporaries; it also uses a 1D strategy for partitioning computation that is better suited for OpenMP.

In the CAF versions, all data transfers are coarse-grain communication arising from co-array section assignments. We rely on the back-end Fortran 90 compiler to scalarize the transformed copies efficiently in *CAF-shm*. Sequential performance measurements used as a baseline were performed using the NPB 2.3-serial release.

For each benchmark, we present the parallel efficiency of the MPI, CAF and OpenMP implementations. On an Altix, we evaluate these benchmarks for both the *single* and *dual* processor configurations. The *single* placement corresponds to running one process per dual-processor node; in the *dual* placement two processes are run on both CPUs of a node, sharing the local memory bandwidth. The experimental results for problem size class C are shown on the Figures 4.7 and 4.8. For SP, both CAF versions achieve similar performance — comparable to the standard MPI versions. For MG, the *CAF-cluster* version performs better than the *CAF-shm* version. Since both versions use coarse-grain communication, the performance difference shows that the architecture-tuned memory-copy subroutine performs better than the compiler scalarized data copy; it effectively hides the interconnect latency by keeping the optimal number of memory operations in flight. The *CAF-cluster* version outperforms the MPI version for both the single and dual configurations. The results for the OpenMP versions are not directly comparable since they are based on version 3.0 source, but they are known to be well designed and tuned for OpenMP execution. The

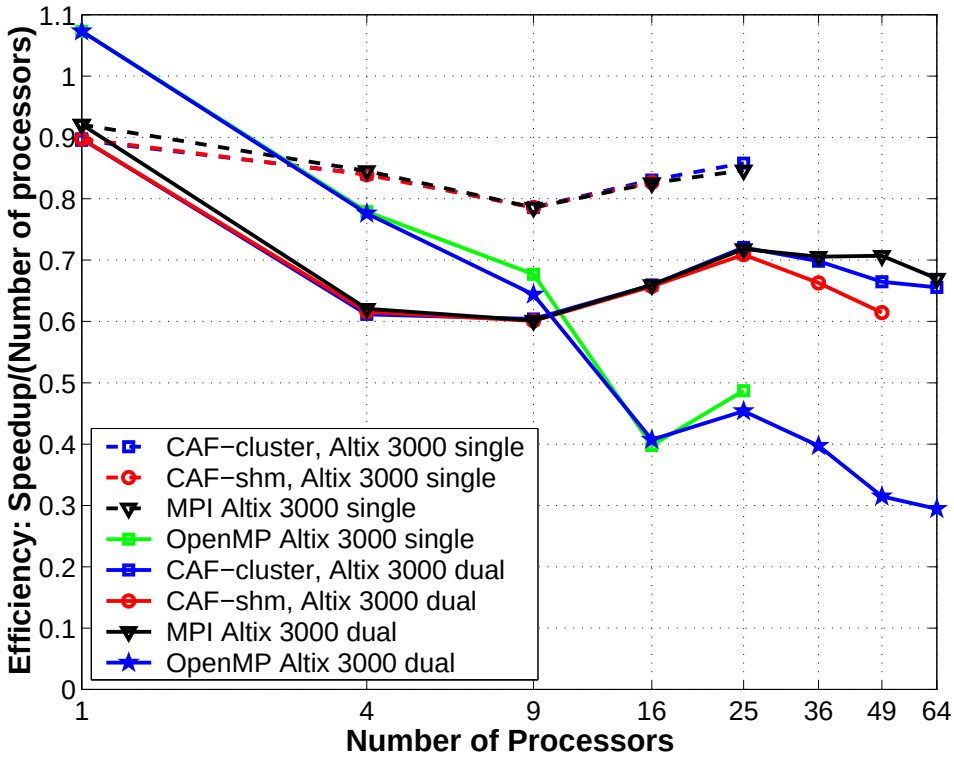


Figure 4.7 : Comparison of parallel efficiencies of the MPI, CAF with general communication, CAF with shared-memory communication, and OpenMP versions of the NAS SP benchmark on an SGI Altix 3000.

OpenMP performance is good for a small number of processors (up to 8-9), but then tails off compared to the MPI and CAF versions.

4.2.3 Cluster architectures

Without efficient communication, a parallel program yields poor performance and scalability. On cluster architectures, communication vectorization and aggregation are essential to increase the granularity of communication. An advantage of CAF over other languages is that communication vectorization can be conveniently expressed in the source code using Fortran 90 triplet notations. Thus, programmers do not need to use temporary communication buffers to pack/unpack strided data. In reality, manual packing/unpacking provides the best performance on some architectures due to inefficiencies of run-time libraries [47].

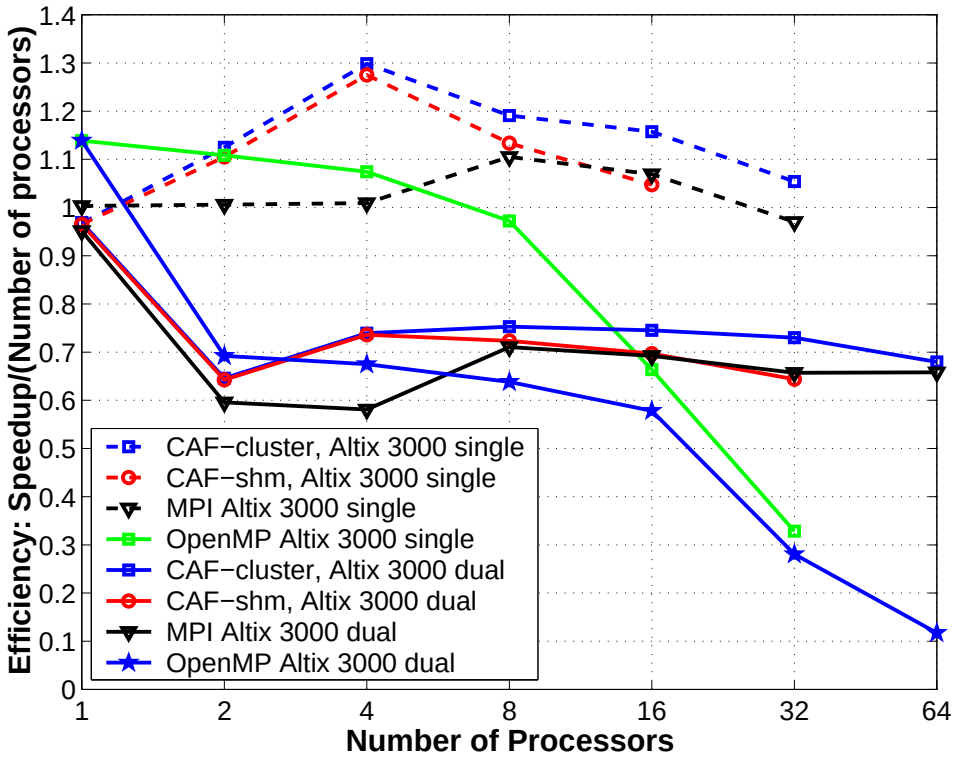


Figure 4.8 : Comparison of parallel efficiencies of the MPI, CAF with general communication, CAF with shared-memory communication, and OpenMP versions of the NAS MG benchmark on an SGI Altix 3000.

In [30], we identified that other transformations are useful to increase performance. The conversion of GETs into PUTs allows exploiting the RDMA capabilities of certain interconnect, *e.g.*, Myrinet 2000 [9], that have support for RDMA PUT, but not for GET. Another benefit of this transformation is that a value can be PUT to the destination as soon as it is produced. While the opportunities for automatic conversion are hard to identify by the compiler, it is a code style recommendation to the programmers.

In our multi-platform performance evaluation study [47], we provided detailed analysis of what transformations are necessary for CAF codes to match the performance of MPI versions for NAS MG, SP, BT, CG, and LU. The experiments were performed on three cluster architectures (see Section 3.3): the Alpha+Quadrics cluster, the Itanium2+Myrinet cluster, and the Itanium2+Quadrics cluster. Here we summarize the results for NAS MG

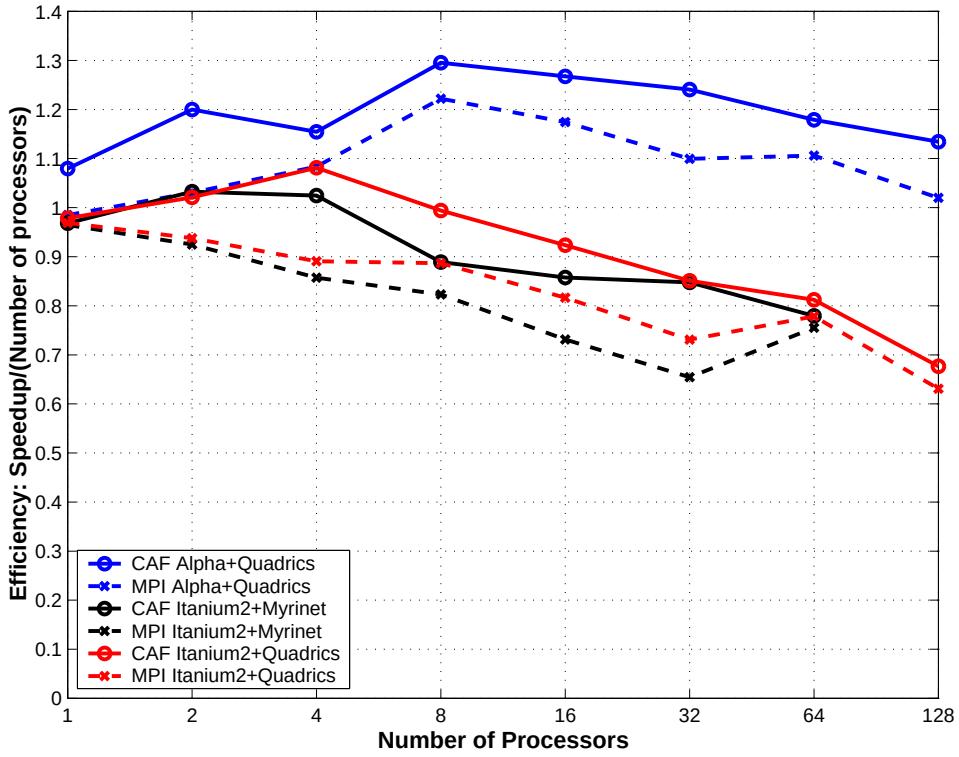


Figure 4.9 : Comparison of MPI and CAF parallel efficiency for NAS MG on Alpha+Quadrics, Itanium2+Myrinet and Itanium2+Quadrics clusters.

and BT. The complete results of our study can be found elsewhere [47].

The important optimizations for MG are: communication vectorization, point-to-point synchronization, and using PUTs for communication. Figure 4.9 illustrates that our CAF version of NAS MG class C (512^3, 20 iterations) achieves performance superior to that of the MPI version on all three platforms. On the Alpha+Quadrics cluster, our CAF version outperforms MPI by up to 16% (11% on 128 processors); on the Itanium2+Myrinet cluster, the CAF version of MG exceeds the MPI performance by up to 30% (3% on 64 processors); on the Itanium2+Quadrics cluster, MG CAF surpasses MPI by up to 18% (7% on 128 processors). The best-performing CAF version uses procedure splitting and non-blocking communication.

The MPI implementation of NAS BT attempts to hide communication latency by over-

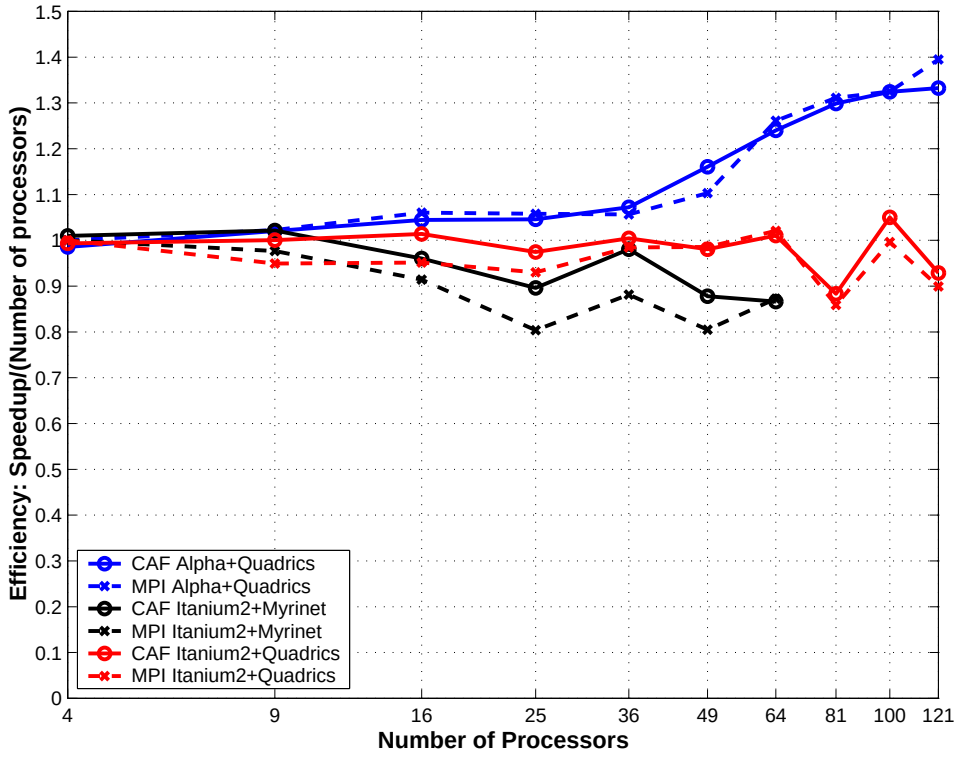


Figure 4.10 : Comparison of MPI and CAF parallel efficiency for NAS BT on Alpha+Quadrics, Itanium2+Myrinet and Itanium2+Quadrics clusters.

lapping communication with computation, using non-blocking communication primitives. The high-payoff code transformations for the CAF version are communication vectorization, packing/unpacking of strided communication, and trade-off between communication buffer space and amount of necessary synchronization. The performance achieved by the CAF version of BT class C (162^3, 200 iterations) is presented in Figure 4.10. On the Alpha+Quadrics cluster, the performance of the CAF version of BT is comparable to that of the MPI version. On the Itanium2+Myrinet cluster, CAF BT outperforms the MPI versions by as much as 8% (and is comparable for 64 processors); on the Itanium2+Quadrics cluster, our CAF version of BT exceeds the MPI performance by up to 6% (3% on 121 processors).

4.2.4 Point-to-point vs. barrier-based synchronization

The original CAF specification [87] had support only for barrier and team synchronization. In our studies [47, 30, 31, 32], we demonstrated that it is necessary to use unidirectional, point-to-point synchronization (notify and wait) to match the performance of MPI codes; for instance, NAS CG class A showed up to 59% improvement for 64-processor runs when using point-to-point synchronization instead of barriers. Point-to-point synchronization provides two benefits over barrier-based synchronization. First, fewer synchronization messages are required when each processor synchronizes with a small subset of neighbor processors (nearest-neighbor communication). Second, point-to-point synchronization allows more asynchrony in the sense that processors synchronize only with the necessary subset of neighbors and the synchronization is not collective. On the contrary, a barrier synchronizes all process images and every image is delayed waiting for the slowest one. The updated CAF language specification [86] provides unidirectional point-to-point synchronization primitives.

However, programming using point-to-point synchronization is hard. In Chapters 5, 6, and 7, we develop a technique that would enable automatic conversion of barriers into weaker point-to-point synchronization for a large class of parallel codes.

4.2.5 Improving synchronization via buffering

We found that using extra communication buffers can reduce the amount of synchronization, *e.g.*, for X-, Y- and Z-sweeps in NAS SP and BT. In our study [31, 32], we presented a hand-coded *multi-buffer* communication scheme for Sweep3D that exceeds the performance of the MPI version by up to 10% on several architectures. Chapter 8 presents multi-version variables (MVVs) that simplify development of such applications with producer-consumer communication patterns by insulating programmers from the details of buffering and pipelined synchronization. At the same time, experiments show that MVVs deliver the performance of the best hand-optimized (multi-buffer) versions.

4.2.6 Performance evaluation of CAF and UPC

We performed a thorough comparison of CAF and UPC programming models [33] using NAS benchmarks. The study revealed that it is much more difficult to match the performance of MPI+Fortran codes in UPC in comparison to CAF. The main reason is that UPC uses C as the target sequential language and does not support true multi-dimensional array abstraction. We compared the performance of MPI, CAF, and UPC versions of NAS MG, CG, SP, and BT (see Section 3.3) on four parallel architectures (see Section 3.3): the Itanium2+Myrinet cluster, the Alpha+Quadrics cluster, and the SGI Altix 3000 and the SGI Origin 2000 shared-memory machines. We used `cafc` to compile CAF codes and the Berkeley and the Intrepid UPC compilers to compile the UPC versions of the benchmarks. In the following, we summarize the results for the NAS MG and SP codes. A more detailed description of our study and results can be found elsewhere [33].

Unified Parallel C Compilers

The Berkeley UPC (BUPC) compiler [28] performs source-to-source translation. It first converts UPC programs into platform-independent ANSI-C compliant code, tailors the generated code to the the target architecture (cluster or shared-memory), and augments it with calls to the Berkeley UPC run-time system, which in turn, invokes a lower level one-sided communication library called GASNet [17]. The GASNet library is optimized for a variety of target architectures and delivers high performance communication by applying communication optimizations such as message coalescing and aggregation as well as optimizing accesses to local shared data. We used both the 2.0.1 and 2.1.0 versions of the Berkeley UPC compiler in our study.

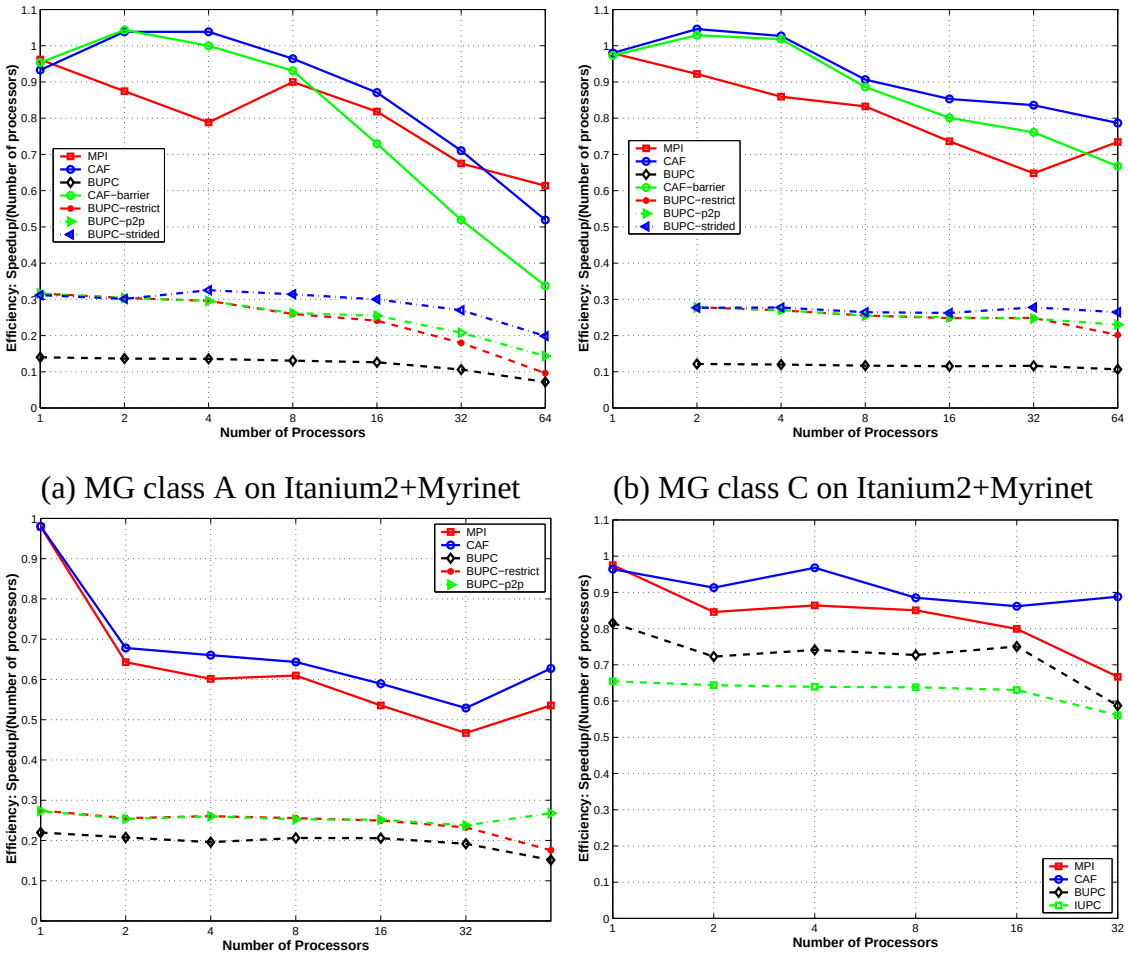
The Intrepid UPC (IUPC) compiler [71] is based on the GCC compiler infrastructure and supports compilation to shared-memory systems including the SGI Origin, Cray T3E, and Linux SMPs. The GCC-UPC compiler used in our study is version 3.3.2.9, with the 64-bit extensions enabled. This version incorporates inlining optimizations and utilizes the GASNet communication library for distributed memory systems.

NAS MG

Figures 4.11 (a) and (b) present the performance of classes A (problem size 256^3) and C (problem size 512^3) on an Itanium2 cluster with a Myrinet 2000 interconnect. The *MPI* curve is the baseline for comparison as it represents the performance of the NPB-2.3 official benchmark. The *CAF* curve represents the efficiency of the fastest code variant written in Co-array Fortran and compiled with `cafc`. To achieve high performance, the *CAF* code uses communication vectorization, synchronization strength reduction, procedure splitting and non-blocking communication. The *CAF-barrier* version is similar to *CAF*, but uses barriers for synchronization.

In Figure 4.11, the *BUPC, BUPC-restrict, BUPC-strided,* and *BUPC-p2p* curves display the efficiency of NAS MG coded in UPC and compiled with the BUPC compiler. The UPC implementation uses a program structure similar to that of the *MPI* version. All UPC versions declare local pointers for each level of the grid for more efficient access to local portions of shared arrays. The *BUPC-restrict*, *BUPC-p2p* and *BUPC-strided* differ from *BUPC* by declaring these local pointers as restricted, using the C99 `restrict` keyword, to improve alias analysis in the back-end C compiler. *BUPC* and *BUPC-restrict* use barriers for interprocessor synchronization; *BUPC-p2p* and *BUPC-strided* use point-to-point synchronization implemented at the UPC language level. *BUPC, BUPC-restrict* and *BUPC-p2p* use `upc_memput` for bulk data transfers; *BUPC-strided* uses UPC extensions to perform bulk transfers of strided data.

The results show that *CAF* has an efficiency comparable to that of *MPI*; the *CAF-barrier* performance is similar to that of *MPI* for small numbers of CPUs, but the performance degrades for larger numbers of processors. The original *BUPC* version is as much as seven times slower than *MPI* and *CAF*. We identified three major causes for this performance difference. The principal cause is lower scalar performance due to source-to-source translation issues, such as failing to convey aliasing information to the back-end compiler and inefficient code generated for linearized indexing of multi-dimensional data in UPC. Second, using barrier synchronization when point-to-point synchronization suffices degrades



(a) MG class A on Itanium2+Myrinet

(b) MG class C on Itanium2+Myrinet

(c) MG class B on Altix 3000

(d) MG class B on Origin 2000

Figure 4.11 : Comparison of MPI, CAF, and UPC parallel efficiency for NAS MG.

performance and scalability. Third, communicating non-contiguous data in UPC is currently expensive.

Source-to-source translation challenges. The following code fragment is for the residual calculation, `resid`, which is computationally intensive. *MPI*, *CAF*, and *CAF-barrier* use multi-dimensional arrays to access private data.

```
subroutine resid(u,v,r,n1,n2,n3,...)
  integer n1,n2,n3
  double precision u(n1,n2,n3),v(n1,n2,n3),r(n1,n2,n3),a(0:3)
! loop nest accounting for 33% of total walltime
  r(i1,i2,i3) = v(i1,i2,i3) - a(0) * u(i1,i2,i3) - ...
  ...
end subroutine resid
```

The corresponding routine in the UPC versions uses C pointers to access the local parts of shared arrays as shown below.

```
typedef struct sh_arr_s sh_arr_t;
struct sh_arr_s {
  shared [] double *arr;
};

void resid( shared sh_arr_t *u, shared sh_arr_t *v,
  shared sh_arr_t *r, int n1, int n2, int n3, ...) {
#define u(iz,iy,ix) u_ptr[(iz)*n2*n1 + (iy)*n1 + ix]
#define v(iz,iy,ix) v_ptr[(iz)*n2*n1 + (iy)*n1 + ix]
#define r(iz,iy,ix) r_ptr[(iz)*n2*n1 + (iy)*n1 + ix]
  double *restrict u_ptr, *restrict v_ptr, *restrict r_ptr;
  u_ptr = & u[MYTHREAD].arr[0];
  v_ptr = & v[MYTHREAD].arr[0];
  r_ptr = & r[MYTHREAD].arr[0];
  // loop nest accounting for 60% of total walltime
  r(i3, i2, i1) = v(i3, i2, i1) - a[0] * u(i3, i2, i1) - ...
  ...
}
```

If `u` were used to access shared local data via UPC's run-time address resolution for shared pointers [21, 28], the performance would suffer from executing a branch per data access. The use of `u_ptr`, a regular C pointer, enables the local portion of the shared array `u` to be accessed without the need for run-time address resolution.

In Fortran, `u` is a subroutine argument and cannot alias other variables, while in C, `u_ptr` is a pointer. Hence, lacking sophisticated alias analysis, a C compiler conservatively assumes that `u_ptr` can alias other variables. In turn, this prevents the C compiler from doing some high-level loop nest optimizations. Using Rice's HPCToolkit [100, 79] (a suite of tools for profile-based performance analysis using statistical sampling of hardware performance counters) we analyzed one-processor versions of MG class B. We discovered that the *BUPC* version of `resid` had 2.08 times more retired instructions and executed ten

times slower than its Fortran counterparts. For the entire benchmark, the performance of the *BUPC* version was seven times lower (144 vs. 21 seconds).

To inform the back-end C compiler that `u_ptr` does not alias other variables, we annotated the declaration of `u_ptr` with the C99 `restrict` keyword. Restricting all relevant pointers in `resid` resulted in a 20% reduction in the number of retired instructions and yielded a factor of two speedup for this routine. Using `restrict` for *BUPC-restrict* where it was safe to do so resulted in a 2.3 times performance improvement, reducing the execution time to 63 seconds, only three times slower than MPI instead of the original factor of seven.

With CAF, we encountered a similar difficulty with overly conservative assumptions about aliasing in back-end Fortran compilers when computing on the local parts of COMMON/SAVE co-arrays. In CAF, global co-arrays do not alias, but their pointer-based representation does not convey this information to back-end Fortran compilers. To address this problem, we developed a source-to-source transformation known as procedure splitting (see Section 4.1.5). This transformation eliminates overly conservative assumptions about aliasing by transforming a pointer-based representation for co-array data into one based on dummy arguments, which are correctly understood to be free of aliases.

While alias analysis of UPC programs can be improved by having programmers or (in some cases) UPC compilers add a `restrict` keyword, there is another fundamental issue preventing efficient optimization of scientific C codes. The Fortran code snippet above uses multi-dimensional arrays with symbolic bounds, expressed as specification expressions by parameters passed to the `resid` subroutine. In UPC MG, the macro `u` creates the syntactic illusion of a multi-dimensional array, but in fact, this macro linearizes the subscript computation. C does not have the ability to index `u` using a vector of subscripts. Thus, to safely reorder such references during optimization, C compilers must perform dependence analysis of linearized subscripts, which is harder than analyzing a vector of subscripts. This tends to degrade the precision of dependence analysis, which limits the ability of C compilers to exploit some high-level optimizations, and, thus, yields slower code. To estimate

the performance degradation due to linearized subscripts in C, we linearized subscripts in a Fortran version of `resid`. This change doubled the execution time of the Fortran version of `resid` and degraded the overall performance of MG class B by 30% on the Itanium2+Myrinet cluster.

Point-to-point synchronization. In MG, each SPMD thread needs to synchronize only with a small number of neighbors. While a collective barrier can be used to provide sufficient synchronization, it provides more synchronization than necessary. Our experiments show that unnecessary collective synchronization degrades performance on loosely-coupled architectures. This effect can be seen in Figures 4.11 (a) and (b) by comparing the efficiency of the *BUPC-restrict* and *BUPC-p2p* versions. We derived *BUPC-p2p* from *BUPC-restrict* by using a reference language-level implementation of point-to-point synchronization. The performance boost is evident for the larger number of processors and amounts to 49% for class A and 14% for class C in 64 processor executions. The class A executions benefit more from using point-to-point synchronization because they are more communication bound. A similar effect can be seen for CAF: for 64 processors, *CAF-barrier* shows a 54% slowdown for class A and a 18% slowdown for class C.[2]

Non-contiguous data transfers. For certain programs, efficient communication of non-contiguous data can be essential for high efficiency. For MG, the y-direction transfers of *BUPC-restrict* are performed using several communication events, each transferring a contiguous chunk of memory equal to one row of a 3D volume. The *BUPC-strided* version is derived from *BUPC-p2p*. It moves data in the y-direction by invoking a library primitive to perform a strided data transfer; this primitive is a member of a set of proposed UPC language extensions for strided data transfers [18]. Even using Berkeley's reference implementation of the strided communication operation (as opposed to a carefully-optimized implementation) yielded a 28% performance improvement of *BUPC-strided* over *BUPC-p2p* for class A on 64 processors and a 13% efficiency improvements for class C on 64

[2]In later studies (see Chapter 7), we eliminated redundant barriers and observed somewhat smaller improvements.

processors. The most efficient communication can be achieved by packing data into a contiguous communication buffer and sending it as one contiguous chunk. A version that uses packing is marginally more efficient than *BUPC-strided*, thus, we do not show it on the plot.

While in most cases using the UPC strided communication extensions is more convenient than packing and unpacking data on the source and destination, we found it more difficult to use such library primitives than simply reading or writing multi-dimensional co-array sections in CAF using Fortran 90 triplet notation, which `cafc` automatically transforms into equivalent strided communication. For CAF programs, a compiler can automatically infer the parameters of a strided transfer, such as memory strides, chunk sizes, and stride counts; whereas in UPC, these parameters must be explicitly managed by the user.

Figure 4.11 (c) presents the performance results of NAS MG class B (problem size 256^3) for the Altix 3000 architecture. The *MPI*, *CAF*, *BUPC*, *BUPC-restrict*, and *BUPC-p2p* curves are similar to the ones presented for the Itanium2+Myrinet 2000 cluster. We used the same versions of the Intel Fortran and C compilers. Therefore, we expected similar trends for the scalar performance of *MPI*, *CAF* and *BUPC*. Indeed, *MPI* and *CAF* versions show comparable performance, while *BUPC* is up to 4.5 times slower and *BUPC-restrict* is 3.6 times slower than *CAF*. The efficiency of all programs is lower on this architecture compared to that on the Itanium2+Myrinet2000 cluster, because in our experiments on the Altix architecture we ran two processes per dual node, sharing the same memory bus.

For CAF, using barrier-based instead of point-to-point synchronization does not cause a significant loss of performance on this architecture for 32 or fewer processors. However, for 64 processors, we observed a performance degradation of 29% when CAF MG used barriers for synchronization. For UPC, *BUPC-p2p* outperforms *BUPC-restrict* by 52% for NAS MG on 64 processors.

Figure 4.11 (d) presents the performance results on the Origin 2000 machine for NAS MG class B (problem size 256^3). The *MPI* curve corresponds to the original MPI version implemented in Fortran. The *CAF* curve gives the performance of the optimized CAF

version with the same optimizations as described previously, except that non-blocking communication is not used, because the architecture supports only synchronous interprocessor memory transfers. The *BUPC* and *IUPC* curves describe the performance of the UPC version of MG compiled with the Berkeley UPC and the Intrepid UPC compilers, respectively.

The *CAF* version slightly outperforms the *MPI* version due to more efficient one-sided communication [48]. The *MPI* version slightly outperforms *BUPC* which, in turn, slightly outperforms *IUPC*. The MIPSPro C compiler, which is used as a back-end compiler for *BUPC*, performs more aggressive optimizations compared to the Intel C compiler. In fact, using the `restrict` keyword does not yield additional improvement, because the alias analysis done by the MIPSPro C compiler is more precise. Nonetheless, it is our belief that the lack of multi-dimensional arrays in the C language prevents the MIPSPro C compiler from applying high-level loop transformations such as unroll & jam and software pipelining, resulting in an 18% slowdown of *BUPC* MG class B on one processor relative to the one-processor *MPI* version. The *IUPC* version was compiled with the Intrepid compiler based on GCC [71], which performs less aggressive optimization than the MIPSPro compiler. Lower scalar performance of the *IUPC* version results in a similar 48% slowdown.

The one-processor *BUPC* versions of MG class A execute approximately 17% slower than the corresponding *CAF* version (65 seconds vs. 55 seconds). To determine the cause of this performance difference, we used SGI's `perfex` hardware counter-based analysis tool to obtain a global picture of the application's behavior with regards to the machine resources. A more detailed analysis using SGI's `ssrun` and Rice's HPCToolkit led us to conclude that the *BUPC* version completes 51% more loads than the *CAF* version. The cause of this was the failure of the MIPSPro C compiler to apply loop fusion and alignment to the most computationally intensive loop nest in the application (in the `resid()` routine). The MIPSPro Fortran compiler performed loop fusion and alignment. This reduced the memory traffic by reusing results produced in registers, which in turn improved the software-pipelined schedule for the loop. We expect similar issues to inhibit the performance of other less computationally intensive loops in the BUPC-compiled application.

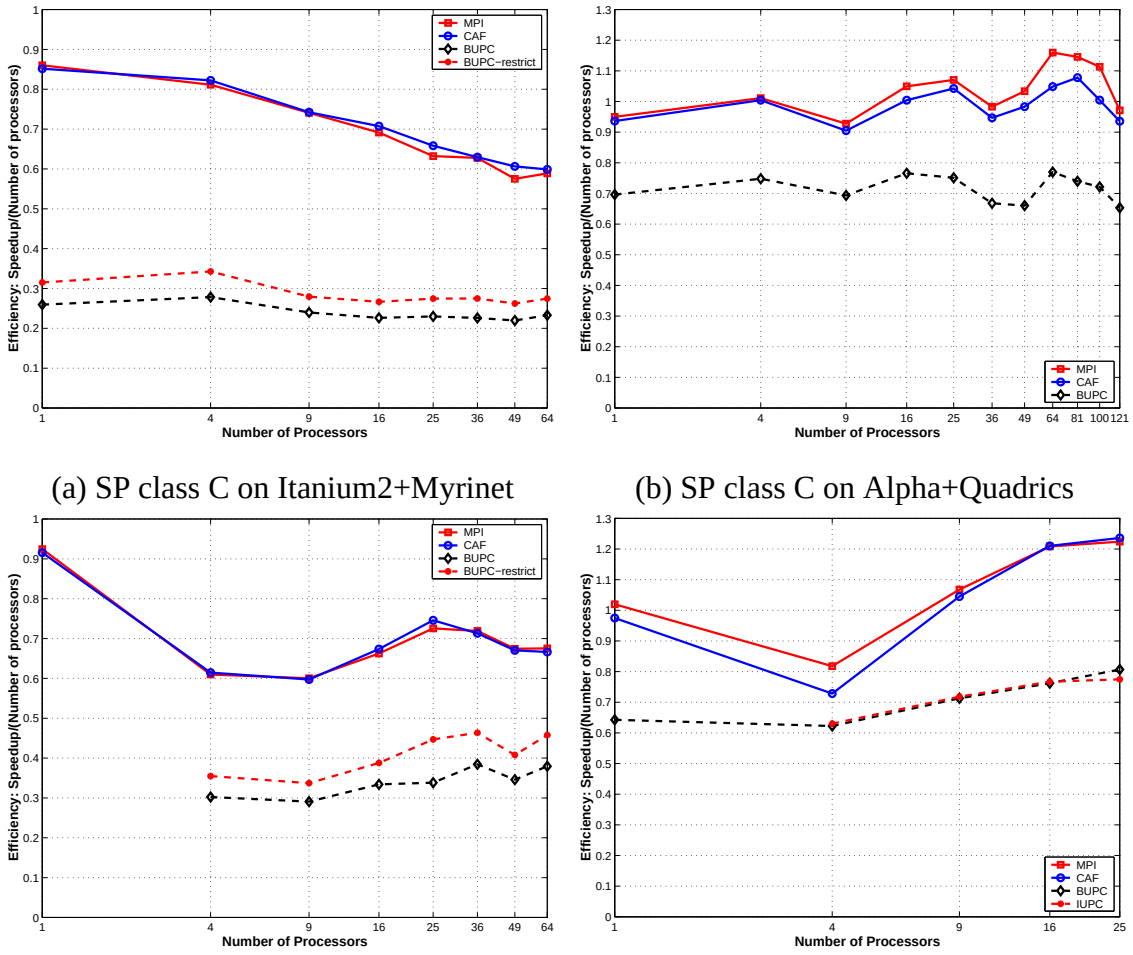
NAS SP

Figure 4.12 (a) shows the parallel efficiency curves for NAS SP class C (problem size 162^3) on the Itanium2+Myrinet2000 cluster. The *MPI* curve serves as the baseline for comparison and represents the performance of the original NPB-2.3 SP benchmark. The *CAF* curve shows the performance of the fastest CAF variant. It uses point-to-point synchronization and employs non-blocking communication to better overlap communication with computation. The *BUPC* and *BUPC-restrict* curves show the performance of two versions of SP compiled with the Berkeley UPC compiler.

The performance of the *CAF* version is roughly equal to that of *MPI*. Similar to the other UPC NAS benchmarks compiled using the back-end Intel C compiler, the scalar performance suffers from poor alias analysis: the one-processor version of *BUPC* class C is 3.3 times slower than the one-processor *MPI* version. Using the `restrict` keyword to improve alias analysis yielded a performance boost: the one-processor version of *BUPC-restrict* class C is 18% faster than *BUPC*. The trend is similar for larger numbers of CPUs.

There is a conceptual difference in the communication implementation of the dimensional sweeps in *CAF* and *BUPC*. The CAF implementation uses point-to-point synchronization, while the UPC implementation uses split-phase barrier synchronization. In general, it is simpler to use split-phase barrier synchronization; however, for NAS SP, point-to-point and split-phase barrier synchronization are equally complex. Since barrier synchronization is stronger than necessary in this context, it could potentially degrade performance.

Figure 4.12 (b) reports the parallel efficiency of the *MPI*, *CAF*, and *BUPC* versions of NAS SP class C (problem size 162^3) on the Alpha+Quadrics platform. It can be observed that the performance of *CAF* is slightly worse than that of *MPI*. We attribute this to the lack of non-blocking notification support in the CAF run-time layer. The performance of the *BUPC* version is lower than that of *MPI* due to worse scalar performance of the C code: it is 1.4 times slower for one processor and increases to 1.5 times slower for 121 processors. The use of the `restrict` keyword does not have any effect on performance because of the high quality dependence analysis of the Alpha C compiler.



(a) SP class C on Itanium2+Myrinet

(b) SP class C on Alpha+Quadrics

(c) SP class C on Altix 3000

(d) SP class B on Origin 2000

Figure 4.12 : Comparison of MPI, CAF, and UPC parallel efficiency for NAS SP.

Figure 4.12 (c) shows the efficiency of *MPI*, *CAF*, *BUPC*, and *BUPC-restrict* versions of NAS SP class C (problem size 162^3) on the Altix 3000 system. The performance of *CAF* and *MPI* is virtually identical, while *BUPC* is a factor of two slower on four processors (we were not able to run the one-processor version due to memory constraints). Using the `restrict` keyword improves performance by 17% on average.

Figure 4.12 (d) shows the parallel efficiency of *MPI*, *CAF*, *BUPC*, and *IUPC* versions of NAS SP for class B (problem size 102^3) on the Origin 2000 machine. The performance

of *CAF* is very close to that of *MPI*. Both UPC versions have similar performance and are slower than *MPI* or *CAF*. Again, the difference is attributable to lower scalar performance. For the one-processor SP class B, *BUPC* was 57% slower than *MPI*.

We observed that the one-processor BUPC-compiled version of SP class A executes approximately 43% slower than the corresponding *MPI* version. Using hardware performance counters, we found that the BUPC-compiled version executed twice as many instructions as the CAF version.

A detailed analysis of this difference using HPCToolkit and SGI's `ssrun` helped us identify that a computation-intensive, single-statement loop nest present in both versions was getting compiled ineffectively for UPC. In Fortran, the loop, which accesses multi-dimensional array parameters, was unrolled & jammed to improve outer loop reuse and software pipelined by the MIPSPro Fortran compiler. The corresponding UPC loop, which accesses 1D linearized C arrays through pointers, was not unrolled or software pipelined by the MIPSPro C compiler, leading to less efficient code. This more than doubled the number of graduated instructions for the loop in the *BUPC* version compared to the *MPI* version.

The generated code for the Fortran loop was able to reuse not only floating point data, but integer address arithmetic as well. We believe that this specific observation applies to many of the single-statement computationally intensive loops present in SP's routines. For multi-statement loops, the difference in the number of executed instructions between the UPC and MPI versions was not as significant. Such loops already have opportunities for reusing address arithmetic and floating point values even without applying transformations such as unroll & jam.

Chapter 5

Co-spaces: Communication Topologies for CAF

The co-array co-shape [86] defines how a co-array is mapped onto CAF program images. It is defined by a declaration or allocation of a co-array variable, *e.g.*, `integer a[4,2,*]`. The target image of each PUT/GET is specified for a co-array access by using one or more co-subscripts [86] with bracket notation, *e.g.*, `a[1,2,3]`. The co-shape maps a co-dimension subscript vector into a linear image index in the range from 1 to `num_images()`. This index is used to identify the target image for a one-sided communication operation. The co-shape "arranges" images into a virtual *communication topology*, which is analogous to a Cartesian space without periodic boundaries.

While convenient for a limited number of applications that use processor decompositions of a Cartesian shape without periodic boundaries, a multi-dimensional co-shape is not well suited to express other communication topologies. As a result, programmers often use *neighbor arrays* to store the target image indices of each image's communication partners. Neighbor arrays are used to implement commonly-used communication topologies. A neighbor array on image `p` stores the image indices of the logical communication partners with whom `p` communicates via PUT/GET. For example, in an application that uses a 3D data decomposition onto processors, a 3-element vector `successor` of successors and a 3-element vector `predecessor` of predecessors can be used to store the image indices of logical successor and predecessor images along each of the coordinate axes. The programmer can declare a co-array used for data exchange between images as `integer a[*]` and use neighbor arrays to specify the target of one-sided communication, *e.g.*, `a[successor(dim)]`, where `dim` $\in [1,3]$. This idiom appears in many scientific CAF codes such as NAS benchmarks and Sweep3D.

Neighbor arrays can express an arbitrary communication topology; however, they have two major disadvantages. First, they are not standardized and they are typically used to implement commonly used topologies in an ad-hoc way. This imposes the burden of topology implementation on programmers. Second, neighbor arrays complicate compiler analysis of communication patterns, because they are initialized and used explicitly, and it is hard to infer the properties of the communication topology from the initialization code and communication structure from their usage.

We explore using communication topologies to replace multi-dimensional co-shapes in CAF. We focus on three commonly used communication topologies: group, Cartesian, and graph, inspired by MPI communicators and process topologies [112, 62]. We call CAF communication topologies *co-spaces* by analogy with CAF's co-dimension, co-rank, co-shape, etc., co-array terminology [86]. Co-spaces are CAF programming model objects, in the sense that they can be a part of the language or run-time library and compiler (see Section 5.5), that take the place of ad-hoc neighbor arrays. They provide a mapping from a virtual communication topology to the linear image number in the range $[1, N]$, where N is the total number of images. This number can be used to specify the target image of not only communication, but also synchronization or a co-function invocation (see Chapter 9). The goal of co-spaces is to improve programmability by providing commonly used reusable abstractions to programmers. Besides, co-spaces expose the information about the communication topology and communication partners to a CAF compiler which enables global program analysis (see Chapter 6) and powerful communication and synchronization optimization such as synchronization strength reduction (see Chapter 7).

In the rest of this chapter, we discuss CAF co-shapes in more detail and formalize the concept of group, Cartesian, and graph co-spaces. Chapters 6 and 7 describe how co-spaces help the global analysis and optimization of CAF programs.

5.1 Communication topologies in CAF

Each co-array declaration/allocation in CAF defines a co-array co-shape [86] that can be thought of as a Cartesian communication topology or processor space. For instance, `integer a(10,10)[*]` defines co-array `a` that has a 1D Cartesian communication topology without periodic boundaries where each image is identified by a unique number from 1 to `num_images()`. The declaration `integer b(100)[N,*]` specifies a co-array with a 2D Cartesian processor decomposition without periodic boundaries where images are arranged into a virtual 2D array of processors in column-major order. A part of `b` is associated with each process image that is identified by a pair of coordinates (i, j) in a 2D Cartesian processor space with dimensions N and M, where M is the minimum number such that $N \times M \geq$ `num_images()`. The legal coordinate values are $1 \leq i \leq N$ and $1 \leq j \leq M$ with an additional constraint on j: the process image index p, computed as $p = (i - 1) + N \times (j - 1) + 1$, must not exceed the number of images. For example, if the total number of images is three and a co-array is declared as `a[2,*]`, the legal co-dimension indices are: $(1, 1)$ corresponding to process image index 1, $(2, 1)$ to 2, $(1, 2)$ to 3. However, referencing image $(2, 2)$, which corresponds to process image index 4, causes a critical run-time error. The topology is incompletely filled in the sense that coordinate $(2, 2)$ is legal in the virtual topology, but there is no process image associated with it. This inconsistency would require programmers to implement extra logic to address corner cases for communication. To date, we have not seen an application that uses such an incompletely filled co-shape. Additional details of CAF co-space semantics can be found in [86].

We summarize the weaknesses of the CAF's multi-dimensional co-shape before formalizing the concept of co-spaces that, we think, is a better abstraction to express the target of one-sided communication in many CAF applications.

- As mentioned above, a co-array co-shape provides only one communication topology type – a Cartesian processor grid without periodic boundaries. To express any other type of topology, programmers declare co-arrays with only one co-dimension and typically use neighbor arrays that provide the mapping from the application's virtual

topology into the image index. This approach is ad-hoc and error-prone.

- A co-shape may be incompletely filled if the product of all co-dimensions is not equal to the number of process images, which requires extra programming logic to handle special cases.

- A co-shape is not inherited across a procedure call. It has to be redeclared afresh in each subroutine for each co-array dummy argument. Usually, the co-shape information is passed as extra arguments to the subroutine.

- The target images of point-to-point and team synchronization [86] are specified differently than those of communication. Synchronization primitives accept only image indices; the co-shape does not exist for them. Programmers can use neighbor arrays or the `image_index` intrinsic function [86] to obtain the target image of a synchronization event, where `image_index(a,sub)` returns the process image index corresponding to to the set of co-subscripts `sub` for co-array `a`. Both of these approaches are cumbersome.

- A co-shape can be used to specify the target of a co-subroutine/co-function invocation discussed in detail in Chapter 9.

- A co-shape is defined across all process images. It is not useful for expressing communication topologies for subsets of process images.

- Co-shape Cartesian coordinates are global. Many codes are better expressed using coordinates relative to each image. For example, consider an application based on nearest-neighbor communication. It is easier and more intuitive to think about the communication partners in relative terms, *e.g.*, my left neighbor, my right neighbor, etc. The `image_index` intrinsic can be used for Cartesian topologies without periodic boundaries; otherwise, a natural choice is to use neighbor arrays.

- Because co-shapes are not expressive enough, programmers have to use neighbor

arrays to specify communication target images. The communication topology information specified using a neighbor array is known to the programmer, but nearly impossible to infer automatically by a CAF compiler. However, understanding communication topology is essential for analysis and optimization of communication & synchronization.

General inconvenience and complication of compiler analysis when communication is expressed via co-array co-shapes and neighbor arrays forced us to rethink the organization of process topologies in CAF. We believe that more general and reusable group, Cartesian, and graph co-spaces will simplify programming of many CAF codes and enable program analysis by exposing the properties of communication topologies and PUT/GET targets to CAF compilers.

5.2 Co-space types

The MPI standard [112, 62] has extensive support for communicators and process topologies using a design based on years of experience. We leverage these ideas to design group, Cartesian, and graph *co-spaces* – virtual communication topologies in CAF. A co-space is a CAF object that represents a virtual communication topology. In essence, co-spaces encapsulate the same information as the neighbor arrays, but they do this in a systematic way. Each co-space object represents an ordered group of images that might have a logical structure overlaid on the members of the group. For a Cartesian co-space, this structure can be a Cartesian topology with or without periodic boundaries. For a graph co-space, the relation among images is determined by a directed graph. In addition, a group co-space enables the programmer to permute the image indices of an existing co-space group or to specify a subset of a process group. Each co-space type has a well-defined interface, presented below. The interface includes functionality for creating and destroying co-spaces and a number of topology query functions such as membership and neighbor relationship.

Each co-space has a group of member images ordered by their unique rank. The size of a co-space is the size of its group. Each member has a unique rank from the interval

$[0, N-1]$, where N is the group size. Each member is a real image with the index from the $[1, \texttt{num\_images}()]$ range. The image index is used to specify the target image of PUT/GET or synchronization. A co-space provides a mapping from the group rank to the image index. For example, group ranks 0, 1, and 2 may correspond to image indices 1, 4, and 7. In addition, the group co-space allows permutation of the members of the group. For example, group ranks 0, 1, and 2 may correspond to image indices 7, 1, and 4. A new co-space can be created by a collective operation among the members of an existing co-space.

There is a predefined `CAF_WORLD` co-space that includes all process images from 1 to `num_images()` ordered according to their ranks from 0 to `num_images()`-1. It corresponds to a co-array declaration with one co-dimension and the lower bound of 0, *e.g.*, `a(10)[0:*]`. `CAF_WORLD` has the topology of a 1D Cartesian co-space without periodic boundaries.

We use the following notation for the co-space specification:

- Each function is prefixed with *CS* to provide the co-space interface namespace.
- A co-space parameter of type `type(XXX)` can be of any co-space type: `type(Group)` for group co-space, `type(Cartesian)` for Cartesian, or `type(Graph)` for graph; or the parameter can also be `CAF_WORLD`.
- An argument in brackets, *e.g.*, `[x]`, is optional. Notation `[x=a]` means that optional argument `x` has default value `a`. In particular, `image=this_image()` means that the optional argument `image` is not present, and its value is the index of the invoking process image.
- `NOERROR` is an integer constant that indicates that a function call was successful.
- `NORANK` is an integer constant that indicates that there is no rank associated with an image.
- `NOIMAGE` is an integer constant that indicates that the image does not exists.

Most functionality of the group co-space management is the same for Cartesian and graph co-spaces, and we describe it only for the group co-space.

5.2.1 Group

This section formalizes the interface functions for group co-spaces.

- **`CS_Create`**

```
integer function CS_Create(cs, ecs, member, [rank])
    type(Group), intent(out) ::  cs
    type(XXX), intent(in) ::  ecs
    logical, intent(in) ::  member
    integer, intent(in), optional ::  rank
```

  Creates a new group co-space object `cs`; returns `NOERROR` if successful. Only images of an existing co-space `ecs` participate in the collective call. If `member` is equal to `.true.`, the invoking image becomes a member of the new co-space with the rank equal to `rank`. `rank` determines the rank of the image in the group and can define a permutation of group members. If rank is not specified, the ordering of images in `cs` is consistent with their ordering in the `ecs` group. `rank` must be unique for each member of `cs`. `rank` determines the ordering of the image in the group and must be in the range from 0 to `CS_Size(cs)`-1, where `CS_Size(cs)` returns the size of the group.

- **`CS_Destroy`**

```
subroutine CS_Destroy(cs)
    type(XXX), intent(inout) ::  cs
```

  Destroys the co-space object `cs` of any type. If `cs` is `CAF_WORLD`, it is a critical run-time error.

- **`CS_Image`**

```
integer function CS_Image([cs=CAF_WORLD], [rank])
    type(XXX), intent(in), optional ::  cs
    integer, intent(in), optional ::  rank
```

Returns the image index for the group member with rank `rank`. `rank` must be in [0,`CS_Size(cs)`−1] range. If `rank` is not specified, it is assumed to be the rank of the invoking image returned by the `CS_Rank(cs)` function, so a call without the `rank` argument is equivalent to CAF's `this_image()` intrinsic.

- **`CS_Rank`**

```
integer function CS_Rank([cs=CAF_WORLD], [image=this_image()])
    type(XXX), intent(in), optional ::  cs
    integer, intent(in), optional ::  image
```

Returns the co-space group rank of process image with index `image`. Returns `NORANK` if the `image` is not a member of `cs`. If `image` is not specified, the invoking image is assumed.

- **`CS_Size`**

```
integer function CS_Size([cs=CAF_WORLD])
    type(XXX), intent(in), optional ::  cs
```

Returns the size of the co-space `cs`.

- **`CS_IsMember`**

```
logical function CS_IsMember(cs, [image=this_image()])
    type(XXX), intent(in) ::  cs
    integer, intent(in), optional ::  image
```

Returns `.true.` if the process image with index `image` is a member of the co-space `cs`. If co-space object `cs` is uninitialized, `CS_IsMember` returns `.false.`.

- **`CS_Group`**

```
subroutine CS_Group(cs, images)
    type(XXX), intent(in) ::  cs
    integer, intent(out) ::  images(*)
```

Fills the `images` array argument with the indices of co-space `cs` images in the order of their member ranks. The array should be sufficiently long, at least `CS_Size(cs)`, to fit all image indices.

5.2.2 Cartesian

This section formalizes the interface functions for Cartesian co-spaces.

- **`CS_Create`**

```
integer function CS_Create(cs, ecs, numDims, dims, [periods],
 [order])
    type(Cartesian), intent(out) ::  cs
    type(XXX), intent(in) ::  ecs
    integer, intent(in) ::  numDims
    integer, intent(in) ::  dims(numDims)
    integer, intent(in), optional ::  periods(numDims)
    integer, intent(in), optional ::  order
```

  Creates a new Cartesian co-space object `cs`; returns `NOERROR` if successful. Only images of an existing co-space `ecs` participate in the call. Each member of `ecs` becomes a member of `cs` with the same group rank. `numDims`, `dims`, `periods`, and `order` must be the same on every invoking image. The `numDims` is the number of dimensions of the Cartesian topology; it must be positive. The size of each dimension is determined by the `dims(i)`, $1 \leq i \leq$`numDims`; it must be positive. `dims(1:numDims)` defines the shape of the Cartesian topology in the column-major order. The product of all dimensions must be equal to the number of `cs` group members. If `periods(i)` is equal to `.false.`, dimension `i` is not periodic; otherwise, it is periodic. If `periods` is not specified, all dimensions are assumed to be periodic. The optional parameter `order` determines the arrangement of dimensions. It can be either `CS_ColumnMajor` or `CS_RowMajor`. As the names imply, the first corresponds to the column-major order, the second – to the row-major order. The default order is row-major.

- **`CS_Neighbor`** for one axis

```
integer function CS_Neighbor(cs, axis, offset, [image=this_image()])
    type(Cartesian), intent(in) ::  cs
    integer, intent(in) ::  axis
    integer, intent(in) ::  offset
    integer, intent(in), optional ::  image
```

  Returns the image index of the `offset`-th Cartesian neighbor along the dimension `axis`. `axis` must have the value in $[1, N]$, where N is the number of dimensions.

The `offset` can be positive, negative, or zero. The offset of zero corresponds to the coordinates of the invoking image and the function returns the index of the invoking image. If dimension `axis` is not periodic and the neighbor does not exist, the call returns `NOIMAGE`.

- **`CS_Neighbor`** for several axes

```
integer function CS_Neighbor(cs, offsets, [image=this_image()])
    type(Cartesian), intent(in) ::  cs
    integer, intent(in) ::  offsets(*)
    integer, intent(in), optional ::  image
```

Returns the image index of the Cartesian topology member specified by the offset vector `offsets`. `offsets(i)` is the offset along dimension `i`, $1 \leq$`i`$\leq N$, where N is the number of the topology dimensions. If the member does not exist, the call returns `NOIMAGE`.

- **`CS_GetNumDimensions`**

```
integer function CS_GetNumDimensions(cs)
    type(Cartesian), intent(in) ::  cs
```

Returns the number of dimensions of the Cartesian topology.

- **`CS_GetDimensions`**

```
subroutine CS_GetDimensions(cs, dims)
    type(Cartesian), intent(in) ::  cs
    integer, intent(out) ::  dims(*)
```

Fills the integer array `dims` with the dimension sizes of the Cartesian co-space `cs`.

- **`CS_GetPeriods`**

```
subroutine CS_GetPeriods(cs, periods)
    type(Cartesian), intent(in) ::  cs
    logical, intent(out) ::  periods(*)
```

Fills the logical array `periods` with the periods of the Cartesian co-space `cs`.

- **`CS_CartCoords`**

```
subroutine CS_CartCoords(cs, coords, [image=this_image()])
    type(Cartesian), intent(in) ::  cs
    integer, intent(out) ::  coords(*)
    integer, intent(in), optional ::  image
```

Returns the integer array `coords` of the image `image` coordinates in the Cartesian co-space `cs`; the lower bound of each dimension is 0.

- **`CS_HasNeighbor`** for one axis

```
logical function CS_HasNeighbor(cs, axis, offset,
 [image=this_image()])
    type(Cartesian), intent(in) ::  cs
    integer, intent(in) ::  axis
    integer, intent(in) ::  offset
    integer, intent(in), optional ::  image
```

  Returns `.false.` if `CS_HasNeighbor(cs,axis,offset,image)` returns `NOIMAGE`; otherwise, returns `.true.`. In other words, returns `.true.` iff the neighbor with offset `offset` along dimension `axis` exists.

- **`CS_HasNeighbor`** for several axes

```
logical function CS_HasNeighbor(cs, offsets, [image=this_image()])
    type(Cartesian), intent(in) ::  cs
    integer, intent(in) ::  offsets(*)
    integer, intent(in), optional ::  image
```

  Returns `.false.` if `CS_HasNeighbor(cs,offsets,image)` returns `NOIMAGE`; otherwise, returns `.true.`.

5.2.3 Graph

A graph communication topology is an arbitrary distributed directed multi-graph $G = (V, E)$. The vertices of the graph are the process images of `cs`. Each edge $e = (p_1, p_2, \langle c, i_c \rangle)$ from image p_1, $p_1 \in V$, to image p_2, $p_2 \in V$, is classified by a pair of "coordinates": edge class c and index i_c within the class. An edge e represents a potential one-sided communication or synchronization operation executed by image p_1 and directed towards image p_2. Note that an undirected edge can be expressed with two corresponding directed edges, potentially doubling the memory requirement. However, most real codes benefit from directional edges, and we do not provide an interface for the support of undirected edges. The intent of the edge class c is to group edges of the same type and index

them within c via edge index i_c. This is useful for expressing communication for a generalized block data distribution (see Section 5.3 for a detailed example).

- **CS_Create**

```
integer function CS_Create(cs, ecs, numNbrs, nbrs,
 [numClasses=1])
    type(Graph), intent(out) ::  cs
    type(XXX), intent(in) ::  ecs
    integer, intent(in) ::  numNbrs(numClasses)
    integer, intent(in) ::  nbrs(*)
    integer, intent(in), optional ::  numClasses
```

Creates a new graph co-space object `cs`; returns `NOERROR` if successful. Only images of an existing co-space `ecs` participate in the call. Each member of `ecs` becomes a member of `cs` with the same rank. Each invoking image p specifies a local part of the graph – all graph edges emanating from p. The collective co-space creation operation may reconstruct the entire distributed graph.

`numClasses` specifies the number of edge classes. Each edge class is a unique number from [1,`numClasses`]. For each edge class c, there are `numNbrs(`c`)` outgoing edges T_c with edge indices i_c, $i_c \in [1, \texttt{numNbrs}(c)]$. The source of each edge e, $e \in T_c$, is the invoking image I. The `nbrs` image array specifies the sink of every edge e, $e \in T_c$. Let S_c be the start position of class c edge sinks in `nbrs`, then $S_1 = 1$, $S_c = S_{c-1} + \texttt{numNbrs}(j)$, $c > 1$. Therefore, set $T_c = \{e | e = (I, \texttt{nbrs}(S_c + i_c - 1), \langle c, i_c \rangle), i_c \in [1, \texttt{numNbrs}(c)]\}$. Set T of all outgoing edges is $T = \bigcup_{c \in [1, \texttt{numClasses}]} T_c$.

- **CS_Neighbors**

```
subroutine CS_Neighbors(cs, nbrs, [classid=1], [dir=successor],
[nbrIndex], [image=this_image()])
    type(Graph), intent(in) ::  cs
    integer, intent(out) ::  nbrs(*)
    integer, intent(in), optional ::  classid
    integer, intent(in), optional ::  dir
    integer, intent(in), optional ::  nbrIndex
    integer, intent(in), optional ::  image
```

The subroutine identifies a set of edges W according to the argument values. It returns the array `nbrs` with process image indices for the set of process images

D that are the end points of W edges. The parameter `dir` determines whether D consists of sources or sinks of W edges. If `dir` is equal to `successor`, D consists of edge sinks: $D = \{\forall w \in W | sink(w)\}$. If `dir` is equal to `predecessor`, D consists of edge sources: $D = \{\forall w \in W | source(w)\}$. The default value of `dir` is `successor`.

If `dir` is equal to `successor`, W consists of edges emanating from process image I with index `image` that have edge class `classid` and edge index `nbrIndex`. If `nbrIndex` is not present, W consists of all such edges with class `classid`. In this case, D consists of the same neighbor image indices in the same order that were specified to the co-space `cs` creation call on image `image` for edge class `classid`. If `nbrIndex` is present, W contains a single edge (or it is an empty set) corresponding to a potential one-sided operation; `CS_Neighbor` function can be used to more conveniently retrieve this single image index.

If `dir` is equal to `predecessor`, W consists of edges with sink I, edge class `classid`, and class index `nbrIndex`. Note that there can be several incoming edges for a pair of `classid` and `nbrIndex` and the order of these edges is not defined. It `nbrIndex` is not present, W consists of all edges with sink I and edge class `classid`.

- **`CS_GetNumNeighbors`**

```
integer function CS_GetNumNeighbors(cs, [classid=1],
 [dir=successor], [nbrIndex], [image=this_image()])
    type(Graph), intent(in) ::  cs
    integer, intent(in), optional ::  classid
    integer, intent(in), optional ::  dir
    integer, intent(in), optional ::  nbrIndex
    integer, intent(in), optional ::  image
```

Returns the number of neighbor images in the `nbrs` array returned by the `CS_Neighbors(cs,nbrs,classid,dir,nbrIndex,image)` call.

- **`CS_Neighbor`**

```
integer function CS_Neighbor(cs, nbrIndex, [classid=1],
[image=this_image()])
    type(Graph), intent(in) ::  cs
    integer, intent(in) ::  nbrIndex
    integer, intent(in), optional ::  classid
    integer, intent(in), optional ::  image
```

  Returns the neighbor image returned in the `nbrs` array by the `CS_Neighbors(cs,nbrs,classid,successor,nbrIndex, image)` call. If the `nbrs` array is empty (neighbor does not exist), the call returns `NOIMAGE`. This function can be used to specify the target of one-sided operation more conveniently than using `CS_Neighbors` subroutine.

5.3 Co-space usage examples

Group co-space

Group co-space can be used to create a group of images. This is particularly useful for coupled applications such as the Community Climate System Model (CCSM) [50] that operate in independent but interacting groups of images. A Cartesian or graph co-space can further be used to impose additional communication topology structure on these groups.

Group co-space can be used to create an alternative numbering of images if the one provided by the run-time library at startup is not satisfactory. A typical example is a mapping of processes on a cluster with dual-processor nodes. During our experiments, we encountered a problem that the cluster job scheduler assigned image numbers by binding processes to processors in the following order: $n_1 : 1, n_2 : 1, \ldots, n_k : 1, n_1 : 2, \ldots, n_k : 2$, where $n_i : j$ denotes j-th processor of the i-th node. However, the ordering that yields better performance is $n_1 : 1, n_1 : 2, n_2 : 1, n_2 : 2, \ldots, n_k : 1, n_k : 2$. Group co-space `myworld` can be created as shown in the following pseudocode and can be used later to specify the target images of one-sided operations.

```
myRank = CS_Rank(CAF_WORLD)
numProcs = CS_Size(CAF_WORLD)
if (myRank < numProcs/2)
   myNewRank = myRank * 2
else
   myNewRank = (myRank-numProcs/2)*2+1
end if
call CS_Create(myworld, CAF_WORLD, .true., myNewRank)
```

Cartesian co-space

This is the most commonly used co-space type. For example, 2D Jacobi iteration may use a 2D Cartesian co-space with periodic boundary conditions and dimensions $N \times M$. Such a co-space can be constructed via

```
call CS_Create(cart_NxM, CAF_WORLD, 2, (/ N,M /) )
```

The neighbors can be obtained via the `CS_Neighbor` function, *e.g.*, the left neighbor is `CS_Neighbor(cart_NxM, 1, -1)`.

Programmers might find it convenient to use a preprocessor to declare shorter and more intuitive macros to specify the target for common communication successors. For example, `left(cart_NxM)` can be used to specify the left neighbor in a Cartesian co-space. It can be defined as the `CS_Neighbor(cart_NxM, 1, -1)` macro. Similarly, `right(cart_NxM)` can be defined as `CS_Neighbor(cart_NxM, 1, 1)`, `up(cart_NxM)` can be defined as `CS_Neighbor(cart_NxM, 2, 1)`, etc.

Graph co-space

Graph co-space can be used to express arbitrary communication topologies, for instance, a sum reduction in the NAS CG benchmark for a group of images, a transpose communication pattern for a 1D FFT [1], or the communication pattern for an unstructured mesh.

Graph co-space can also be used to express a communication topology for generalized block data decompositions. Figure 5.1 shows an example of such 2D decomposition in which the data is distributed in such a way that each process image i is given a single data

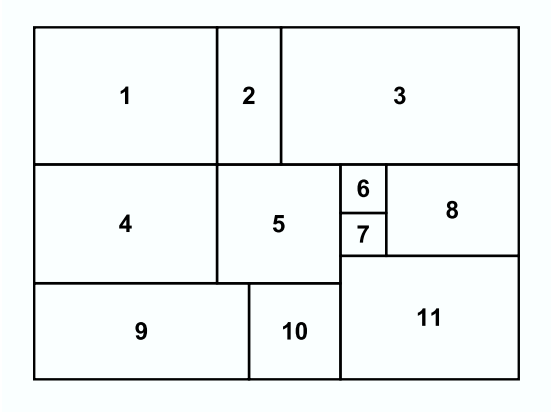


Figure 5.1 : An example of a generalized block distribution.

block denoted via i. One can use graph co-space to represent communication partners and express a shadow-region exchange, using edge classes to group all communication partners in the same spatial direction. For a 2D generalized block data decomposition, class 1 can stand for all communication partners on the left, 2 – on the right, 3 – on the top, and 4 – on the bottom. Figure 5.2 shows the code for graph co-space creation on image 5 and

```
!  co-space setup
numClasses = 4 !  1 is left, 2 is right, 3 is up, 4 is bottom
 if (CS_Image()==5) then
    numNbrs = (/ 1, 3, 2, 2 /) !  the number of neighbors for each class 1-4
    nbrs = (/ 4, 6, 7, 11, 2, 3, 9, 10 /) !  neighbor images of image 5
 end if
 ...  !  set up neighbors on other images
 call CS_Create(cs, CAF_WORLD, numClasses, numNbrs, nbrs)

 !  shadow region exchange
 call sync_all()
 do class = 1, 4 !  for every spatial direction
    do ni = 1, CS_GetNumNeighbors(cs, class) !  for every neighbor index
       nbr = CS_Neighbor(cs, ni, class) !  the communication partner
       !  perform communication; slb, sub - remote bounds; llb, lub - local bounds
       A(slb1(class,ni):sub1(class,ni), slb2(class,ni):sub2(class,ni))[nbr]=
                A(llb1(class,ni):lub1(class,ni), llb2(class,ni):lub2(class,ni))
    end do
 end do
 call sync_all()
```

Figure 5.2 : Shadow region exchange for a 2D generalized block data distribution.

shadow region exchange. Arrays `slb1`, `slb2`, `sub1`, and `sub2` contain the bounds of shadow regions on neighbor images; arrays `llb1`, `llb2`, `lub1`, and `lub2` arrays contain the bounds of the communicated tile inner data.

5.4 Properties of co-space neighbor functions

A one-sided operation O with the target expressed via the `CS_Neighbor` function corresponds to an edge in the directed graph that represents either Cartesian or graph communication topology. Let image `p` be the source and image `q` be the sink of this edge. Below, we show properties of the `CS_Neighbor` function that are essential for determining the source of O. Chapter 6 shows how, using these properties, to determine the origin image(s) of O for several communication patterns common for scientific codes. In turn, Chapter 7 uses the origin image information to convert barriers into point-to-point synchronization.

Cartesian co-space.

Let `q=CS_Neighbor(cs,axis,offset,p)`. If `q` is not equal to `NOIMAGE`, then `CS_Neighbor(cs,axis,-offset,q)` returns `p`.

In other words, if image `p` performs O to image `q` specified via `CS_Neighbor(cs,axis,offset)`, then image `q` is able to compute the origin image of O via `CS_Neighbor(cs,axis,-offset)`, which returns `p`.

Similarly, let `q=CS_Neighbor(cs,offsets,p)`. If `q` is not equal to `NOIMAGE`, then `CS_Neighbor(cs,-offsets,q)` returns `p`.

Graph co-space.

Let `q=CS_Neighbor(cs,idx,classid,p)`. If `q` is not equal to `NOIMAGE`, then execution of `CS_Neighbors(cs,srcNbrs,classid, predecessor,idx,q)` returns the set `srcNbrs` of images that are the source points of edges with class `classid`, class index `idx`, and sink `q`;
and `p` $\in$ `srcNbrs`. Clearly, for each member image `srcNbrs(i)`, where

`i`$\in[1,$`CS_GetNumNeighbors(cs,classid,predecessor,idx,q)`$]$, `CS_Neighbor(cs,idx,classid,srcNbrs(i))` returns `q`.

5.5 Implementation

The co-space abstraction is a powerful concept and, in our opinion, should be included into the CAF programming model. There are two feasible options. First, co-spaces can be CAF language abstractions. This choice would make the concept available in every implementation of a CAF compiler, but it would also require more implementation effort from vendors. Second, the co-space abstraction can be implemented as a standard CAF module. A CAF compiler; however, must understand the semantics of co-space interface functions to perform program analysis and optimization. Under this approach, some CAF compilers might not provide co-space abstraction support. In our opinion, more programmers' experience with co-spaces and more vendor CAF compiler implementations are required to make co-spaces a part of the CAF language.

Co-spaces expose topological properties of a group of images to the compiler. The information about local state is available to every image. However, the state on remote images, *e.g.*, edges of a graph co-space, is not. The co-space interfaces are not designed to enforce any implementation decisions or how much of the distributed state should be cached locally. The first choice is to cache all distributed state locally on every image, *e.g.*, during co-space creation. This would eliminate the overhead of contacting other images during remote information lookup. On the other hand, it might not be a feasible solution for massively parallel machines, because the representation of the relation could be large. The second choice is not to cache any information locally and contact remote images when necessary; however, with this option the overhead might be high. Alternatively, a dynamic caching scheme can be used to remember only some co-space properties of remote images, *e.g.*, s set of graph co-space neighbors of remote images that were retrieved during execution.

We extended `cafc` with a prototype support for group, Cartesian, and graph co-

spaces. The implementation uses a Fortran 95 module to declare the co-space types: `type(Group)` for group, `type(Cartesian)` for Cartesian, and `type(Graph)` for graph co-spaces. Each type has only one `integer(8)` field that is an opaque handle storing a pointer to the run-time co-space representation. The co-space interface functions are module subroutines; Fortran 95 module interfaces allow us to have several functions with the same name via overloading. The module subroutines are thin wrappers around C functions that implement the co-space logic. The current implementation of graph co-spaces caches the entire graph locally when a graph co-space is created.

We used co-spaces to express communication in Jacobi iteration and NAS MG and CG benchmarks. These codes were successfully optimized by the SSR optimization and yielded performance comparable to that of hand-coded versions with point-to-point synchronization. The details of SSR and experiments can be found in Chapter 7.

Chapter 6

Analyzing CAF Programs

CAF is an explicitly-parallel SPMD programming language. The programmer has great flexibility in data partitioning, communication, and synchronization placement as well as full control over the partitioning of computation. However, CAF is impenetrable to compiler analysis under the current language specification, making CAF programmers share the burden of optimizing CAF codes for portable high performance.

6.1 Difficulty of analyzing CAF programs

To analyze communication and synchronization in explicitly-parallel CAF programs, a compiler needs to relate control flow and reason about values of several program images. This is a difficult task, undecidable in the general case, because parallel control flow and each image values are defined by the programmer via variables that are local to each image. CAF offers very few opportunities to infer facts about other images from purely local information.

We focus on inferring the communication structure that is typical for a large class of scientific codes. Namely, nearest-neighbor codes where each process image communicates to a small subset of process images. We concentrate on detecting communication events (PUTs/GETs) that are executed by all images of a co-space[1], where the target image index is expressed via a co-space `CS_Neighbor` function with arguments that are the same on each process image of the co-space. Under certain conditions, such statements can be converted from one-sided PUTs/GETs synchronized with barriers into more efficient two-sided communication, which does not need barriers. In such cases, surrounding barriers

[1] Co-spaces are communication topologies explained in Chapter 5.

may be automatically removed from the code to yield better asynchrony tolerance and higher performance. To make such analysis possible, we explore extending CAF with two language constructs based on features in the Titanium language [66] in addition to the co-spaces introduced in Chapter 5.

6.2 Language enhancements

We explore extending CAF with two new CAF constructs: textual group barrier and group single-valued coercion operator. A textual group barrier ensures that all processes of a process group execute the same instance of the barrier. Group single-valued coercion operator specifies that a value is the same on every process that is a member of the group. A group is defined by a co-space of any type; we use term co-space and group interchangeably.

6.2.1 Textual group barriers

A *global* textual barrier is a synchronization primitive introduced in the Titanium language [66]. A global textual barrier must be executed by all processes and all processes must execute the same barrier statement. In a strongly typed language such as Titanium [66], it is possible to statically verify whether a program using only global textual barriers for synchronization is deadlock free. However, global textual barriers require that *all* processes execute the same barrier statement, which is a serious limitation for applications such as CCSM [50], working as a collection of independent but interacting groups of processes. Without strong type system in CAF, it is not possible to statically verify whether all global barriers are textual. Nonetheless, it is possible to detect some instances of textual barriers that may cause a deadlock.

The limitation of the global textual barriers and the lack of strong type system in CAF suggest that *textual group barriers* should be used for synchronization. A textual barrier of a co-space `cs` is a barrier statement that must be executed by all process images of co-space `cs`. It is specified in the program via `call barrier(cs)`, and we denote it as B_{cs}. A textual co-space barrier means that all process images of the co-space participate in

the barrier and signal arrival at the barrier by executing the same program statement. The compiler, in turn, can infer that some program statements, *e.g.*, in the same basic block as the barrier statement, are also executed by all process images of the co-space.

In CAF, a barrier has an implicit memory fence associated with it, so textual co-space barriers are statements at which the memories of all co-space images become *consistent*. Consistency across the memories means that all accesses to shared variables issued before the barrier have completed; such an access is a local co-array access, a GET, or a PUT. A barrier satisfies all data dependencies between all co-space images (inter-image data dependencies) and this fact is known by all co-space images upon return from a barrier call. We will rely on this observation in Chapter 7 to optimize synchronization while preserving program correctness by satisfying all inter-image dependencies using asymptotically more efficient point-to-point synchronization.

6.2.2 Group single values

The concept of *single-valued* (SV) expressions was introduced by Aiken and Gay [6]. A SV expression evaluates to the same value on *all* process images. Titanium [66] uses the `single` type qualifier to declare a variable as single-valued. Further, Titanium's type system enables one to statically prove that all such variables are assigned only single values.

Because CAF lacks a strong type system and Titanium's single values must be the same on *all* the processes, we explore extending CAF with a coercion operator `single(cs,exp)`, rather than directly borrowing Titanium's `single` type qualifier. `single(cs,exp)` serves as a compiler hint to indicate that expression `exp` evaluates to the same value on all process images that belong to co-space `cs`. The single-valued property can be propagated through the control flow graph (CFG) using static single assignment form (SSA), described in Section 3.4.4, to infer more single values that will be born during execution.

6.3 Inference of group single values and group executable statements

Titanium defines a set of type system inference rules [66] for global single values. The inference can be done via generating a set of data flow constraints using CFG and SSA and solving the constraints to obtain the maximal set of single values [6]. A subset of the inference rules and the inference algorithm can be adapted for CAF to infer co-space single values as well as *group-executable* (GE) statements that are textually executed either by all images of the group or by none; all other statements are non-group-executable (NGE).

6.3.1 Algorithm applicability

We use SV and GE properties to analyze communication events and convert expensive barrier synchronization into asymptotically more efficient point-to-point synchronization. This synchronization strength reduction transformation (SSR) is shown in detail in Chapter 7. We designed SSR to optimize real scientific codes. Since such codes usually use only structured control flow, we designed the SV & GE inference algorithm for only structured control flow. Under this assumption, SV & GE inference can be done as a forward propagation problem on CFG using SSA. For unstructured control flow, the inference can be done by adapting the solution presented in [6].

The forward propagation algorithm handles structured control constructs such as `IF-THEN-ELSE`, `IF-THEN`, and Fortran `DO` loops. Supporting this subset is sufficient to SSR-optimize all known to us CAF codes. The inference is done for a subroutine with textual co-space barrier B_{cs} statements and co-space single-valued coercion operators `single(cs,exp)`, where co-space `cs` is the subroutine invariant.

6.3.2 Forward propagation inference algorithm

We use both a CFG and SSA form to simultaneously forward propagate GE and SV properties. The SSA is used to propagate SV using a three-state lattice with top ($\top$), single-valued (SV), and bottom ($\bot$) states. $\top$ corresponds to unvisited state, SV means that the value is single-valued and $\bot$ means that the value may (we say *is*) not be single-valued (NSV). The

```
procedure initialize
    w ← ∅
    for each basic block b
        state(b) ← GE
        w ← w ∪ {b}
    for each SSA name v
        if x is defined on entry to the subroutine
            latval(x) ← ⊥
        else
            latval(x) ← ⊤
```

Figure 6.1 : SV & GE inference initialization step.

meet operator $\wedge$ is defined by rules: $Any \wedge \top = Any$, $SV \wedge SV = SV$, $Any \wedge \bot = \bot$. The result of the meet operator involving an SSA ϕ-function is the meet of its arguments. Constants and expressions coerced with `single(cs,exp)` are SV. A new SV can be born as a result of evaluating an expression, consisting only of SV terms and constants, in a GE statement. For example, if a, b and i are SV, the expressions $a+1$, $a+b(i)$, $b(2:i-1)$, and $a == 8$ yield a single-valued result for all process images in a co-space if the expressions are evaluated by *all* images of the co-space. However, the same expressions are non-single if they are not evaluated by all images of the co-space. The expressions $a + j$, $b(j)$, and $j == 8$ are non-single if j is non-single.

Our inference algorithm is an iterative fixed point optimistic algorithm. It uses the SSA form namespace, so each name x is defined exactly once. We associate a lattice cell with each SSA name x, denoted as $latval(x)$. $meet$ denotes the meet operator $\wedge$. Each basic block b has a state, denoted as $state(b)$, associated with it. $state(b)$ can be either group-executable (GE) or non-group-executable (NGE).

The initialization step $initialize$ shown in Figure 6.1 optimistically sets the state of each basic block to GE and adds the basic block to the worklist w. It initializes all lattice cells to $\top$ except for the SSA names defined on entry to the subroutine, which are initialized to $\bot$.

```
procedure propagate
    while w ≠ ∅
        changed ← ∅
        while w ≠ ∅
            select basic block b, b ∈ w
            w ← w − {b}
            for each φ-node Φ in b, evalPhiNode(b, Φ)
            for each statement s in b, evalStmt(b, s)
        w ← changed
```

Figure 6.2 : SV & GE inference propagation step.

The propagation step $propagate$ shown in Figure 6.2 symbolically evaluates each Φ-node ($evalPhiNode$) and each statement ($evalStmt$) of each basic block taken from the worklist w.

Lattice values propagate along SSA edges and the GE property propagates over CFG edges. If during evaluation any state of a basic block a changes, a is added to the worklist $changed$. The algorithm can only lower values and can only re-mark basic blocks as NGE, so it is monotonic and converges. It yields a conservative approximation of all co-space single values and GE basic blocks.

Figure 6.3 shows $evalPhiNode$ that evaluates an SSA Φ-node. When control flow

```
procedure evalPhiNode(b, Φ)
    // b is a basic block, Φ is a φ-node x = φ(...)
    if state(b) = GE
        v ← evalExpr(φ(...))
    else
        v ← ⊥
    if latval(x) ≠ v
        // propagate v to uses of x along SSA edges
        for each basic block z containing uses of x
            changed ← changed ∪ {z}
        latval(x) ← v
```

Figure 6.3 : Evaluation of a Φ-node.

```
procedure evalStmt(b, s)
    // b is the basic block, s is the statement
    if s is an assignment, call evalAssignment(b, s)
    if s is an IF-THEN-ELSE, call evalIfThenElse(b, s)
    if s is a DO loop, call evalDo(b, s)
    if state(b) = NGE
        for each dominator tree successor z of b such that state(z) = GE
            state(z) ← NGE
            changed ← changed ∪ {z}
```

Figure 6.4 : Evaluation of a statement and propagation of the NGE property.

merges values in an SSA ϕ-function, the resulting value is non-single ($\perp$) if either the ϕ-function is executed in a NGE basic block or one of its arguments is non-single ($\perp$).

Figure 6.4 shows $evalStmt$ that evaluates a statement of type assignment, `IF-THEN-ELSE`, or `DO`. Subroutine (and function) calls modify lattice values of SSA names according to their side effects; this information is incorporated in the SSA form. If a basic block is NGE, the non-group-executable property propagates to all of its successors in the dominator tree, which are marked as NGE.

Figure 6.5 shows $evalAssignment$ that evaluates an assignment statement. The RHS is evaluated iff the statement is GE, otherwise the result if non-single-valued ($\perp$). If the

```
procedure evalAssignment(b, s)
    // s is a scalar assignment x = e
    if state(b) = GE
        v ← evalExpr(e)
    else
        v ← ⊥
    if latval(x) ≠ v
        // propagate v to uses of x along SSA edges
        for each basic block z containing uses of x
            changed ← changed ∪ {z}
        latval(x) ← v
```

Figure 6.5 : Evaluation of an assignment statement.

procedure $evalIfThenElse(b, s)$
  // let s_c be the conditional expression
  // let s_t be the CFG successor BB of the true branch and s_f – of the false branch
  if $state(b) = GE$
    $c \leftarrow evalExpr(s_c)$
  else
    $c \leftarrow \perp$
  if $c = \perp$ and $state(s_t) = GE$
    // make if-branches NGE
    $state(s_t) \leftarrow NGE$
    $state(s_f) \leftarrow NGE$
    $changed \leftarrow changed \cup \{s_t\} \cup \{s_f\}$

Figure 6.6 : Evaluation of an `IF-THEN-ELSE` statement.

value of LHS x lattice cell changes, this fact is propagated to all uses of x along the SSA edges by adding the basic blocks of uses to the worklist $changed$. Note that the algorithm can be extended to handle array element and section expressions.

Figure 6.6 shows $evalIfThenElse$ that evaluates an `IF-THEN-ELSE` statement. If the conditional of the `IF-THEN-ELSE` (or `IF-THEN`) statement is SV and the statement is GE, the CFG successors that correspond to the `THEN`- and `ELSE`-branches must be GE; otherwise, the successors are NGE. Note that a `SELECT` statement can be handled similarly.

Figure 6.7 shows $evalDo$ that evaluates a `DO` statement. In our CFG, a `DO` loop is always preconditioned. The `DO` loop entry node has only two CFG successors: the first corresponds to the first node of the loop body, the second corresponds to a no-op statement inserted right after the `ENDDO` of the `DO` loop. The range or loop control expression of a `DO` loop is SV iff the lower bound, upper bound, and stride are SV; otherwise, the range is NSV. The `DO` loop rules for GE propagation treat the loop as a region of CFG. If the loop statement is GE and even if the loop conditional is NSV, meaning that the loop can execute different numbers of times on different process images, the control flow of all process images is GE after the `DO` loop execution (right after `ENDDO`). If a `DO` loop entry

```
procedure evalDo(b, s)
    // let lb, ub, str be lower bound, upper bound, and stride of the DO loop range
    // let s_b be the first basic block of the DO loop body
    if state(b) = GE
        // compute the lattice value of the range
        r ← meet(evalExpr(lb), evalExpr(ub), evalExpr(str))
    else
        r ← ⊥
    if r = ⊥ and state(s_b) = GE
        // make the DO loop body NGE
        state(s_b) ← NGE
        changed ← changed ∪ {s_b}
```

Figure 6.7 : Evaluation of a DO statement.

node is GE and the loop control expression is SV, both successors are GE. If a DO loop entry node is GE and the loop control expression is NSV, the ENDDO successor is GE, but the loop body CFG successor is NGE. Note that it is possible to extend the algorithm to handle CYCLE and EXIT loop control statements as well as WHILE loops.

Figure 6.8 shows $evalExpr$ that evaluates an expression. Constants are SV. Values

```
procedure evalExpr(e)
    if e is a constant, return SV
    if e is single(cs,exp), return SV
    if e is an SSA name x, return latval(x)
    if e is a unary operator e = uop(e_1), return evalExpr(e_1)
    if e is a binary operator e = bop(e_1, e_2)
        return meet(evalExpr(e_1), evalExpr(e_2))
    if e is a n-ary operator e = op(e_1, ..., e_n)
        return meet(evalExpr(e_1), ..., evalExpr(e_n))
    if e is a φ-function φ(e_1, ..., e_n), return meet(latval(e_1), ..., latval(e_n))
    if e is a co-space function or CAF intrinsics, evaluate according to its semantics, e.g.,
        if e is CS_IsMember(cs1)
            if cs1 = cs, return SV, else return ⊥
```

Figure 6.8 : Evaluation of an expression.

coerced with `single(cs,exp)` are SV. The SSA name value is its lattice cell value. $evalExpr$ evaluates the value of operators and ϕ-functions using the meet operator. Compiler recognizable functions, such as co-space functions and CAF intrinsics, are handled according to their semantics; Figure 6.8 shows an example for the `CS_IsMember` function.

In addition to the SV & GE inference, the algorithm performs limited program verification. For correct program execution, each basic block containing textual co-space barrier B_{cs} call must be GE. Similarly, each co-space single value coercion operator `single(cs,exp)` must be evaluated in a GE basic block. If either condition is violated, the algorithm warns the programmer about a potential problem and specifies to the next analysis phase (SSR) that the subroutine is not analyzable.

We implemented this inference algorithm in `cafc`. The algorithm is sufficient for the analysis of communication structure in most CAF scientific codes, and the SSR optimization uses its results. A more general, constrained-based, algorithm can be used to extend the inference to non-structured control flow, if the need arises.

6.4 Analysis of communication structure

Most parallel applications do not express communication in an arbitrary way, but rather structure communication in a certain way. For example, many nearest-neighbor codes work in phases: processes perform computation and then exchange boundaries with their neighbor processes. Our analysis algorithm focuses on detecting such structured communication.

We assume that the inference of group-executable (GE) statements and group single-valued (SV) values has been done as described in Section 6.3. We say that a communication event (PUT/GET) is *analyzable* if it falls into one of the communication patterns described in the rest of this chapter. In Chapter 7, we show how SSR optimizes two communication Patterns 6.1 and 6.2, described in the following sections.

6.4.1 Analyzable group-executable PUT/GET

Pattern 6.1 (Analyzable GE PUT/GET) *A group-executable PUT/GET with the target image index specified via a co-space* `CS_Neighbor` *function with group single-valued arguments.*

This is a common communication pattern found in kernels of many nearest-neighbor scientific codes such as NAS benchmarks, Jacobi iteration, LBMHD, etc. For example, each process performs communication to a neighbor process in the same spatial direction. Such communication patterns are typical for a Cartesian processor grid, so the Cartesian co-space does an excellent job of capturing the properties of the communication topology and expressing communication relative to each process image via the co-space `CS_Neighbor` function. However, a graph co-space is also used, for instance to perform group reductions among a collection of neighbors in the NAS CG benchmark (see Figure 7.45) or a transpose in a distributed FFT.

Let us consider Jacobi iteration on a 2D matrix $N \times K$ with periodic boundary conditions. The matrix is decomposed into slabs along the second dimension and equally distributed among all process images. The size of each slab is $N \times M$, $M = \frac{K}{\texttt{num\_images}()}$. The matrix is represented by two co-arrays `real(8)::a(1:N,0:M+1)` and `real(8)::b(1:N,0:M+1)`. Each Jacobi iteration locally computes a five-point stencil into co-array `a` using values from co-array `b`; on the following iteration, the roles of `a` and `b` are changed. The remote data necessary for local computation is stored in the shadow regions. `a(1:N,0)` contains border values that correspond to `a(1:N,M)` of the left neighbor; `a(1:N,M+1)` contains border values that correspond to `a(1:N,0)` of the right neighbor. After the local computation is done, each process image updates its left and right neighbor shadow regions with its border values using PUT, as shown in Figure 6.9. Figure 6.10 shows the communication done by one of the process images. It is natural to organize all process images using a 1D Cartesian co-space `cs` with periodic boundaries to represent communication topology and to express neighbors for the shadow region exchange phase.

```
! perform local computation: a(i,j) = ...

! exchange shadow regions
call barrier(cs)
a(:,  0)[CS_Neighbor(cs,1,+1)] = a(:,M)   ! PUT to the right neighbor
a(:,M+1)[CS_Neighbor(cs,1,-1)] = a(:,1)   ! PUT to the left neighbor
call barrier(cs)

! perform local computation: b(i,j) = ...
```

Figure 6.9 : Jacobi iteration shadow region exchange for periodic boundaries.

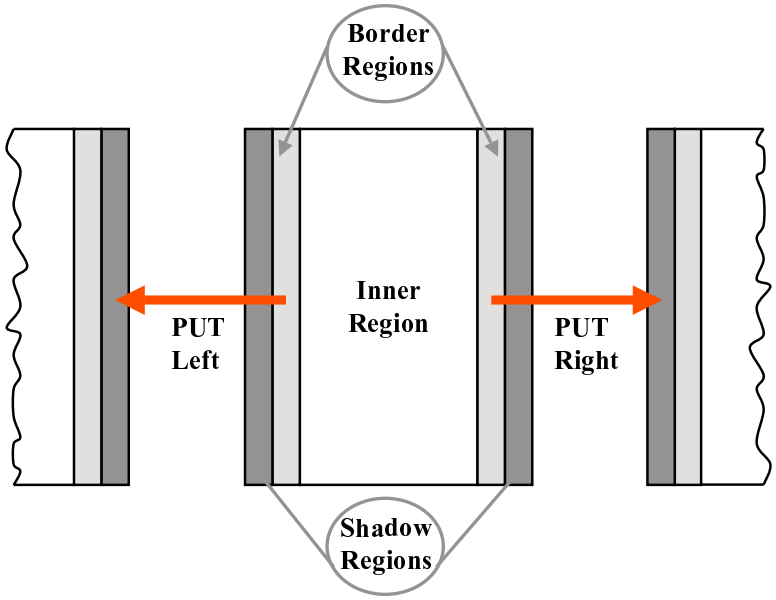


Figure 6.10 : Jacobi shadow region exchange for one processor.

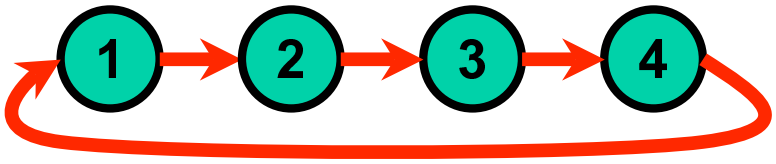


Figure 6.11 : Periodic boundary communication to the right for four processes.

In the first PUT statement, every process image of the co-space `cs` communicates data to its right neighbor. The pattern is visualized in Figure 6.11 for an execution on four processors; for each process image, the right arrow shows the target process image of communication. The pattern is known on *every* process image of co-space `cs` because the statement is GE and the target of the PUT is expressed via `CS_Neighbor(cs,a,o)` function

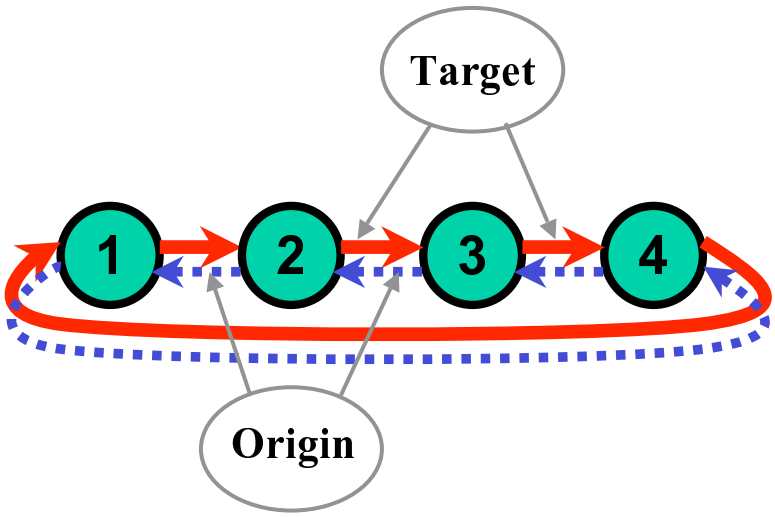


Figure 6.12 : Targets and origins of communication to the right for Jacobi iteration with periodic boundaries.

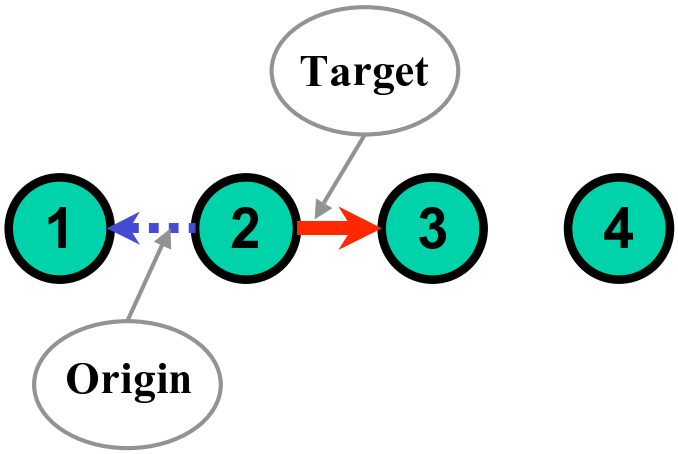


Figure 6.13 : The target (image 3) and the origin (image 1) of communication to the right for process image 2.

call with SV arguments: axis `a` and offset `o`. Each co-space process image can compute the origin image of this communication *locally* using the `CS_Neighbor(cs,a,-o)` function call; in this case, the origin is the left neighbor, and the "inversion" of the pattern showing the origin of communication is depicted as dotted left arrows in Figure 6.12. To clarify, Figure 6.13 shows the target and the origin process images for process image 2.

It is important that the arguments are single-valued and the communication is GE be-

cause each process image can compute the origin of communication from *purely local* information without the need to contact another image. If the statement were not GE, an image would not know what images participate in the communication event. If the arguments were not single-valued, an image would not know how to compute the origin image of communication using only local values. In either case, it would need to contact other process image(s) to determine which is the origin of communication, incurring high overhead and rendering any optimization ineffective. In some sense, an analyzable GE PUT/GET creates a topology "layer" (*e.g.*, see Figure 6.11), a sub-topology of the co-space, determined by the SV layer arguments to the `CS_Neighbor` function. The entire layer is known to each process image locally. The origin(s) of communication initiated onto process image `p` is the source(s) of the sub-topology directed graph edge incident on `p`.

We summarize these ideas in the following observations. Note that there is only one origin image for a GE analyzable PUT/GET on a Cartesian co-space, but that there can be several origins of communication for a GE analyzable PUT/GET on a graph topology.

Observation 6.1 *For a Cartesian co-space* `cs`,

(a) *the origin image index of an analyzable GE PUT/GET with the target image index expressed via* `CS_Neighbor(cs,a,o)` *function with the co-space single-valued arguments can be computed as* `CS_Neighbor(cs,a,-o)`, *where* `a` *denotes the axis parameter, and* `o` *denotes the offset parameter.*

(b) *the origin image index of an analyzable GE PUT/GET with the target image index expressed via* `CS_Neighbor(cs,ov)` *function with the co-space single-valued arguments can be computed as* `CS_Neighbor(cs,-ov)`, *where* `ov` *denotes the vector of offsets parameter.*

Observation 6.2 *For a graph co-space* `cs`*, there can be several origin image indices of an analyzable GE PUT/GET with the target image index expressed via the* `CS_Neighbor(cs,nbrIndex,classid)` *function call with SV arguments. Each image can locally compute the set of origin image indices* `orgNbrs` *by calling* `CS_Neighbors(cs,orgNbrs,predecessor,classid,nbrIndex)`.

Run-time vs. source-level guards for communication/synchronization

The Jacobi iteration example above uses a 1D Cartesian co-space with periodic boundaries, and every image of the co-space executes communication. But what if the communication topology is Cartesian without periodic boundaries? There are two possible ways to express the communication as well as synchronization when not all images of co-space participate in the event.

```
! perform local computation: a(i,j) = ...

! exchange shadow regions
call barrier(cs)
if (CS_HasNeighbor(cs,1,+1)) then
  a(:,0)[CS_Neighbor(cs,1,+1)]=a(:,M)  ! PUT to the right, if it exists
endif
...
call barrier(cs)

! perform local computation: b(i,j) = ...
```

Figure 6.14 : Jacobi iteration shadow region exchange for non-periodic boundaries.

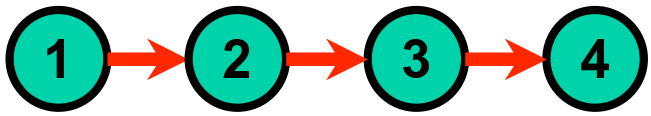


Figure 6.15 : Non-periodic boundary communication to the right for four processes.

First, the programmer can guard the communication explicitly as shown in Figure 6.14. On four process images, this induces a communication pattern visualized in Figure 6.15. The compiler can recognize this pattern and determine the origin of communication on every co-space image as shown in Figure 6.16 and visualized in Figure 6.17 with dotted

```
if (CS_HasNeighbor(cs,1,-1)) then     ! if left exists
  origin = CS_Neighbors(cs,1,-1)      ! left neighbor
else
  origin = NOIMAGE                    ! no left neighbor
endif
```

Figure 6.16 : The origin image index for the communication to the right with non-periodic boundaries.

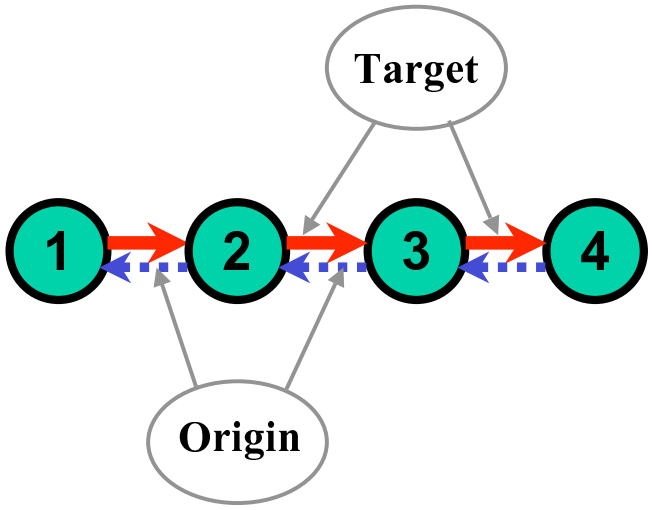


Figure 6.17 : Targets and origins of communication to the right for Jacobi iteration with non-periodic boundaries.

left arrows. However, the programmer has already specified the fact that the neighbors do not exist for the co-space border images when (s)he created the co-space; in our example, via non-periodic boundaries.

An alternative approach is to delegate the guard handling to the run-time layer and avoid communication/synchronization guards in the source code altogether. The `CS_Neighbor` functions return the process image index if the neighbor image exists, otherwise they return the special `NOIMAGE` value. Communication and synchronization primitives interpret the `NOIMAGE` value, specified as the target, as a no-op rather than a real communication/synchronization operation. This is analogous to the approach adopted by MPI for handling non-existing processor ranks in send/receive [112, 62].

For example, the Jacobi iteration code with non-periodic boundaries shown in Fig-

ure 6.14 will be exactly the same as the one with the periodic boundaries, without guards in the source code; the run-time will know how to perform communication correctly based on the co-space properties and co-space interpretation.

The run-time handling of guards for communication/synchronization improves programmability because it removes guards from the source program. It also simplifies compiler analysis. The compiler can analyze (and optimize) such communication in the same way as for an analyzable GE PUT/GET of Pattern 6.1.

6.4.2 Analyzable non-group-executable PUT/GET

Pattern 6.2 (Analyzable NGE PUT/GET) *A non-group-executable PUT/GET with the target image index specified via a co-space* `CS_Neighbor` *function with group single-valued arguments.*

This is a less common communication pattern and can be found, for example, in the NAS MG extrapolation subroutine shown in Figure 7.39. The relevant piece of code is shown in Figure 6.18.

The guard `give_ex(axis,level)` is not single-valued, so not all co-space images execute communication, and it is not possible to infer the origin of communication from only local information. However, if the arguments of the `CS_Neighbor` function are SV, it means that they are available and SV in one of the group-executable dominators of the communication event basic block. If the communication operation were moved outside the `IF-THEN` in Figure 6.18, it would become an analyzable GE PUT/GET (Pattern 6.1) and

```
...
call barrier(cs)
! axis is single-valued
if (give_ex(axis, level)) then  ! non-single-valued guard
  ... ! pack data into buffM(1:buff_len,1)
  buffM(1:buff_len,2)[CS_Neighbor(cs,axis,-1)] = buffM(1:buff_len,1)
endif
...
call barrier(cs)
```

Figure 6.18 : Non-single-valued guard in NAS MG extrapolation subroutine.

could be optimized similarly. We will rely on this observation in Chapter 7 to optimize an analyzable NGE PUT/GET for structured control flow.

6.4.3 Other analyzable communication patterns

We show several other communication patterns found in CAF codes that we studied and sketch how they can be optimized.

Language-level naive broadcast/reduction

The following code fragment shows a naive implementation of a broadcast in CAF. It can also be coded using a graph co-space.

```
call barrier()
if (this_image() == 1) then
  do i = 2, num_images()
    a(i)[i] = a(1)
  enddo
endif
call barrier()
```

A CAF compiler could determine that only image 1 performs communication to every other image $[2, \texttt{num\_images}()]$. This pattern can be replaced by an efficient, platform-tuned library broadcast subroutine. A less preferable solution is to replace barriers with point-to-point synchronization as shown below:

```
if (this_image() == 1) then
  do i = 2, num_images()
    call wait(i)
    a(i)[i] = a(1)
    call notify(i)
  enddo
else
  call notify(1)
  call wait(1)
endif
```

Language-level naive implementations of reductions can be handled similarly. Note that language-level naive reduction/broadcast can also be optimized for a group of images.

```
!  notify every neighbor that it is safe to update my shadow regions
do class = 1, 4 !  1 is left, 2 is right, 3 is up, 4 is bottom
    numOrgNbrs = CS_GetNumNeighbors(cs,class,predecessor) !  number of origins
    call CS_Neighbors(cs,orgNbrs,class,predecessor) !  origins of the class
    call notify(orgNbrs,numOrgNbrs) !  notify all origin neighbors of the class
end do

!  shadow region exchange, no barriers
do class = 1, 4 !  for every spatial direction
    do ni = 1, CS_GetNumNeighbors(cs,class) !  for every neighbor index
       nbr = CS_Neighbor(cs,ni,class) !  the communication partner
       !  wait permission to overwrite remote shadow region
       call wait(nbr)
       !  perform communication; slb, sub - remote bounds; llb, lub - local bounds
       A(slb1(class,ni):sub1(class,ni), slb2(class,ni):sub2(class,ni))[nbr]=
                A(llb1(class,ni):lub1(class,ni), llb2(class,ni):lub2(class,ni))
       !  indicate completion of remote shadow region update
       call notify(nbr)
    end do
end do

!  wait for every neighbor to finish updating my shadow regions
do class = 1, 4
    numOrgNbrs = CS_GetNumNeighbors(cs,class,predecessor) !  number of origins
    call CS_Neighbors(cs,orgNbrs,class,predecessor) !  origins of the class
    call wait(orgNbrs,numOrgNbrs) !  wait for all origin neighbors of the class
end do
```

Figure 6.19 : Shadow region exchange for a 2D generalized block data distribution expressed using point-to-point synchronization.

Generalized block distribution

A CAF compiler could detect communication patterns similar to the one shown in Section 5.3 for the shadow region exchange of a 2D generalized block data distribution. Detailed explanation of this example is available in Section 5.3 (see graph co-space usage). The code for shadow region exchange shown in Figure 5.2 can be transformed to use point-to-point synchronization instead, as shown in Figure 6.19; note that `notify()` and `wait()` accept sets of process image indices.

Chapter 7

Synchronization Strength Reduction

Synchronization strength reduction (SSR) is an optimization that replaces textual barrier-based synchronization with cheaper point-to-point synchronization while preserving the meaning of the program. This chapter presents a procedure-scope SSR algorithm for optimizing CAF codes. Code generated using SSR for several benchmarks delivered performance comparable to that of hand-optimized codes using point-to-point synchronization.

7.1 Motivation

A textual co-space barrier is conceptually the simplest synchronization primitive to use. Textual barrier statements divide program text and execution into phases or epochs that are the same for all of co-space members. We say that an invocation of a textual barrier closes one epoch and opens another epoch. In CAF, execution of a barrier for co-space C ensures that all shared accesses done by each process image of C destined to data co-located with any process image of C in the preceding epoch have completed before any such an access in the following epoch. Hence, the programmer does not need to synchronize individual accesses between members of the group; the barrier synchronizes all of them. For example, $Barrier2$ in Figure 7.1 enforces the ordering of the PUT (`a[q]=x`) and GET (`y=a[q]`) by synchronizing all process images. In compiler terms, an invocation of a textual barrier ensures that all local and inter-image data dependencies "crossing" (the end points belong to different epochs) the barrier are preserved. However, a barrier delays all images until the slowest one has arrived and might synchronize images that do not need to be synchronized. In Figure 7.1, only images `p`, `q`, and `r` need to be synchronized for the communication shown. Note that the arrows show the communication direction for the origin process

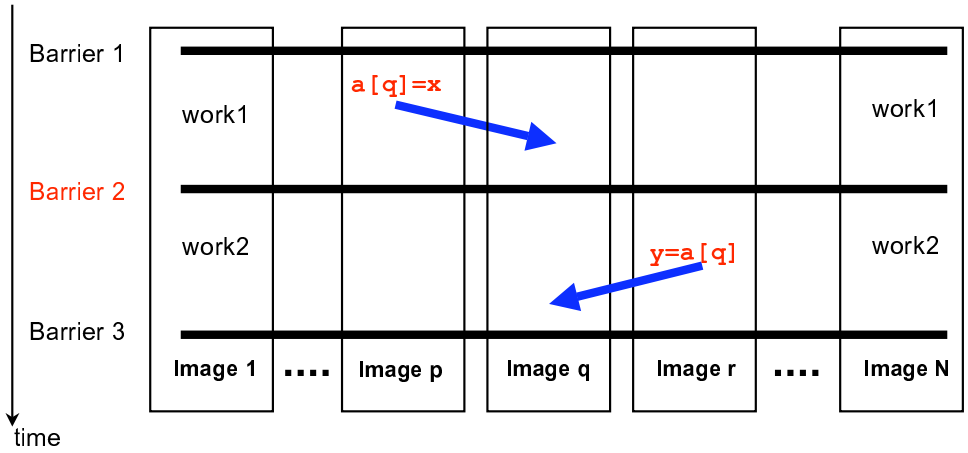


Figure 7.1 : Synchronization with textual barriers.

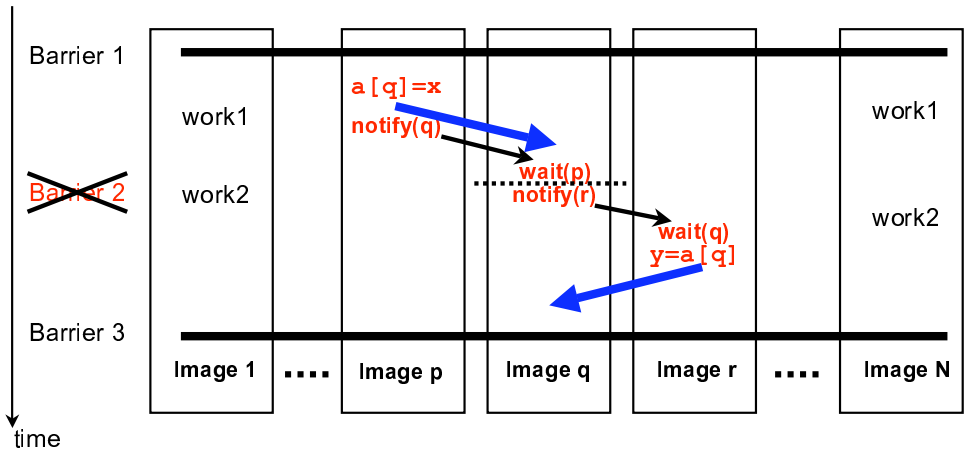


Figure 7.2 : Synchronization with notify/wait.

image that initiates communication to the target process image where the memory being accessed is located; for a PUT, the communication direction coincides with the direction of the data movement; for a GET, the communication direction is the opposite of the data movement direction. Oversynchronized codes are not asynchrony tolerant and result in suboptimal performance as we showed in prior studies [30, 47, 48, 31, 33].

As an alternative to barriers, programmers can use unidirectional point-to-point `notify`/`wait` synchronization (see Section 3.1), which scales much better. Figure 7.2 shows how point-to-point synchronization can be used to synchronize p and r (and q) with-

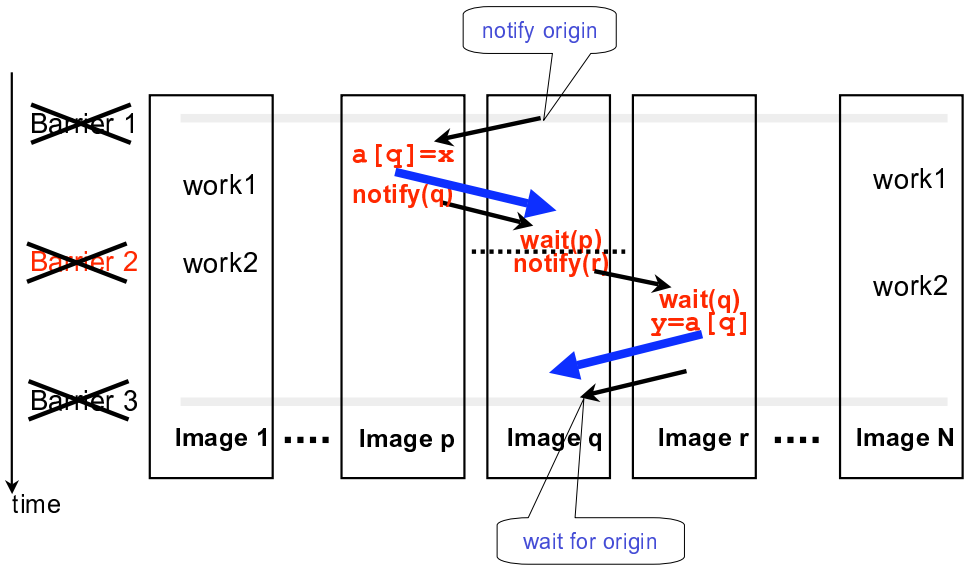


Figure 7.3 : Synchronization with textual barriers.

out $Barrier2$. If it is also safe to use notify/wait instead of $Barrier1$ and $Barrier3$, the code can be transformed as shown in Figure 7.3, which would likely be much faster than the original code using barriers shown in Figure 7.1. However, developing codes using point-to-point synchronization is hard because programmers must synchronize individual shared data accesses and, in some cases, carefully orchestrate `notify`/`wait` to obtain best performance.

The SSR optimization enables programmers to use textual barriers for synchronization. Using SSR, the compiler replaces these barriers with point-to-point synchronization, preserving program correctness while improving performance and scalability. SSR replaces a barrier with point-to-point synchronization only between images that *may* have inter-image data dependencies.

7.2 Intuition behind SSR

Using point-to-point synchronization correctly requires knowing the origin and target of a communication event (PUT/GET), or event for short. However, in an explicitly-parallel

PGAS language such as CAF, the programmer specifies only the target of each communication, but not the origin. It is the job of the compiler to infer the origin of communication from program code, which is difficult, undecidable in general case. In Chapter 6, we present a novel technique that enables one to infer origin(s) of communication patterns that are typical in a large class of the nearest-neighbor scientific codes. The analysis uses a combination of a co-space, textual co-space barriers, and co-space single-valued expressions to determine the communication structure for two Patterns 6.1 and 6.2, stated in Chapter 6. We restate the patterns here:

- **Analyzable group-executable PUT/GET.**
  A group-executable (GE) PUT/GET with the target image index specified via a co-space `CS_Neighbor` function with group single-valued arguments.

- **Analyzable non-group-executable PUT/GET.**
  A non-group-executable (NGE) PUT/GET with the target image index specified via a co-space `CS_Neighbor` function with group single-valued arguments.

Here, we focus on optimizing these two communication patterns by converting barrier-based synchronization into point-to-point synchronization, if legal and profitable. Initially, we will not consider moving the communication or changing the communication primitive. We later consider such optimizations in Section 7.9.

7.2.1 Correctness of SSR for analyzable group-executable PUTs/GETs

Let us consider how an analyzable group-executable PUT/GET can be optimized in a straight line code with several textual co-space barriers. We consider a group-executable communication pattern where each process image of a 4×2 Cartesian co-space `cs` with periodic boundaries PUTs data to its right neighbor along the first dimension. We denote `CS_Neighbor(cs,1,+1)` as `right(cs)`, and `CS_Neighbor(cs,1,-1)` as `left(cs)`. Figure 7.4 (a) presents pseudocode for this communication. Figure 7.5 shows a visualization of this pattern for eight process images.

```
1  ... = a
2  call barrier()


3  a[right(cs)] = ...


4  call barrier()
5  ... = a
```

(a) GE PUT to the right

```
1   ... = a
    ! former barrier
2a  call notify(left(cs))
2b  call wait(right(cs))
3   a[right(cs)] = ...
4a  call notify(right(cs))
4b  call wait(left(cs))
    ! former barrier
5   ... = a
```

(b) permission and completion notify/wait pairs

Figure 7.4 : A PUT to the right for a 4×2 Cartesian co-space with periodic boundaries.

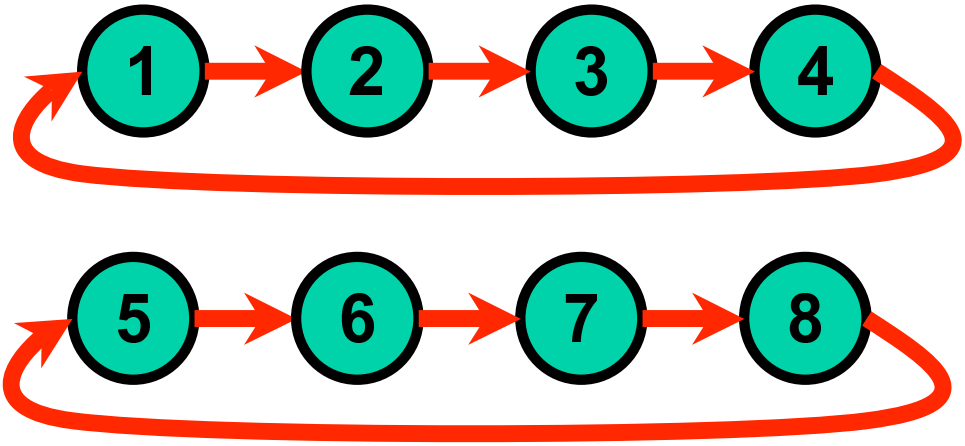


Figure 7.5 : A PUT to the right neighbor on a 4×2 Cartesian co-space with periodic boundaries.

The compiler can infer the origin of this communication event, which is the neighbor process image on the left, as shown in Section 6.4, and insert two notify/wait pairs as shown in Figure 7.4 (b) to synchronize the event. When it is safe (see below), the surrounding barriers can be removed.

The notify/wait pair in lines 2a and 2b of Figure 7.4 (b) gives "permission" to access data on the right co-space neighbor after the neighbor finishes accessing co-array a locally. We call this synchronization the *permission notify*, n_p, and *permission wait*, w_p; n_p/w_p denotes a permission pair. Figure 7.6 shows the permission notify and permission wait operations involving image y. The permission notify issued by the process image y tells

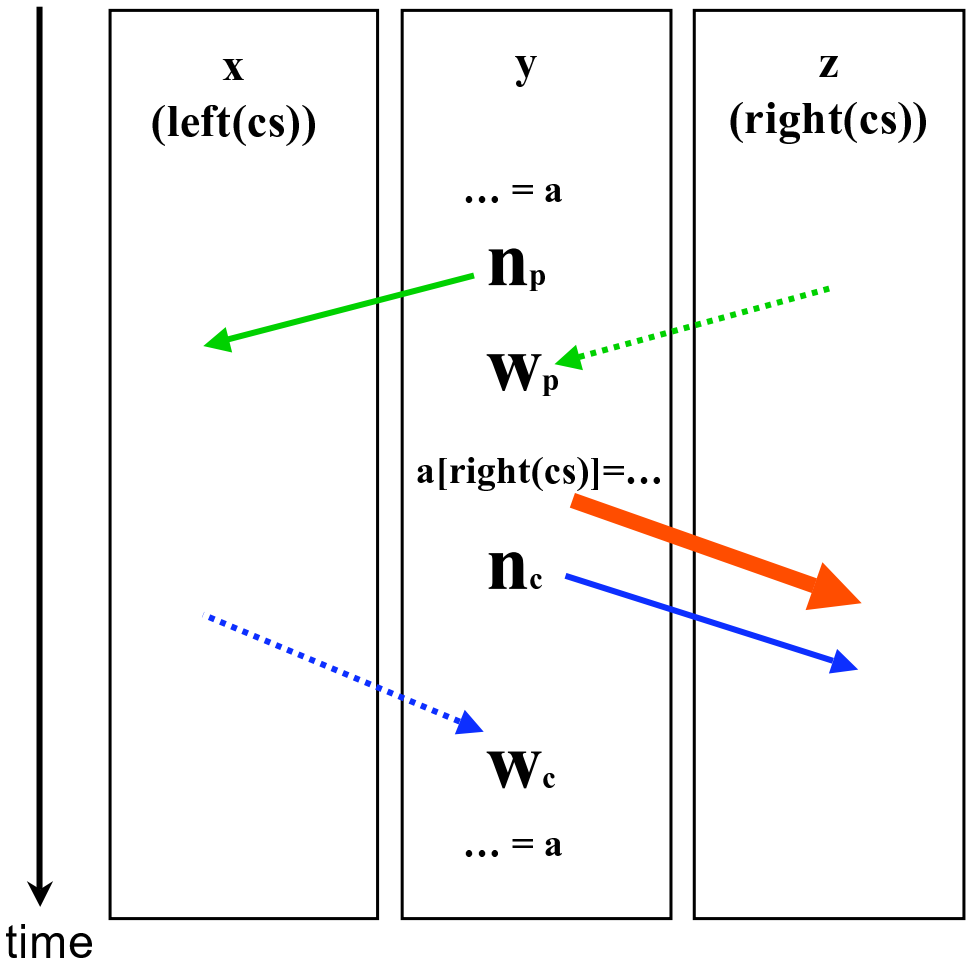


Figure 7.6 : Synchronization with direct communication partners (relative to y view).

the origin image of communication x, which is the left neighbor of `y` in co-space `cs`, that it is safe to access co-array `a` on y. Execution of the permission wait by y waits for a permission notify from the target image of communication z, which is the right neighbor of `y` in co-space `cs`, that tells y that it is safe to access co-array `a` on z. It is safe for an image to access co-array `a` on its right neighbor after the permission wait completes.

The notify/wait pair in lines 4a and 4b of Figure 7.4 (b) signals "completion" of the data access to the target image, so that the target image will be able to safely access co-array `a` locally. We call this synchronization the *completion notify*, n_c, and *completion wait*, w_c; n_c/w_c denotes a completion pair. Figure 7.6 shows the completion notify and completion

wait operations involving image `y`. The completion notify issued by y signals to the target image z that y has finished accessing z's co-array `a`. Execution of the completion wait by y waits for a completion notify from the origin image x that tells y that x has finished accessing co-array `a` on y. It is safe for an image to access `a` locally after the completion wait completes.

The permission & completion pairs safely synchronize the target and origin images of communication (both PUTs and GETs) by enforcing inter-image data dependencies with the *direct* communication partners. The permission pair ensures that the read of `a` on line 1 of Figure 7.4 (b) by each process image finishes before its left neighbor overwrites `a` with the PUT `a[right(cs)]` on line 3; the completion pair ensures that this PUT finishes before the read of `a` on line 5. However, barrier-based synchronization does more. It also enforces *transitive* inter-image data dependencies.

Let us consider an example of communication to the right neighbor followed by communication to the upper neighbor. Let `up(cs)` denote `CS_Neighbor(cs,2,+1)`, and `down(cs)` denote `CS_Neighbor(cs,2,-1)`. Figure 7.7 (a) shows the pseudocode. Consider the barrier on line 4. As shown in Figure 7.7 (b), the barrier can be replaced with a completion pair for the communication to the right inserted before the former barrier, on lines 4a and 4b, and a permission pair for the communication to the upper neighbor inserted after the former barrier, on lines 4c and 4d. This enforces transitive dependencies. Thinking relative to some process image, we notice that the transitive dependencies are enforced by the sequential execution of the completion wait, `call wait(left(cs))`, on line 4b and the permission notify, `call notify(down(cs))`, on line 4c. This synchronization ensures that the PUT `a[right(cs)]` has completed on an image before its value is read by the GET `a[up(cs)]`. Figure 7.8 shows communication from image 1 to image 2 (arrow with label (1)) followed by communication from image 6 to image 2 (arrow with label (2)). Figure 7.9 shows the time diagram of how image 2 enforces an inter-image dependence between image 1 that writes `a[2]` and image 6 that reads `a[2]` with the pattern of notify/wait synchronization shown in Figure 7.7 (b). Other process images

```
1   a = a + 1
2   call barrier()

3   a[right(cs)] = ...

4   call barrier()

5   ... = a[up(cs)]

6   call barrier()
7   a = a + 1
```

(a) PUTs to the right and above

```
1   a = a + 1
2   call barrier()
    ! permission notify(left) ?
    ! permission wait(right)  ?
3   a[right(cs)] = ...
4a  call notify(right(cs))  ! completion
4b  call wait(left(cs))     ! completion
    ! former barrier
4c  call notify(down(cs))   ! permission
4d  call wait(up(cs))       ! permission
5   ... = a[up(cs)]
    ! completion notify(up)  ?
    ! completion wait(down)  ?
6   call barrier()
7   a = a + 1
```

(b) enforcing transitive inter-image dependencies

Figure 7.7 : Communication to the right neighbor followed by communication to the upper neighbor for a 4×2 Cartesian co-space with periodic boundaries.

perform these operations as well with their communication partners, but only the operations incident on image 2 are shown in Figure 7.9. The thick arrows denote the direction of communication, not the direction where data is moving; thin arrows denote unidirectional point-to-point synchronization messages. The dependence between image 1 and image 6 is enforced *not* by a direct synchronization between them but rather transitively via ordering of the completion pair between images 1 and 2 and the permission pair between images 2 and 6 that results from the execution order of the completion wait on image 2 before the permission notify on image 2.

Since the permission & completion pairs enforce all data dependencies between accesses to co-array `a` on lines 3 and 5, they are sufficient to preserve the meaning of the program, and the barrier on line 4 can be removed. This would usually result in faster execution because the code uses one-way synchronization messages with shorter critical path than that of barrier synchronization. The question is: should we also insert a permission pair to synchronize the PUT on line 3 and a completion pair to synchronize the GET on line 5 instead of the barriers on lines 2 and 6, respectively? If all data dependencies crossing

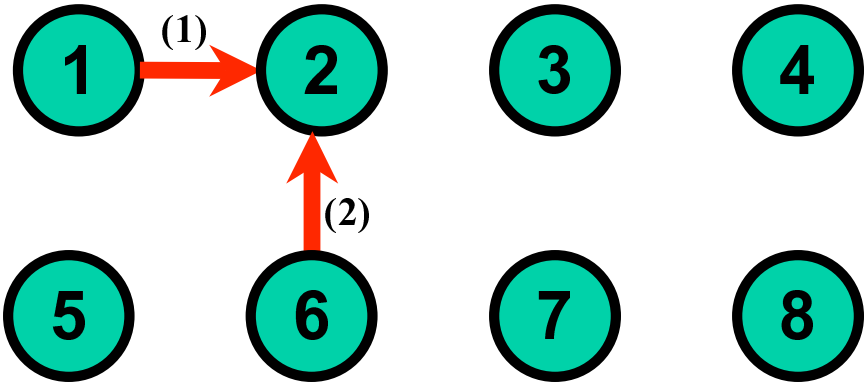


Figure 7.8 : Communication for images 1 and 6 accessing the same co-array `a[2]`.

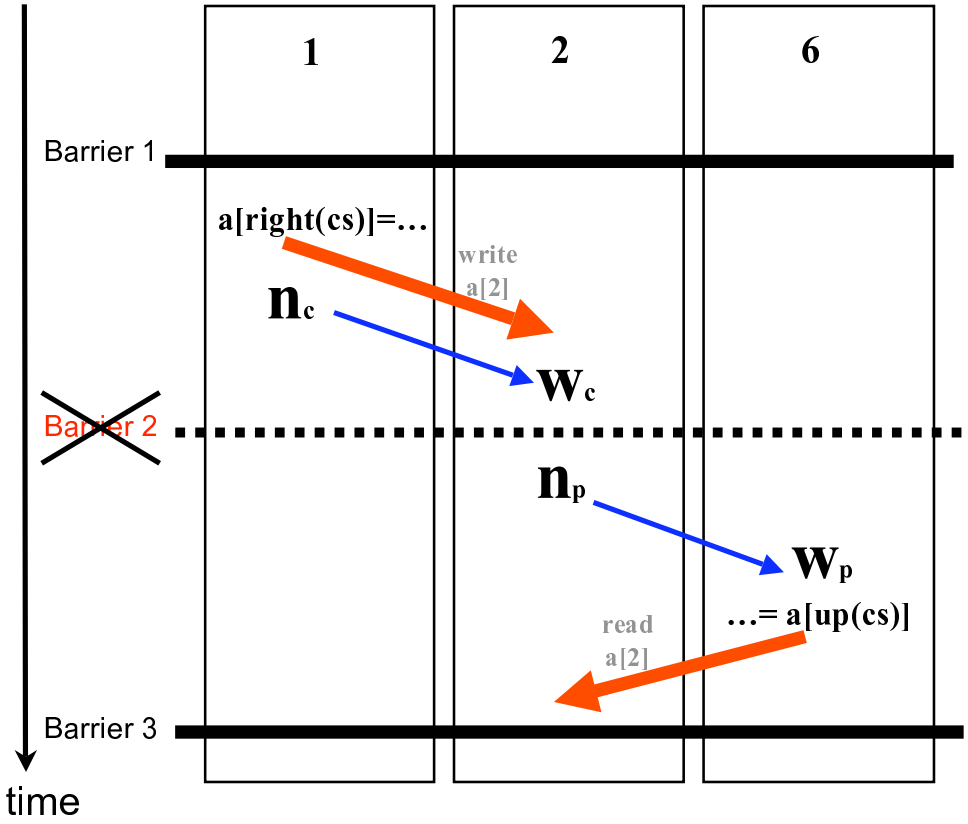


Figure 7.9 : Time diagram for communication for images 1 and 6 accessing the same co-array memory `a[2]`.

the barrier on line 2 can be analyzed and synchronized with point-to-point synchronization, then it would be profitable to remove this barrier, replacing it with an equivalent set of

point-to-point synchronization that enforces all such dependencies. If this cannot be done, the barrier on line 2 must be left intact, and a permission pair should not be generated for the PUT on line 3 since the barrier on line 2 already synchronizes it properly. Similar reasoning applies to the barrier on line 6. This intuition lays a foundation for our reducibility analysis and the following observation.

We say that a barrier b reaches a communication event e if there is a barrier-free path in the control flow graph (CFG) from b to e. We say that an event e reaches b if there is a barrier-free path in the CFG from e to b.

Observation 7.1 *A textual co-space barrier b can safely be removed from the code and replaced by an equivalent set of point-to-point permission & completion synchronization pairs that preserve correctness iff (1) each communication event that may reach b in any execution can be analyzed and is synchronized with one or more completion pairs, and (2) each communication event that b may reach in any execution can be analyzed and is synchronized with one or more permission pairs.*

7.2.2 Correctness of SSR for analyzable non-group-executable PUTs/GETs

So far, we have considered only analyzable group-executable PUT/GET. Using point-to-point synchronization to synchronize an analyzable non-group-executable PUT/GET is slightly different. An example of such an event is shown in Figure 6.18. Intuitively, we can place a permission pair and a completion pair around the `if` statement, as shown in Figure 7.10. This preserves correctness, but may introduce unnecessary point-to-point synchronization for images that do not actually perform the PUT on line 4. In experiments (see Section 7.8.2), we found that codes with such extra point-to-point synchronization are less synchronous and faster than their barrier-based counterparts because the critical path of unidirectional notify messages is shorter than that of a barrier.

For a non-group-executable PUT/GET, permission & completion pairs cannot be placed

```
   ...
   ! former barrier
1  call notify(CS_Neighbor(cs,axis,+1))   ! permission notify
2  call wait(CS_Neighbor(cs,axis,-1))     ! permission wait
3  if (give_ex(axis, level)) then  ! non-single-valued guard
4    ... ! pack data into buffM(1:buff_len,1)
5    buffM(1:buff_len,2)[CS_Neighbor(cs,axis,-1)] = buffM(1:buff_len,1)
6  endif
7  call notify(CS_Neighbor(cs,axis,-1))   ! completion notify
8  call wait(CS_Neighbor(cs,axis,+1))     ! completion wait
   ...
   ! former barrier
```

Figure 7.10 : Non-single-valued guard in NAS MG extrapolation subroutine synchronized with permission & completion pairs instead of barriers.

directly around the event as for analyzable group-executable PUT/GET since notify and wait of each completion & permission pair must be matched and not all images would necessarily perform the synchronization (see Section 6.4.2). Instead, one could place a permission pair earlier in execution, in a group-executable CFG node n that executes if the event executes, provided that it is possible to compute the target image of communication in n. Such placement would enable the compiler to find the origin(s) of communication. Thus, the best node to place a permission pair is the *closest*[1] group-executable CFG dominator d of the event, provided the arguments of the `CS_Neighbor` function can be computed in d; note that if the arguments are available in d, they are single-valued in d. Otherwise, SSR cannot optimize synchronization for a non-group-executable event. If a permission pair can be placed, the corresponding completion pair can be placed in the closest group-executable CFG postdominator p of the event node, later in execution than the event. Note that d and p are control equivalent for the structured control flow that we support and the shape of our CFG (see Section 7.3).

The described synchronization with permission & completion pairs instead of barriers ensures correct synchronization because the placements of the permission & completion pairs are no further from the communication than the textual barriers they replace; thus, the permission & completion pairs provide equivalent synchronization. Let us consider the

[1]To execute as little unnecessary point-to-point synchronization as possible.

case of the permission pair. Let e be the non-group-executable event CFG node, $cGEdom$ be the closest group-executable dominator of e. With the CFG restricted to structured control flow, any CFG node b that belongs to a path from $cGEdom$ to e is non-group-executable according the the inference algorithm in Section 6.3. Therefore, b cannot contain a textual co-space barrier; if it does, it is a program error, a possible deadlock, and SSR is not applied. Similar reasoning holds for the completion pair.

7.2.3 Hiding exposed synchronization latency

It is common for scientific codes to perform some local work between communication events. Figure 7.11 (a) shows an example and denotes local work as `work1` and `work2`. Figure 7.11 (b) shows the synchronization using permission & completion pairs inserted right around the PUT `a[right(cs)]`. Placing the permission pair right before an event exposes the latency to deliver the permission notify message because, assuming that execution of all images is approximately balanced, every image issues a permission notify and immediately blocks in a permission wait until the corresponding permission notify message arrives from a remote image. Similarly, placing the completion pair right after an event exposes both the latency to transfer data and the latency to deliver the completion notify message.

If legal, it is profitable to move the permission notify earlier in the execution and the completion wait later in the execution, as shown in Figure 7.11 (c). This overlaps the permission notify latency with local computation `work1`, and the PUT and completion notify latencies with `work2`. We limit the movement of a permission notify by the availability of arguments (inputs) for `CS_Neighbor` function to compute the origin(s) of communication[2] and by the upward barrier(s)[3]. However, we limit the movement of a comple-

[2]In the presence of control flow, we accumulate the execution guard along the way and do not move a permission notify and a completion wait outside of loops. We postpone this discussion until Section 7.6.

[3]Because the synchronization must happen somewhere between execution of the barrier and the event. It may be possible to move a permission notify past the upward barrier(s), preserving inter-image data depen-

```
...
call barrier()

work1

a[right(cs)]=...

work2

call barrier()
...
```

(a) barrier-based synchronization

```
...
! former barrier

work1
call notify(left(cs))
call wait(right(cs))
a[right(cs)] = ...
call notify(right(cs))
call wait(left(cs))
work2

! former barrier
...
```

(b) synchronization with permission & completion pairs

```
...
! former barrier
call notify(left(cs))
work1

call wait(right(cs))
a[right(cs)] = ...
call notify(right(cs))

work2
call wait(left(cs))
! former barrier
...
```

(c) pairwise synchronization with latency hiding

Figure 7.11 : Communication to the right for a 4×2 Cartesian co-space with periodic boundaries.

tion wait only by the downward barrier(s) because the origin(s) of communication can be computed at the event's permission wait point for both group-executable and non-group-executable events, stored in compiler-generated temporaries, and used for synchronization at the downward barrier(s). An invariant that must be maintained when moving a permission notify and a completion wait is (1) to execute a *single* permission notify per permission wait execution and (2) to execute a *single* completion wait per completion notify execution. This essentially matches permission notify and permission wait, and completion notify and completion wait. In Section 7.6, we show a formal algorithm how to move a permission notify and a completion wait maintaining the invariant.

7.3 Overview of procedure-scope SSR

We present an SSR algorithm that operates within a procedure scope. SSR can analyze codes that use textual barriers of co-space `cs` for synchronization and `single(cs,exp)` to specify co-space single-valued expressions. The co-space `cs` must be a single-valued

dencies; however, codes we have studied do not present opportunities for this.

procedure invariant. SSR supports codes with structured control flow in the form of `DO` loops and `IF-THEN-ELSE` or `IF-THEN` statements. The class of programs expressible with this set of constructs is broad enough to include all scientific CAF codes that we encountered. However, it is necessary to extend SSR's scope beyond a single procedure to use SSR to optimize real scientific codes. In this dissertation, we use compiler hints to achieve this effect until interprocedural analysis is available.

SSR has four major phases. We summarize them here; the rest of the chapter presents them in detail.

- **Preliminary analysis** checks that SSR can be applied, prepares the CFG for the following stages, identifies DO loop regions, and collects various information about the CFG (see Section 7.4). Finally, this stage performs the inference of co-space single values and group-executable statements as described in Chapter 6.

- **Reducibility analysis** detects analyzable group-executable and non-group-executable PUTs/GETs and runs a fixed point iterative algorithm to find all textual co-space barriers that can be reduced as well as to determine what notify/wait synchronization is required to preserve program correctness if the barriers are to be eliminated.

- **Optimization of notify/wait synchronization** overlaps the latency of permission and completion notifies with local computation and eliminates redundant notify/wait synchronization.

- **Code generation** phase instantiates notify/wait synchronization and removes barriers that are no longer necessary. In addition, it detects PUTs that can be made non-blocking and converts them into non-blocking form.

In the next section, we begin with an overview of concepts used in SSR.

7.3.1 Notation and terminology

- A *communication event* is either a PUT or a GET. An *event placeholder* is an "empty" event used for analysis to simplify flow equations. In the following, an *event* refers to either a communication event or an event placeholder; it is represented by an $Event$ data structure shown in Figure 7.19. Sometimes, we use *event* to refer to a communication event, when the difference is clear from the context.

- A *synchronization fence* limits movement of notify/wait synchronization and helps to simplify analysis flow equations. Its properties resemble that of a *barrier*; however, a synchronization fence is never present in the code. We might replace a synchronization fence with a barrier when it is either necessary or profitable. In the following, a *fence* refers to either a barrier or a synchronization fence.

- We say that a fence f reaches an event e if there is a fence-free path in the CFG from f to e.

- We say that an event e reaches a fence f if there is a fence-free path in the CFG from e to f.

- For a fence f, the set $eventsBeforeFence(f)$ is the set of all events that reach f; the set $eventsAfterFence(f)$ is the set of all events that f reaches.

- For an event e, the set $fencesBeforeEvent(e)$ is the set of all fences that reach e; the set $fencesAfterEvent(e)$ is the set of all fences that e reaches.

- A barrier b is *reducible* if it can be safely removed from the code by replacing it with point-to-point permission & completion notify/wait pairs, coordinated with guards, to preserve data access ordering.

- A communication event e is *upwardly synchronizable* if it is synchronized with a permission pair instead of barriers that reach e.

- A communication event e is *downwardly synchronizable* if it is synchronized with a completion pair instead of barriers that e reaches.
- For a CFG node n, $idom(n)$ denotes the immediate dominator of n; $ipostdom(n)$ denotes the immediate postdominator of n.
- A CFG node containing a fence is called *fence node*. A CFG node containing a barrier is called *barrier node*.
- $FenceIDF$ stands for iterated dominance frontier [41] for fence CFG nodes. This is the set of CFG nodes in which each member belongs to the iterated dominance frontier of some fence node. Each node in $FenceIDF$, called a fence merge point, is reachable by at least two different fences. We use fence merge points to recur along different CFG paths while moving a permission notify upward in the CFG.
- $FenceIRDF$ stands for iterated reverse dominance frontier for fence CFG nodes. This is the set of CFG nodes in which each member belongs to the iterated reverse dominance frontier of some fence node. Each node in $FenceIRDF$, called a fence split point, reaches at least two different fences. We use fence split points to recur along different CFG paths while moving a completion wait downward in the CFG.

Synchronization primitives

n_p and w_p denote a permission notify(s) and a permission wait, respectively. n_c and w_c denote a completion notify and a completion wait(s), respectively.

The primitives used for n_p/w_p and n_c/w_c are *not* CAF's `notify` and `wait`. CAF's primitives may be used by the programmer and are not composable. We use primitives `notify(cs,q)` and `wait(cs,r)` that are similar to CAF's, but they are "private" to co-space `cs` and appear only in compiler-generated code. Moreover, while w_p and n_c always have only one target image — the target image of communication, n_p and w_c actually denote sets of notifies/waits because there can be several origin images (*e.g.*, for a graph co-space). Section 7.7 has more detailed discussion.

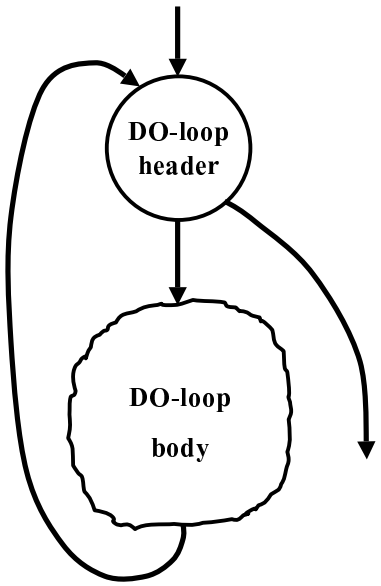


Figure 7.12 : Preconditioned `DO` loop.

CFG shape

To simplify insertion and movement of notify/wait, our CFG follows the following guidelines:

1. Each CFG node contains at most one program statement and might contain SSA Φ-nodes block at the node's entry and/or SSA side effect block if the statement is a procedure call. A side effect block contains definitions of SSA names that might be modified by the corresponding procedure call. It is inserted after the procedure call. We would use Φ-node and side effect definitions to limit the upward movement of computations (the target of communication) in the CFG.

2. For each `IF-THEN-ELSE` or `IF-THEN`, there is a no-op (comment) statement inserted right after the `ENDIF` in the source code, so that there are no critical edges[4]

[4]A critical edge is a CFG edge whose source has multiple successors and whose destination has multiple predecessors.

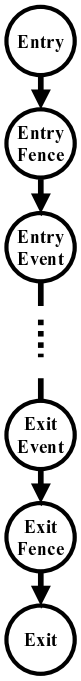


Figure 7.13 : $EntryFence$, $EntryEvent$, $ExitEvent$, and $ExitFence$.

for nested `IF-THEN-ELSE`/`IF-THEN` statements. In addition, all `IF-THEN` are converted into `IF-THEN-ELSE`.

3. Each `DO` loop is preconditioned and has the shape shown in Figure 7.12. Each loop has only one entry n node and only one exit node, which is the same as n. The case of unstructured control flow and exit branches from the loop is discussed in Section 7.9.

7.3.2 Synchronization and event placeholders

$EntryFence$ is a synchronization fence at procedure entry. $ExitFence$ is a synchronization fence at procedure exit. $EntryEvent$ is an event placeholder at procedure entry. $ExitEvent$ is an event placeholder at procedure exit. Figure 7.13 shows their placement in the CFG. These four placeholders are used to control barrier reducibility and event synchronizability conditions at procedure entry and exit, as discussed in Section 7.5.

We augment each `DO` loop that may execute a barrier with $Preloop$, $Postloop$,

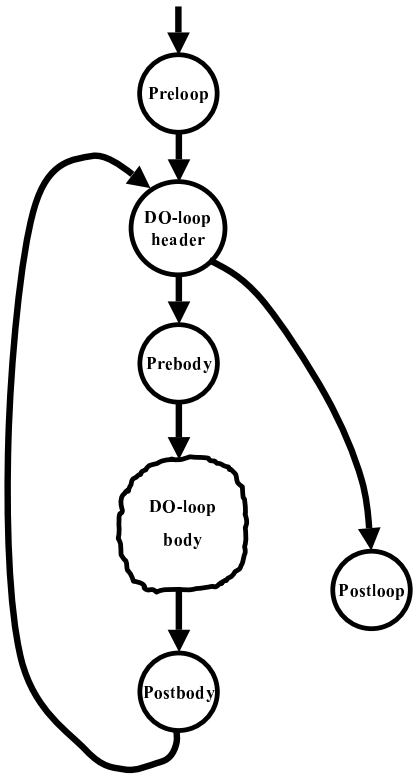


Figure 7.14 : Preconditioned DO loop with $Preloop$, $Postloop$, $Prebody$, and $Postbody$ placeholders.

$Prebody$, and $Postbody$ placeholders, as shown in Figure 7.14. Each placeholder contains two nodes: one with a synchronization fence, the other with an event placeholder. $Preloop$ contains $PreloopEvent$, followed by $PreloopFence$. $Postloop$ contains $PostloopFence$, followed by $PostloopEvent$. $Prebody$ contains $PrebodyFence$, followed by $PrebodyEvent$. $Postbody$ contains $PostbodyEvent$, followed by $PostbodyFence$.

7.3.3 Pseudocode data structures

Figures 7.15, 7.16, 7.17, 7.18, and 7.19 show the data structures used in pseudocode in the following sections.

Figure 7.15 shows the $Node$ data structure that represent a CFG node b. The $state$

struct $Node$
- $state$ // GE or NGE
- Φ-$nodes$ // Φ-nodes of the node
- $stmt$ // program AST statement, only one

Figure 7.15 : CFG node structure.

struct $Fence$
- $reducible$ // reducible or non-reducible
- $node$ // CFG node
- $eventsBeforeFence$ // the set of events $Event$ that reach the fence
- $eventsAfterFence$ // the set of events $Event$ reachable by the fence

Figure 7.16 : Fence structure for a barrier or a synchronization fence.

struct $Region$
- $header$ // DO header node
- $preloopFence$ // $Preloop$ fence of type $Fence$
- $postloopFence$ // $Postloop$ fence of type $Fence$
- $prebodyFence$ // $Prebody$ fence of type $Fence$
- $postbodyFence$ // $Postbody$ fence of type $Fence$
- $preloopEvent$ // $Preloop$ event placeholder of type $Event$
- $postloopEvent$ // $Postloop$ event placeholder of type $Event$
- $prebodyEvent$ // $Prebody$ event placeholder of type $Event$
- $postbodyEvent$ // $Postbody$ event placeholder of type $Event$

Figure 7.17 : DO loop region structure.

struct $Place$
- $node$ // CFG node
- $whereToInsert$ // insert notify/wait before or after $node.stmt$
- $guard_{n_p}$ // the guard expression for n_p

Figure 7.18 : Place structure for a notify or a wait.

field specifies whether b is group-executable or non-group-executable, as determined by the inference analysis in Section 6.3. The Φ-$nodes$ field represents the SSA form Φ-$nodes$ of b. The $stmt$ field represents b's Open64/SL Whirl abstract syntax tree (AST) statement.

Figure 7.16 shows the $Fence$ data structure that represents a fence f, which

struct $Event$

$upwardlySynchronizable$	// synchronizable with a n_p/w_p pair, or not
$downwardlySynchronizable$	// synchronizable with a n_c/w_c pair, or not
$node$	// CFG node
$fencesBeforeEvent$	// the set of fences $Fence$ that reach the event
$fencesAfterEvent$	// the set of fences $Fence$ reachable by the event
$image$	// the AST expression of the event's target image
$n_pPlaces$	// set of $Place$ for permission notifies
w_pPlace	// the $Place$ for permission wait
n_cPlace	// the $Place$ for completion notify
$w_cPlaces$	// set of $Place$ for completion waits

Figure 7.19 : Event structure for a PUT/GET or an event placeholder.

can be either a barrier or a synchronization fence. The $reducible$ field determines the reducibility state of the fence, which can be reducible or non-reducible. The $node$ field specifies the CFG node of f. The $eventsBeforeFence$ field represents the set $eventsBeforeFence(f)$. The $eventsAfterFence$ field represents the set $eventsAfterFence(f)$.

Figure 7.17 shows the $Region$ data structure that represents the CFG region for a `DO` loop that may execute a barrier; no $Region$ structure is associated with a `DO` loop that does not execute a barrier. The $header$ field is the `DO` loop entry (and exit) CFG node. The other fields represent helper communication fences and event placeholders for the reducibility analysis and notify/wait optimization.

Figure 7.18 shows the $Place$ data structure for a placement of a notify or a wait in the CFG. The $node$ field is the CFG node in which the notify or wait resides and will be inserted in the code generation stage. The $whereToInsert$ field can be either $beforeStmt$ or $afterStmt$ specifying whether to insert the notify/wait before $node.stmt$ or after $node.stmt$, respectively. The $guard_{n_p}$ field represents the AST expression for the permission notify guard necessary for code generation to match the permission notify and permission wait.

Figure 7.19 shows the $Event$ data structure that represents an event e, which can be a

PUT/GET or an event placeholder. The $upwardlySynchronizable$ field specifies whether e should be synchronized with a permission pair. The $downwardlySynchronizable$ field specifies whether e should be synchronized with a completion pair. The $node$ field is e's CFG node. The fields $fencesBeforeEvent$ and $fencesAfterEvent$ represent sets $fencesBeforeEvent(e)$ and $fencesAfterEvent(e)$, respectively. For a PUT/GET, the $image$ field is e's target image AST expression; for an analyzable PUT/GET, it is a call to the co-space `CS_Neighbor` function. The $image$ field is undefined for event placeholders. If e is upwardly synchronizable, the $n_pPlaces$ field is the set of all CFG places for e's permission notifies, and the w_pPlace field is the CFG place for e's permission wait. If e is downwardly synchronizable, the n_cPlace field is the CFG place for e's completion notify, and the $w_cPlaces$ field is the set of all CFG places for e's permission waits.

7.3.4 Hints to increase SSR scope beyond the procedure level

The SSR algorithm presented in this dissertation works for the procedure scope; however, optimization of real programs often requires a scope beyond a single procedure. We describe SSR hints here because they are incorporated into the following stages of SSR.

The scope of SSR can be increased in three ways: devising an interprocedural analysis, inlining procedures, and using hints. While a fully automatic solution is preferable, `cafc` does not yet have infrastructure for any interprocedural analysis or procedure inlining, including interprocedural analysis or procedure inlining to support SSR. This leaves us with two options. First, the programmer can inline procedures manually. Second, the programmer can use directives to provide extra information to `cafc`. We believe that the second choice imposes less burden on programmers, and enables one to reap the benefits of interprocedural SSR today. Moreover, even in the presence of interprocedural analysis in the future, its capabilities would be limited by separate compilation and libraries, unless link-time analysis and code generation are used. Directives can be useful to improve optimization in the absence of more sophisticated link-time optimization. We discuss the nature of a possible interprocedural analysis in Section 7.9. We introduce two `cafc` directives to

convey information about remote co-array accesses and inter-image data dependencies beyond the scope of one procedure.

The `local` attribute for a procedure `foo` conveys the fact that neither `foo` nor any procedure called transitively from `foo` performs any PUT/GET.

The `synch_context(cs)` hint specifies a "synchronization clean point" for all co-array accesses of co-space `cs` at which all co-space inter-image data dependencies are known to be enforced by either a barrier or point-to-point synchronization. We are particularly interested in `synch_context(cs)` hints at procedure entry or exit, because this enables SSR to optimize synchronization between consecutive procedure invocations. Let's assume that the invocations are I_1 followed by I_2; I_1 executes procedure S_1, and I_2 executes procedure S_2. Note that S_1 and S_2 can be the same procedure.

At S_1's exit, `synch_context(cs)` instructs SSR to complete all S_1's PUTs/GETs[5] for the co-space `cs` that reach S_1's exit. If possible, it is preferable to complete local-scope communication by using point-to-point completion pairs; otherwise, SSR must synchronize using a co-space barrier at S_1's exit. The hint indicates that it is safe to synchronize using completion pairs because the following scope, *e.g.*, I_2 invocation of S_2, synchronizes S_2's communication events reachable from S_2's entry with point-to-point permission pairs, if possible, or, otherwise, with a barrier. At S_2's entry, `synch_context(cs)` indicates that all prior PUTs/GETs for co-space `cs` have been completed by either a barrier or point-to-point completion pairs. However, the hint also requires us to synchronize all S_2's scope communication events reachable from S_2's entry by using point-to-point permission pairs, if possible; otherwise, SSR must synchronize using a barrier at S_2's entry.

`synch_context(cs)` hints bear a resemblance to a split synchronization fence: one part of the fence is at S_1's exit for the invocation I_1, the other is at S_2's entry for the invocation I_2, where S_1 and S_2 can be the same procedure. `synch_context(cs)` hints limit the movement of permission & completion pairs, and they are the points where data dependencies are enforced. However, `synch_context(cs)` is stronger than a synchronization

[5]For each image, local co-array accesses are completed because of the image's program execution.

fence because it *must* become a barrier if it is not possible to use point-to-point synchronization to synchronize local scope PUTs/GETs; a synchronization fence may become a barrier only if profitable (see Section 7.5).

Our implementation supports `synch_context(cs)` inserted at procedure entry or exit. At procedure entry, SSR might insert a barrier if a `synch_context(cs)` hint is present, but SSR cannot upwardly synchronize events reachable from the procedure entry by permission pairs. At procedure exit, SSR might insert a barrier if a `synch_context(cs)` hint is present, but SSR cannot downwardly synchronize events reaching the exit of the procedure by completion pairs. If either of these barriers is instantiated, SSR issues a warning message, because the barriers might increase the amount of synchronization in the code. In scientific codes available to us, SSR never had to insert such a barrier.

7.4 Preliminary analysis

First, we preprocess the program abstract syntax tree and insert an empty statement after each `ENDIF`. This avoids critical CFG edges for nested `IF-THEN-ELSE`/`IF-THEN` constructs. We convert all `IF-THEN` statements into `IF-THEN-ELSE` statements to make the CFG more uniform and to apply the same set of rules for `IF-THEN-ELSE` and `IF-THEN` statements when moving notify and wait during the optimization phase. Also, our SSR CFG has at most one statement in each CFG node.

Second, we verify that SSR can be applied by performing the following steps. If any condition does not hold, we do not apply SSR.

1. **Verifying control flow.** We verify that the CFG contains only `IF-THEN-ELSE` and `IF-THEN` control flow statements and `DO` loops.

2. **Detecting SSR co-space.** We verify whether all textual co-space barriers, co-space coercion operators, and `synch_context(cs)` hints use the same co-space variable `cs`. `cs` must not be redefined in the procedure.

Third, to prepare the CFG for SSR analysis, we normalize its form by applying the following transformations.

1. **Insulating loops containing barriers from communication outside the loop.**

   Each `DO` loop is represented in the CFG as a pre-conditioned loop with exactly two successors and two predecessors, as shown in Figure 7.12. SSR treats a `DO` loop that may execute a barrier as an independent CFG region.

   To detect `DO` loops that may execute a barrier, we declare a *barrier helper variable* (`BHV`) and facilitate the SSA form. We insert a helper statement: `BHV=BHV+1` right after each barrier statement and build the SSA. Each `DO` loop whose header node has a Φ-node for `BHV` may execute a barrier and represents a `DO` loop CFG region.

   We insert four placeholders for $Preloop$, $Postloop$, $Prebody$, and $Postbody$ synchronization fences and event placeholders, as shown in Figure 7.14. The synchronization fences separate the communication and synchronization inside the loop from the outside. They also do not allow outside synchronization to move past the loop statement. The event placeholders are used to control how barrier non-reducibility propagates into and out of the loop.

2. **Insulating procedures from communication outside procedures.**

   Immediately after the CFG entry node, we insert the $EntryFence$ synchronization fence placeholder followed by the $EntryEvent$ event placeholder. Immediately before the CFG exit node, we insert the $ExitEvent$ event placeholder, followed by the $ExitFence$ synchronization fence placeholder. This is shown in Figure 7.13. The fences are used to separate procedure communication/synchronization from the outside communication. For each call site of a procedure without `local` attribute, we insert two event placeholders: one before the call site, the other after the call site. The fences and events are used to analyze barrier reducibility in Section 7.5.

Fourth, we collect analysis data to support further analysis.

1. We construct the CFG and SSA form.
2. We compute the dominator tree (DT) and postdominator tree (PDT).
3. We perform the inference of group-executable (GE) statements and co-space single values (SV), as described in Section 6.3.
4. We associate a $Node$ structure with each CFG node. If the statement of the node is group-executable, we initialize $Node.state$ to GE; otherwise, it is NGE.
5. We associate a $Fence$ structure with each barrier and each synchronization fence.
6. We build iterated dominance frontier and iterated reverse dominance frontier for fence nodes; we denote them as $FenceIDF$ and $FenceIRDF$, respectively.
7. We associate a $Region$ structure with each `DO` loop region. Note that a `DO` loop region can reuse DT and PDT of the entire CFG that are restricted by the $Prebody$ and $Postbody$ fences.
8. We associate an $Event$ structure with each communication event (PUT/GET) or an event placeholder.

7.5 Reducibility analysis

The goal of reducibility analysis is to identify barriers that can be *reduced*; *i.e.*, be removed from the code and replaced by a set of permission & completion synchronizations. The analysis should not introduce more synchronization than the original code has. Thus, it also determines whether a communication event e is *upwardly synchronizable* and *downwardly synchronizable* with point-to-point synchronization. The synchronizability property of each end is independent from that of the other end. If e is upwardly synchronizable, it can safely be upwardly synchronized with a permission pair instead of barriers reaching e. If e is downwardly synchronizable, it can safely be downwardly synchronized with a

completion pair instead of barriers that e reaches. In addition, the analysis also determines the placement of the permission wait and the completion notify for each event.

Reducible barriers and synchronizable events can be detected by an optimistic monotonic fixed-point iterative algorithm that propagates properties from fences to events and from events to fences. The analysis initially assumes that all barriers are reducible. Intuitively, if an event e reaches barrier b and e is not downwardly synchronizable, b must be non-reducible because b must be used to synchronize e to preserve a potential inter-image data dependence emanating from e; thus, b cannot be removed from the code. Similarly, if an event e reaches barrier b and b is non-reducible, b is already sufficient to synchronize e, and a completion pair should not be generated for e not to introduce unnecessary synchronization; thus, e should not be downwardly synchronizable. Similar reasoning applies for the situation where a barrier b reaches an event e. We give more details while describing the propagation step below.

7.5.1 Initialize flow equations

The initialization step consists of two parts. The first one determines whether each communication event is initially synchronizable as well as placement of a permission wait and a completion notify for each PUT/GET. The second one deals with various flow conditions.

Initialize communication events

Figure 7.20 shows pseudocode for initializing the state of each communication event e. e is non-analyzable if SSR cannot qualify it as an analyzable group-executable or non-group-executable PUT/GET (see Patterns 6.1 and 6.2 in Chapter 6). For example, e is non-analyzable if its target image is not expressed via a co-space `CS_Neighbor` function, or if the `CS_Neighbor` function does not have single-valued arguments. If e is non-analyzable, SSR cannot synchronize it with point-to-point synchronization. Thus, e is not upwardly or downwardly synchronizable. If e is analyzable, we determine the placement for e's w_p and n_c and whether e is synchronizable, as shown in Figure 7.21, unless e is a special event

procedure $initializeCommunicationEvents$
  for each communication event e
    if e is non-analyzable (not an analyzable GE or NGE PUT or GET)
      $e.upwardlySynchronizable \leftarrow false$
      $e.downwardlySynchronizable \leftarrow false$
    else // analyzable GE or NGE PUT or GET
      if $\exists$ DO loop region R such that
        there is a fence-free path in the CFG from the $R.prebody$ fence to e and
        there is a fence-free path from e to the $R.postbody$ fence
          // heuristic: avoid unnecessary synchronization in a loop executing a barrier
          $e.upwardlySynchronizable \leftarrow false$
          $e.downwardlySynchronizable \leftarrow false$
      else
          // determine the initial placement of w_p and n_c
          call $determineInitialW_pN_cPlacement(e)$ // see Figure 7.21

Figure 7.20 : Detecting synchronizable PUT/GET events.

in a DO loop. If e executes in a DO loop containing a barrier (the DO loop has a region R associated with it) and e may execute without being synchronized with a barrier on some iteration of the loop, we do not optimize e, not to introduce more synchronization into the program. This is demonstrated by the following example:

```
do i = 1, 101
  if (i == 101) then
    call barrier(cs)
  else
    a(i)[CS_Neighbor(cs, i)] = b(i)
  endif
enddo
```

If the barrier is reduced, the permission & completion pairs execute on 100 iterations of the loop; however, the original version does not execute any synchronization except the barrier on the last iteration. The original version would probably be faster than the reduced one.

Figure 7.21 determines the placement of a permission wait and a completion notify for an analyzable group-executable/non-group-executable event. If the event e node is group-executable, which corresponds to the first analyzable Pattern 6.1, a permission wait and a

```
procedure determineInitialW_pN_cPlacement(Event e)
    // e is an analyzable GE or NGE PUT or GET
    if e.node.state = GE // an analyzable GE PUT or GET
        // place w_p and n_c at the node of the event
        e.w_pPlace ← new Place(e.node, beforeStmt)
        e.n_cPlace ← new Place(e.node, afterStmt)
        e.upwardlySynchronizable ← true
        e.downwardlySynchronizable ← true
        return
    else // an analyzable NGE PUT or GET
        inputs ← SSA names referenced by the e.image expression
        // place w_p: assume that the event cannot be analyzed by SSR
        e.w_pPlace ← new Place(e.node, beforeStmt)
        e.upwardlySynchronizable ← false
        e.downwardlySynchronizable ← false
        while e.w_pPlace.node.state = NGE
            if ∃ SSA name n, n ∈ inputs, that is defined by either
              e.w_pPlace.node's statement or Φ-nodes
                return
            // move to the immediate CFG dominator
            e.w_pPlace.node ← idom(e.w_pPlace.node, beforeStmt)
        // found the closets GE dominator with available inputs to compute target image
        if e.w_pPlace.node is not a DO-loop entry node
            // the event can be analyzed (e.w_pPlace.node contains IF statement)
            e.upwardlySynchronizable ← true
            e.downwardlySynchronizable ← true
            // place n_c
            e.n_cPlace ← new Place(ipostdom(e.w_pPlace.node, afterStmt))
```

Figure 7.21 : Determining synchronizable PUT/GET and placement for w_p and n_c.

completion notify can be placed right around the event in the same node, as discussed in Section 7.2; e is synchronizable. If e is an analyzable non-group-executable PUT/GET, we try to find the closest group-executable dominator d of $e.node$ such that the SSA names (inputs) referenced by the target image expression $e.image$ are available in d, as discussed in Section 7.2. If d is found, e is synchronizable, and d is the node to place a permission wait; the immediate postdominator of d is then the place for a completion notify. If d

cannot be found, SSR cannot analyze e and makes e non-synchronizable. Note that our implementation of SSR neither vectorizes permission & completion pairs nor hoists them outside of a `DO` loop that does not contain a barrier because we have not encountered opportunities in the codes that we have studied. We discuss this possibilities in Section 7.9.

Initialize reducibility state

Figure 7.22 shows pseudocode for the rest of initialization. Each fence is optimistically assumed to be reducible; each event placeholder is optimistically assumed to be synchronizable.

To keep the flow equations uniform, we introduce two event placeholders, represented by the $Event$ structure, with the same reducibility properties as a communication event. The $EntryEvent$ placeholder is inserted in the CFG right after the $EntryFence$ synchronization fence at procedure entry. The $ExitEvent$ is inserted right before the $ExitFence$ synchronization fence at procedure exit, as shown in Figure 7.13.

If a `synch_context(cs)` hint at procedure entry is present, the $EntryFence$ is initialized to be reducible, and the $EntryEvent$ is initialized to be synchronizable. Otherwise, SSR must be conservative; *i.e.*, it initializes them to be non-reducible and non-synchronizable, respectively. In the propagation step, a non-reducible $EntryFence$ will make all events reachable from the procedure entry upwardly non-synchronizable, and a downwardly non-synchronizable $EntryEvent$ will make all fences reachable from the procedure entry non-reducible; this is exactly what we want when a `synch_context(cs)` hint at procedure entry is not present, which means that a downwardly non-synchronizable communication event may reach the procedure invocation and, thus, the procedure entry.

If a `synch_context(cs)` hint at procedure exit is present, the $ExitFence$ is initialized to be reducible, and the $ExitEvent$ is initialized to be synchronizable; otherwise, SSR must be conservative and similar reasoning is applicable as for the procedure entry.

For each procedure call that may execute a PUT/GET (no `local` attribute), we insert two event placeholders around the call site. SSR initializes them to be non-synchronizable

```
procedure initializeReducibilityState
    // optimistically initialize all fences to be reducible
    for each fence f
        f.reducible ← true

    // optimistically initialize all event placeholders to be synchronizable
    for each event placeholder e
        e.upwardlySynchronizable ← true
        e.downwardlySynchronizable ← true

   // procedure entry
    if there is no synch_context(cs) hint at procedure entry
        EntryFence.reducible ← false
        EntryEvent.upwardlySynchronizable ← false
        EntryEvent.downwardlySynchronizable ← false

    // procedure exit
    if there is no synch_context(cs) hint at procedure exit
        ExitFence.reducible ← false
        ExitEvent.upwardlySynchronizable ← false
        ExitEvent.downwardlySynchronizable ← false

    // procedure calls containing possible PUT/GET
    for each call site of a procedure without the local attribute
        // let e1 and e2 be event placeholders before and after the call, respectively
        e1.upwardlySynchronizable ← false
        e1.downwardlySynchronizable ← false
        e2.upwardlySynchronizable ← false
        e2.downwardlySynchronizable ← false
```

Figure 7.22 : Initializing reducibility state.

because it cannot analyze events outside procedure scope and must be conservative. SSR assumes that there might be non-synchronized communication events reachable from the point right before the call site and reaching the point right after the call site; these possible unsynchronized conflicting communication events are represented by the two non-synchronizable event placeholders.

```
procedure buildSets
    // initialization
    for each fence f
        f.eventsBeforeFence ← ∅
        f.eventsAfterFence ← ∅
    for each event e
        e.fencesBeforeEvent ← ∅
        e.fencesAfterEvent ← ∅
    for each event e
        call moveUpward(e, e.node) // see Figure 7.24
        call moveDownward(e, e.node) // see Figure 7.24
```

Figure 7.23 : Building $fencesBeforeEvent$, $fencesAfterEvent$, $eventsBeforeFence$, and $eventsAfterFence$ sets.

7.5.2 Detect reducible barriers and synchronizable communication events

To propagate non-reducibility of fences and non-synchronizability of events, we first construct reachability sets. For each event e, we build $fencesBeforeEvent(e)$ and $fencesAfterEvent(e)$ sets. For each fence f, we build $eventsBeforeFence(f)$ and $eventsAfterFence(f)$ sets. The pseudocode for initialization and recursive traversal of the CFG is shown in Figures 7.23 and 7.24.

We use synchronization fences to simplify flow equations. Synchronization fences are *not* real barriers. They are present to give more control over how the analysis treats `DO` loops, procedure calls, and procedure entry/exit. However, a synchronization fence may become a real barrier when it is profitable or necessary to satisfy the assumptions of the `synch_context(cs)` hints, as discussed below.

Figure 7.25 shows the propagation step where information flows from events to fences and from fences to events. Figure 7.26 shows how the results of the propagation step are used to optimize `DO` loops and to handle `synch_context(cs)` hints at procedure entry/exit.

The first three steps, shown in Figure 7.25, iteratively, transitively propagate fence non-

```
procedure moveUpward(Event e, Node n)
    if n contains a fence f
        f.eventsAfterFence ← f.eventsAfterFence ∪ {e}
        e.fencesBeforeEvent ← e.fencesBeforeEvent ∪ {f}
        return
    if n ∈ FenceIDF // a merge point for fences
        for each node p ∈ pred(n) // for each CFG predecessor (no back edges)
            call moveUpward(e, p)
    else
        call moveUpward(e, idom(n)) // move to the immediate dominator

procedure moveDownward(Event e, Node n)
    if n contains a fence f
        f.eventsBeforeFence ← f.eventsBeforeFence ∪ {e}
        e.fencesAfterEvent ← e.fencesAfterEvent ∪ {f}
        return
    if n ∈ FenceIRDF // a split point for fences
        for each node s ∈ succ(n) // for each CFG successor (no back edges)
            call moveDownward(e, s)
    else
        call moveDownward(e, ipostdom(n)) // move to the immediate postdominator
```

Figure 7.24 : Recursive procedures to build reachability sets.

reducibility and event upward/downward non-synchronizability. Step I captures propagation of the fact that if an event cannot be synchronized safely with point-to-point synchronization, barriers used to synchronize the event cannot be removed. Any upwardly non-synchronizable event e that is reachable by a fence f makes f non-reducible since a barrier must be used to enforce inter-image data dependencies (a permission pair is not enough). Similarly, any downwardly non-synchronizable event that reaches a fence f makes f non-reducible since a barrier must be used to enforce inter-image data dependences (a completion pair is not enough).

Step II captures propagation of the fact that if a fence used to synchronize an event is non-reducible, it is unnecessary to synchronize the event with additional point-to-point synchronization. If an event e reaches a non-reducible fence f, e is not downwardly syn-

```
procedure propagate
    while any updated field changes value

        // Step I. Propagate information from events to fences
        for each event e such that ¬e.upwardlySynchronizable
            for each fence f such that f ∈ e.fencesBeforeEvent
                f.reducible ← false
        for each event e such that ¬e.downwardlySynchronizable
            for each fence f such that f ∈ e.fencesAfterEvent
                f.reducible ← false

        // Step II. Propagate information from fences to events
        for each fence f such that ¬f.reducible
            for each event e such that e ∈ f.eventsBeforeFence
                e.downwardlySynchronizable ← false
            for each event e such that e ∈ f.eventsAfterFence
                e.upwardlySynchronizable ← false

        // Step III. Propagate information for DO loop regions
        for each DO loop region R
            // non-synchronizable events reaching body fences inside the loop
            if ¬R.prebodyFence.reducible or ¬R.postbodyFence.reducible
                call markNoSynchOptForLoop(R)
            // non-synchronizable events outside of the loop
            if (¬R.preloopFence.reducible or ¬R.postloopFence.reducible) and
              (heuristic: the loop does not always execute a barrier)
                call markNoSynchOptForLoop(R)

procedure markNoSynchOptForLoop(Region R)
    R.prebodyEvent.downwardlySynchronizable ← false
    R.postbodyEvent.upwardlySynchronizable ← false
    R.prebodyFence ← false; R.postbodyFence ← false
    R.preloopEvent.upwardlySynchronizable ← false
    R.postloopEvent.downwardlySynchronizable ← false
    R.preloopFence ← false; R.postloopFence ← false
```

Figure 7.25 : Iterative propagation step.

```
procedure finalize
    // Step IV. Heuristic: isolate synchronizable events inside a DO loop region
    // from non-synchronizable events outside the loop
    for each DO loop region R
        if ¬R.preloopFence.reducible and
          R.prebodyFence.reducible and R.postbodyFence.reducible
            insert a barrier in R.preloopFence.node
        if ¬R.postloopFence.reducible and
          R.prebodyFence.reducible and R.postbodyFence.reducible
            insert a barrier in R.postloopFence.node

    // Step V. Satisfy the assumptions of synch_context(cs) hints
    if there is synch_context(cs) at procedure entry and ¬EntryFence.reducible
        insert a barrier in EntryFence.node at procedure entry and issue a warning
    if there is synch_context(cs) at procedure exit and ¬ExitFence.reducible
        insert a barrier in ExitFence.node at procedure exit and issue a warning
```

Figure 7.26 : Post-propagation step.

chronizable. Similarly, if a non-reducible fence f reaches an event e, e is not upwardly synchronizable. Note that, in part, this is a heuristic that worked well for all scientific codes available to us. The algorithm might have synchronized a communication event with extra point-to-point synchronization to reduce additional barriers. For example, if a non-reducible barrier b_1 and a reducible barrier b_2 reach an event e from above, we could still synchronize e with a permission pair, which is redundant for b_1, but may keep the state of b_2 reducible.

Before discussing steps III and IV, we explain Step V, which ensures that the assumptions of the `synch_context(cs)` hint hold. If a `synch_context(cs)` hint is present at procedure S_1's entry and the $EntryFence$ is non-reducible, it means that SSR was not able to use permission pairs to safely optimize S_1's communication events reachable from S_1's entry. Another SSR-optimized procedure S_2, executing before an invocation of S_1, with `synch_context(cs)` hint at S_2's exit relies on the assumption that S_1's events reachable from S_1's entry are synchronized (with either point-to-point synchronization or

a barrier); therefore, SSR inserts a barrier when it fails to optimize events reachable from S_1's entry. However, SSR warns the programmer that an extra barrier is inserted. The reasoning is similar for the procedure exit. In all scientific codes that we have, SSR never inserted a barrier at procedure entry or exit.

If a `synch_context(cs)` hint is not present at procedure entry, the $EntryFence$ will make reachable events upwardly non-synchronizable. At the same time, downwardly non-synchronizable $EntryEvent$ will make all reachable barriers and synchronization fences non-reducible. Thus, SSR conservatively assumes that a downwardly non-synchronizable event may reach an invocation of the procedure. Similar reasoning holds for the procedure exit as well. There may be upwardly non-synchronizable communication events following a procedure invocation.

Note that non-synchronizable event placeholders inserted around each call site of a procedure that may execute PUT/GET (no `local` attribute). For each call site c, these placeholders will make barriers and fences that are reachable by c and barriers and fences that c reaches non-reducible. SSR conservatively assumes that c may execute a non-synchronizable communication event that may need to be synchronized by the barriers of the procedure being analyzed.

SSR propagation for DO loops and profitability heuristics

Each `DO` loop that may execute a barrier is "insulated" into a CFG `DO` loop region from the rest of the program, as shown in Figure 7.14, using four synchronization fences: $PreloopFence$, $PostloopFence$, $PrebodyFence$, and $PostbodyFence$, and four event placeholders: $PreloopEvent$, $PostloopEvent$, $PrebodyEvent$, and $PostbodyEvent$. SSR uses these helper fences and events to have more control over how to propagate barrier non-reducibility out of and into a `DO` loop region. For example, SSR may choose to insert a real barrier before `DO` loop to "guard" optimizable communication/synchronization inside the loop from an outside downwardly non-synchronizable event before the loop.

In step III, there are two possible cases. First, if there is a non-synchronizable event

inside the loop that is reachable by the $Prebody$ fence or that reaches the $Postbody$ fence, SSR *conservatively*[6] gives up optimizing the loop[7]. This case happens when an event e inside the loop R, reachable from $R.prebodyFence$ or reaching $R.postbodyFence$, cannot be optimized, making either $R.prebodyFence$ or $R.postbodyFence$ non-reducible. The $markNoSynchOptForLoop$ procedure marks all four helper fences non-reducible and four helper events non-synchronizable, propagating non-reducibility to every path into or out of the loop, to $Prebody$, $Postbody$, $Preloop$, and $Postloop$.

Propagation of non-reducibility inside a `DO` loop R is different. If SSR knows that the loop executes at least one barrier, it does not propagate outside non-reducibility into the loop. The rationale is to optimize communication events inside the loop as much as possible since loops usually execute many times. However, to "protect" the loop from outside influence, SSR may insert a barrier before or after the loop in the post-propagation step IV. The state of $R.preloopFence$ (or $R.postloopFence$) determines whether there is outside non-reducibility; $R.preloopFence$ (or $R.postloopFence$) could have received the non-reducible property only from a non-synchronizable event outside of the loop. However, what if the loop does not execute at all or might not execute a barrier as shown in the following example:

```
do i = 1, 100
  if (i == 101) then
    call barrier(cs)
  endif
enddo
```

In this case, we do not want to make $PreloopFence$ or $PostloopFence$ a real barrier, to avoid increasing the amount of synchronization. Therefore in our SSR implementation, the heuristic in step III is formulated as "there is a barrier-free path from loop $PrebodyFence$ to $PostbodyFence$ or the loop trip is zero". The heuristic formulation means that the loop does not always execute a barrier.

---

[6]There was no point to explore better heuristics without scientific codes that might use them.

[7]SSR might still be able to optimize a CFG region inside the loop that is isolated by barriers from the influence of these non-synchronizable events.

After the analysis reaches a fixed point, step IV may insert real barriers at the loop $PreloopFence$ or/and $PostloopFence$ fence places to protect loop internal synchronizable events from external non-synchronizable events. Note that the heuristic in step III guarantees that a real barrier is inserted only if the loop always executes a barrier.

7.6 Optimization of notify/wait synchronization

After reducibility analysis, for each communication event e (PUT/GET), SSR knows whether to generate a permission pair (if $e.upwardlySynchronizable$ is equal to $true$) and/or to generate a completion pair (if $e.downwardlySynchronizable$ is equal to $true$). However, placement of the permission notify and the completion wait should be optimized to overlap the permission and completion notify synchronization latencies with local computation, as discussed in Section 7.2.3. We first present an algorithm which does this, then we discuss how to eliminate redundant notify/wait synchronization.

7.6.1 Hiding synchronization latency

Initially, a permission notify can be placed immediately before a permission wait for an upwardly synchronizable communication event; a completion wait can be placed immediately after a completion notify for a downwardly synchronizable communication event. To overlap the permission notify latency with local computation, the permission notify should be moved earlier in the execution. To overlap the completion notify latency with local computation, the completion wait should be moved later in the execution. The driver procedure is shown in Figure 7.27. It moves completion waits downward and permission notifies upward. We describe the downward movement of waits first because it is simpler. Our version of SSR uses barriers (or fences) to limit the movement. For downward movement, SSR must maintain the property that a completion wait executes once iff the matching completion notify executes once. For upward movement, SSR must maintain the property that a permission notify executes once iff the matching permission wait executes once. Therefore, SSR must carefully guard the execution of the permission notify & completion wait

```
procedure moveW_cAndN_p
    for each communication event e
        if e.downwardlySynchronizable
            call moveW_cDownward(e, e.n_cPlace.node)
        if e.upwardlySynchronizable
            inputs ← SSA names referenced by e.image
            guard_{n_p} ← true
            call moveN_pUpward(e, e.w_pPlace.node, inputs, guard_{n_p})

procedure addPlace(SetOfPlaces, n, whereToInsert, guard_{n_p} = NULL)
    place ← new Place(n, whereToInsert, guard_{n_p})
    SetOfPlaces ← SetOfPlaces ∪ {place}
```

Figure 7.27 : Movement of the completion wait w_c and permission notify n_p.

and, in particular, not move the permission notify & completion wait outside of `DO` loops. Note that this version of SSR does not support point-to-point synchronization vectorization or hoisting.

The downward movement of a completion wait is simpler than upward movement of a permission notify, because the completion wait executes after the completion notify. Thus, whether the completion notify should execute can be captured in a compiler-generated guard variable; this variable is used to match the execution of the completion notify and the completion wait, as shown in Section 7.7. Note that the point-to-point synchronization statements are group-executable; therefore, the state of the guard variable is the same for all co-space images. In addition, the arguments of the communication target image specified via a `CS_Neighbor` function are evaluated at the permission wait point. Their values can be stored in compiler-generated variables and used by the completion wait to compute the origin(s) of communication, as shown in Section 7.7.

Figure 7.28 shows how downward movement is performed for a completion wait. The movement starts at the location of the corresponding completion notify, which was found during the reducibility analysis stage. When a fence is encountered, movement stops; this is a new location for the completion wait. Procedure $addPlace$, shown in Figure 7.27, creates

```
procedure moveW_cDownward(Event e, Node n)
    if n contains a fence
        // add a new place for w_c before the fence statement
        addPlace(e.w_cPlaces, n, beforeStmt)
        return
    if n ∈ FenceIRDF // a fence split point
        for each node s ∈ succ(n) // for each CFG successor
            call moveW_cDownward(e, s)
    else
        p ← ipostdom(n)
        // do not more w_c outside of a DO loop
        if ⟨n, p⟩ is a DO loop back edge
            // add a new place for w_c as the last statement of DO-loop body
            addPlace(e.w_cPlaces, n, afterStmt)
            return
        else
            call moveW_cDownward(e, p)
```

Figure 7.28 : Downward movement of a completion wait w_c.

a new $Place$ and adds it to the set of all $e.w_cPlaces$ places. When a fence split point is encountered, the procedure recurs over all CFG successors to find all reachable fences. This cannot move the completion wait out of a DO loop containing a barrier, because such a loop has a $PostbodyFence$ limiting the downward movement. Otherwise, the completion wait is moved to the immediate postdominator p node. The presented SSR version does not move the completion wait outside of a DO loop that does not execute barriers (not a region) by checking whether $\langle n, p\rangle$ is a DO loop back edge. If $\langle n, p\rangle$ is a back edge, we stop the movement. When we append a new completion wait place to the set $e.w_cPlaces$ of all completion wait CFG places, we also check for duplicates so that there is only one completion wait per place; this is incorporated into the set union operation and not shown in Figure 7.28.

Figure 7.29 shows how SSR moves a permission notify upward, earlier in the execution. The process is similar to the downward movement of a completion wait; however,

procedure $moveN_pUpward(Event\ e, Node\ n, inputs, guard_{n_p})$
  // $inputs$ are SSA names necessary to compute $e.image$ and $guard_{n_p}$
  // $guard_{n_p}$ is the guard of n_p to match n_p and w_p executions
  if n contains a fence
    // add a new place for n_p after the fence statement
    $addPlace(e.n_pPlaces, n, afterStmt, guard_{n_p})$
    return
  if $\exists\ name \in inputs$ such that $name$ is killed by $n.stmt$ or proc.-call side effects
    // add a new place for n_p after n's statement
    $addPlace(e.n_pPlaces, n, afterStmt, guard_{n_p})$
    return
  if $\exists\ name \in inputs$ such that $name$ is killed by $n.\Phi$-$nodes$
    // add a new place for n_p before n's statement unless it is a `DO` loop
    $addPlace(e.n_pPlaces, n, (n.stmt$ is a `DO` $)?\ afterStmt : beforeStmt, guard_{n_p})$
    return
  if $n \in FenceIDF$ // a fence merge point
    for each node $p \in pred(n)$ // for each CFG predecessor
      call $moveN_pUpward(e, p, inputs, guard_{n_p})$
  else
    $d \leftarrow idom(n)$
    if $\langle d, n\rangle$ is a `DO` loop body entry edge // do not move n_p outside of a `DO` loop
      // add a new place for n_p as the first statement of the `DO` loop body
      $addPlace(e.n_pPlaces, n, beforeStmt, guard_{n_p})$
      return
    else
      if $ipostdom(d) \neq n$ // moving outside of an `IF-THEN-ELSE`
        // let $d.stmt$ be an `IF-THEN-ELSE` with guard expression $guard_{if}$
        if n is the then-branch successor of d
          $guard_{n_p} \leftarrow (guard_{if} \wedge guard_{n_p})$
        else // n is the else-branch successor of d
          $guard_{n_p} \leftarrow (\neg guard_{if} \wedge guard_{n_p})$
        $inputs_{guard_{if}} \leftarrow$ SSA names referenced by $guard_{if}$
        $inputs \leftarrow inputs \cup inputs_{guard_{if}}$
      // d and n are control equivalent; same $guard_{n_p}$ and $inputs$
      call $moveN_pUpward(e, d, inputs, guard_{n_p})$

Figure 7.29 : Upward movement of a permission notify n_p.

we must also ensure that SSA names' $inputs$, used to evaluate the target image of communication $e.image$ as well as the permission notify guard $guard_{n_p}$, are available at each permission notify place. The movement starts at the corresponding permission wait node, which was found during the reducibility analysis stage. The $guard_{n_p}$ expression is initially $true$, which corresponds to executing the permission wait right after the corresponding permission notify; hence, the initial inputs are the SSA names of the $e.image$ expression (see Figure 7.27).

If a fence is encountered, the movement stops and a new place is created. The $Place$ structure also contains the guard expression used by the code generation stage to guard execution of the permission notify at the node $Place.node$ (see Section 7.7). If one of the SSA names necessary to perform guarded execution of the permission notify is killed, the movement stops and a new permission notify place is created. If a fence merge point is encountered, the movement recursively proceeds into each CFG predecessor. Similar to the downward motion, SSR cannot move the permission notify outside any `DO` loop (region or not). Otherwise, the permission notify is moved into the immediate CFG dominator.

Three cases are possible. First, if we move the permission notify outside of an `IF-THEN-ELSE` along the then-branch, we must ensure that the permission notify executes iff the control takes the true-branch during program execution. Thus, we extend the $guard_{n_p}$ with a conjunction of the `IF` guard expression $guard_{if}$. Second, if we move the permission notify outside of an `IF-THEN-ELSE` else-branch, we extend the $guard_{n_p}$ with a conjunction of the negated `IF` guard expression $\neg guard_{if}$. Third, if we move the permission notify to a control-equivalent dominator, there is no need to change the $guard_{n_p}$ (and the $inputs$ as a consequence). Accumulation of the guard expression is somewhat similar to if-conversion. Each permission notify placement node and the corresponding permission notify guard defines a *unique* control-flow path in the CFG of how execution reaches the location of the permission wait; therefore, for each execution of the permission wait, there is one and only one execution of the matching permission notify.

Note that SSR limits the motion of a permission notify and a completion wait with bar-

```
call barrier(cs)
x[left(cs)] = ...
y[left(cs)] = ...
call barrier(cs)
```

(a) Events with the same target

```
! former barrier
call notify(right(cs)) ! permission (x)
call notify(right(cs)) ! permission (y) Redundant
call wait(left(cs))    ! permission (x)
x[left(cs)] = ...
call notify(left(cs))  ! completion (x) Redundant
call wait(left(cs))    ! permission (y) Redundant
y[left(cs)] = ...
call notify(left(cs))  ! completion (y)
call wait(right(cs))   ! completion (x) Redundant
call wait(right(cs))   ! completion (y)
! former barrier
```

(b) SSR-transformed code

Figure 7.30 : Redundant point-to-point synchronization in SSR-transformed code.

riers (fences). It does not differentiate between individual inter-image data dependencies, but rather treats each variable as the entire memory, just as the barrier does, and therefore, it is a suboptimal over-approximation. However, this is sufficient to optimally SSR-optimize all CAF codes available to us that would benefit from the transformation.

7.6.2 Eliminating redundant point-to-point synchronization

We present two techniques that eliminate redundant permission & completion pairs for a set of communication events E such that every communication event e, $e \in E$, reaches a single barrier $b_{postdom}$ and e is reachable by a single barrier b_{dom}. This is a typical case for many scientific codes.

Eliminating redundant notify/wait for PUTs/GETs with the same target image

Under SSR placement for permission & completion pairs, communication events with the same target image that execute in the same communication epoch defined by textual barriers b_{dom} and $b_{postdom}$ might induce redundant point-to-point synchronization. Figure 7.30 (a) shows an example of two PUTs to the same target image, the neighbor on the left[8]. Fig-

[8] `left(cs)` is a macro for `CS_Neighbor(cs,1,-1)`. `right(cs)` is a macro for `CS_Neighbor(cs,1,+1)`

procedure $optimizeEventsWithTheSameTargetImage$
  for each pair of communication events e_1 and e_2, $e_1 \neq e_2$
    if e_1 and s_2 satisfy Rule I ($w_p^{e_1}$ dominates $w_p^{e_2}$)
      $e_2.upwardlySynchronizable \leftarrow false$
    if e_1 and e_2 satisfy Rule II ($n_c^{e_2}$ postdominates $n_c^{e_1}$)
      $e_1.downwardlySynchronizable \leftarrow false$

Figure 7.31 : Marking redundant synchronization.

ure 7.30 (b) shows SSR-transformed code, which has redundant point-to-point synchronization. The permission notify n_p^x and wait w_p^x for co-array `x` are sufficient to upwardly synchronize the PUT for `x` *and* also the PUT for `y`; therefore, n_p^y and w_p^y pair is redundant (denoted as $Redundant$ in Figure 7.30 (b)). Similarly, the n_c^y and w_c^y pair is sufficient to synchronize both PUTs and makes the n_c^x and w_c^x pair redundant (denoted as $Redundant$ in Figure 7.30 (b)).

In the general case, it is hard to determine when the targets of two communication events are the same. We do it for analyzable group-executable/non-group-executable events whose `CS_Neighbor` arguments are either the same constants or the same SSA names. We now formulate two rules, generalizing the example, for finding redundant notify/wait pair; the algorithms are straight-forward and we do not show their pseudocode.

Let e_1 and e_2 be two communication events with the same target image.

Rule I. If barrier b_{dom} is the only barrier that reaches both e_1 and e_2 and $w_p^{e_1}$ dominates $w_p^{e_2}$, then $w_p^{e_2}$ and $n_p^{e_2}$ are redundant. $n_p^{e_1}$ and $w_p^{e_1}$ are sufficient to upwardly synchronize both e_1 and e_2.

Rule II. If barrier $b_{postdom}$ is the only barrier that both e_1 and e_2 reach and $n_c^{e_2}$ post-dominates $n_c^{e_1}$, then $n_c^{e_1}$ and $w_c^{e_1}$ are redundant. $n_c^{e_2}$ and $w_c^{e_2}$ are sufficient to downwardly synchronize both e_1 and e_2.

Figure 7.31 shows pseudocode to make events with redundant synchronization pairs non-synchronizable, so that the following code-generation stage (see Section 7.7.2) does not instantiate redundant point-to-point synchronization.

```
call barrier(cs)
do i = 1, 1000
  ... compute a using b ...
  // exchange shadow regions
  a(:,M+1)[left(cs)]  = a(:,1)  ! left  neighbor shadow region
  a(:,  0)[right(cs)] = a(:,M)  ! right neighbor shadow region
  call barrier(cs)
  ... compute b using a ...
  // exchange shadow regions
  b(:,M+1)[left(cs)]  = b(:,1)  ! left  neighbor shadow region
  b(:,  0)[right(cs)] = b(:,M)  ! right neighbor shadow region
  call barrier(cs)
done}
```

Figure 7.32 : Shadow region exchange for Jacobi iteration.

Eliding redundant notify/wait for a Cartesian co-space

A symmetric nearest-neighbor exchange is a typical communication pattern found, *e.g.*, in the shadow-region exchange of Jacobi iteration. Figure 6.10 shows a visualization of the shadow region-exchange for Jacobi iteration decomposed along the second dimension onto a 1D Cartesian topology with periodic boundaries (see Section 6.4). Figure 7.32 shows the relevant piece of pseudocode, in which each process image exchanges data with its neighbor process images on the left and on the right.

When there is such symmetry, after SSR places permission & completion pairs, the completion n_c/w_c pair of a communication to the left (right) is similar to the permission n_p/w_p pair of a communication to the right (left). Only one of these two pairs is necessary to correctly enforce the corresponding inter-image data dependencies; the other pair can be elided. We select to elide the permission pair.

Let us consider a fragment of SSR-transformed code, shown in Figure 7.33, for statements `a(:,M+1)[left(cs)]=a(:,1)` and `b(:,0)[right(cs)]=b(:,M)`. The completion pair of the first PUT (for `a`) already enforces inter-image data dependencies that the permission pair was inserted to enforce for the second PUT (for `b`); therefore, the permission pair for `b(:,0)[right(cs)]=b(:,M)` is not necessary.

```
...
a(:,M+1)[left(cs)]  = a(:,1)  ! left  neighbor shadow region
call notify(left(cs))         ! completion
...
call wait(right(cs))          ! completion
...
! former barrier
call notify(left(cs))         ! permission
...
call wait(right(cs))          ! permission
b(:,  0)[right(cs)] = b(:,M)  ! right neighbor shadow region
...
```

Figure 7.33 : A fragment of SSR-generated code for Jacobi shadow region exchange.

Figure 7.34 (a) shows the SSR-transformed Jacobi shadow-region exchange code, in which each communication epoch is optimized independently from the others. Figure 7.34 (b) shows optimized code where epochs are optimized together; it has half as much of the original synchronization, as the other half was removed (crossed out statements) due to the symmetry. The barrier before the loop is necessary to upwardly synchronize communication events inside the loop that are reachable from the loop entry (since we removed the permission pairs); the overhead of the barrier before the loop is minimal compared to that of redundant point-to-point synchronization of unoptimized code in the loop. We also optimize epochs "wrapped" around the loop back-edge. Figure 7.35 (b) shows why each synchronization event was removed by SSR. Note that if we eliminated completion pairs instead of permission pairs, we would insert the barrier after the loop.

We optimize the case when each communication event is reachable by a single barrier and reaches a single barrier. Let b be the barrier[9] separating two communication epochs. The epoch $epoch_1$ precedes b; the epoch $epoch_2$ succeeds b. Figure 7.36 shows high-level pseudocode for the elision of permission pairs. Let e be a communication event in $epoch_2$ used to upwardly synchronize variable `x`. We try to move e's permission notify upward,

[9] b can correspond to $PrebodyFence$ and $PostbodyFence$ fences; ***i.e.***, it is "wrapped" around a `DO` loop back-edge.

```
! fence
do i = 1, 1000
    ! fence
    call notify(right(cs))    ! n_p^{a(:,M+1)}
    call notify(left(cs))     ! n_p^{a(:,0)}
    ... compute a using b ...
    ! exchange shadow regions
    call wait(left(cs))       ! w_p^{a(:,M+1)}
    a(:,M+1)[left(cs)] = a(:,1)
    call notify(left(cs))     ! n_c^{a(:,M+1)}
    call wait(right(cs))      ! w_p^{a(:,0)}
    a(:,0)[right(cs)] = a(:,M)
    call notify(right(cs))    ! n_c^{a(:,0)}
    call wait(right(cs))      ! w_c^{a(:,M+1)}
    call wait(left(cs))       ! w_c^{a(:,0)}
    ! fence
    call notify(right(cs))    ! n_p^{b(:,M+1)}
    call notify(left(cs))     ! n_p^{b(:,0)}
    ... compute b using a ...
    ! exchange shadow regions
    call wait(left(cs))       ! w_p^{b(:,M+1)}
    b(:,M+1)[left(cs)] = b(:,1)
    call notify(left(cs))     ! n_c^{b(:,M+1)}
    call wait(right(cs))      ! w_p^{b(:,0)}
    b(:,0)[right(cs)] = b(:,M)
    call notify(right(cs))    ! n_c^{b(:,0)}
    call wait(right(cs))      ! w_c^{b(:,M+1)}
    call wait(left(cs))       ! w_c^{b(:,0)}
    ! fence
done
```

(a) SSR-transformed code

```
call barrier(cs)
do i = 1, 1000
    ! fence
    ~~call notify(right(cs))    ! n_p^{a(:,M+1)}~~
    ~~call notify(left(cs))     ! n_p^{a(:,0)}~~
    ... compute a using b ...
    ! exchange shadow regions
    ~~call wait(left(cs))       ! w_p^{a(:,M+1)}~~
    a(:,M+1)[left(cs)] = a(:,1)
    call notify(left(cs))     ! n_c^{a(:,M+1)}
    ~~call wait(right(cs))      ! w_p^{a(:,0)}~~
    a(:,0)[right(cs)] = a(:,M)
    call notify(right(cs))    ! n_c^{a(:,0)}
    call wait(right(cs))      ! w_c^{a(:,M+1)}
    call wait(left(cs))       ! w_c^{a(:,0)}
    ! fence
    ~~call notify(right(cs))    ! n_p^{b(:,M+1)}~~
    ~~call notify(left(cs))     ! n_p^{b(:,0)}~~
    ... compute b using a ...
    ! exchange shadow regions
    ~~call wait(left(cs))       ! w_p^{b(:,M+1)}~~
    b(:,M+1)[left(cs)] = b(:,1)
    call notify(left(cs))     ! n_c^{b(:,M+1)}
    ~~call wait(right(cs))      ! w_p^{b(:,0)}~~
    b(:,0)[right(cs)] = b(:,M)
    call notify(right(cs))    ! n_c^{b(:,0)}
    call wait(right(cs))      ! w_c^{b(:,M+1)}
    call wait(left(cs))       ! w_c^{b(:,0)}
    ! fence
done
```

(b) After synchronization elision

Figure 7.34 : SSR-reduced Jacobi iteration shadow region exchange.

beyond b into $epoch_1$, perhaps, wrapping around the DO loop back-edge. If we find an event e_1 that is control equivalent with e and e_1's completion notify has the same target image as e's permission notify, then e's permission pair is redundant. While moving e's permission notify earlier in the execution, we check that x is not accessed between e_1's completion notify and b. If this is not the case, such an access creates a potential inter-image data dependence that is not downwardly synchronized with e_1's completion pair; thus, e's per-

```
! fence
do i = 1, 1000
    ! fence
    call notify(right(cs))    ! n_p^{a(:,M+1)}
    call notify(left(cs))     ! n_p^{a(:,0)}
    ... compute a using b ...
    ! exchange shadow regions
    call wait(left(cs))       ! w_p^{a(:,M+1)}
    a(:,M+1)[left(cs)] = a(:,1)
    call notify(left(cs))     ! n_c^{a(:,M+1)}
    call wait(right(cs))      ! w_p^{a(:,0)}
    a(:,0)[right(cs)] = a(:,M)
    call notify(right(cs))    ! n_c^{a(:,0)}
    call wait(right(cs))      ! w_c^{a(:,M+1)}
    call wait(left(cs))       ! w_c^{a(:,0)}
    ! fence
    call notify(right(cs))    ! n_p^{b(:,M+1)}
    call notify(left(cs))     ! n_p^{b(:,0)}
    ... compute b using a ...
    ! exchange shadow regions
    call wait(left(cs))       ! w_p^{b(:,M+1)}
    b(:,M+1)[left(cs)] = b(:,1)
    call notify(left(cs))     ! n_c^{b(:,M+1)}
    call wait(right(cs))      ! w_p^{b(:,0)}
    b(:,0)[right(cs)] = b(:,M)
    call notify(right(cs))    ! n_c^{b(:,0)}
    call wait(right(cs))      ! w_c^{b(:,M+1)}
    call wait(left(cs))       ! w_c^{b(:,0)}
    ! fence
done
```

(a) SSR-transformed code

```
call barrier(cs)
do i = 1, 1000
    ! fence
    ! n_p^{a(:,M+1)} elided: covered by n_c^{b(:,0)}
    ! n_p^{a(:,0)} elided: covered by n_c^{b(:,M+1)}
    ... compute a using b ...
    ! exchange shadow regions
    ! w_p^{a(:,M+1)} elided: covered by w_c^{b(:,0)}
    a(:,M+1)[left(cs)] = a(:,1)
    call notify(left(cs))     ! n_c^{a(:,M+1)}
    ! w_p^{a(:,0)} elided: covered by w_c^{b(:,M+1)}
    a(:,0)[right(cs)] = a(:,M)
    call notify(right(cs))    ! n_c^{a(:,0)}
    call wait(right(cs))      ! w_c^{a(:,M+1)}
    call wait(left(cs))       ! w_c^{a(:,0)}
    ! fence
    ! n_p^{b(:,M+1)} elided: covered by n_c^{a(:,0)}
    ! n_p^{b(:,0)} elided: covered by n_c^{a(:,M+1)}
    ... compute b using a ...
    ! exchange shadow regions
    ! w_p^{b(:,M+1)} elided: covered by w_c^{a(:,0)}
    b(:,M+1)[left(cs)] = b(:,1)
    call notify(left(cs))     ! n_c^{b(:,M+1)}
    ! w_p^{b(:,0)} elided: covered by w_c^{a(:,M+1)}
    b(:,0)[right(cs)] = b(:,M)
    call notify(right(cs))    ! n_c^{b(:,0)}
    call wait(right(cs))      ! w_c^{b(:,M+1)}
    call wait(left(cs))       ! w_c^{b(:,0)}
    ! fence
done
```

(b) After synchronization elision

Figure 7.35 : SSR-reduced Jacobi iteration shadow region exchange: explanation for the synchronization elision.

mission pair must be used to synchronize the dependence, and it cannot be removed. The implementation is straightforward using the information for the permission & completion pair placement. After this optimization, each upwardly synchronizable event e that was optimized is made upwardly non-synchronizable ($e.upwardlySynchronizable = false$) to avoid generating the redundant permission pair. SSR enhanced with this strategy re-

```
procedure elideSynchronizationPairs
  for each event e such that there is only one n_pPlace and n_pPlace.guard_{n_p} = true
    // let e.image^{-1} denote Cartesian inversion of e's target image expression e.image
    // try to find a "covering" completion notify by moving permission notify upward
    n ← n_pPlace.node
    wrapped ← false
    while true
      if any input of e.image is defined in n   // cannot move n_p earlier in execution
        break
      if n contains a completion notify for event e_1 such that e_1.image = e.image^{-1}
       and e_1.n_cPlace.node and n_pPlace.node are control equivalent
        // found a "covering" completion pair: elide e's permission pair
        e.upwardlySynchronizable ← false
        if wrapped = true   // wrapped around a DO loop back edge
          make R.preloopFence a real barrier
        break
      if e's co-array variable is accessed in n.stmt // potential inter-image dependence
        break
      if n is the prebody fence node R.PrebodyFence of a DO loop region R
        // wrap around the back edge
        n ← R.postbodyFence's node
        wrapped ← true
        continue
      d ← idom(n)
      if d is not control equivalent to n   // coming from within control flow
        break
      n ← d   // move upward to the immediate dominator
      if n = n_pPlace.node   // already visited the node
        break
```

Figure 7.36 : Eliding redundant permission pairs for a Cartesian co-space.

moves redundant permission pairs in the Jacobi iteration, so that the performance of an SSR-optimized version matches that of hand-optimized code that uses manually placed point-to-point synchronization.

7.7 Code generation

The information collected in the previous stages of SSR must be instantiated, which produces a faster code with better synchronization. We first describe synchronization primitives used by SSR.

7.7.1 Synchronization primitives for SSR

SSR should not use CAF's `notify` and `wait` point-to-point synchronization primitives, because the programmer may use them to synchronize the program; this might interfere with the compiler-generated code. Instead, SSR uses the `N(cs,q)` and `W(cs,r)` primitives, which are similar to `notify` and `wait`, to replace all textual barriers of co-space `cs`. `N(cs,q)` and `W(cs,r)` are not visible to the programmer and their state is private to the co-space `cs`, which allows the compiler to freely mix CAF's primitives as well as `N`/`W` of other co-spaces. `N(cs,q)` and `W(cs,r)` can only be executed by members of `cs`, and the target images `q` and `r` must be members of `cs`.

The implementation of `N`/`W` is similar to that of `notify` and `wait` *pairwise* counters. There are two possibilities. The first option is to maintain the full set of pairwise counters, one for each pair of images. This results in $O(P)$ space usage per each co-space image, where P is the size of the co-space group. Thus, this option is feasible for medium-scale parallel architectures, but might not be feasible for large-scale clusters, for which maintaining the entire set of pairwise counters may result in high memory-space overhead. As of this writing, `cafc` runtime implementation maintains the entire set of pairwise counters.

The second option is based on the observation that most scientific codes communicate only with a relatively small subset of neighbors. Thus, pairwise synchronization state should be created on-demand during execution, only when two images synchronize the first time for the co-space. This is similar to the process of establishing a connection for a multi-version variable; see Section 8.3. When image `p` waits/notifies image `q`, their pairwise synchronization state is established on both `p` and `q`, *e.g.*, by using an Active Message (AM). Each pairwise counter state is $O(1)$; more precisely, p's state with q has three in-

tegers to track how many notifies p sent to q, how many notifies p received from q, and how many notifies, received from q, p consumed. Because the counter is pairwise, an implementation can use the RDMA PUT operation to update the remote value of received notifications. The total space requirement on image p is $O(N)$, where N is the number of neighbors with which p synchronizes during execution. Since scientific codes usually synchronize many times with the same neighbors, the overhead of establishing a synchronization state is amortized. This option makes SSR `N/W` primitive implementation feasible for a large-scale parallel architecture.

Note that eventcounts [99] could potentially be used for SSR synchronization; however, the abstraction of eventcount is more general because several images can increment an event count. Therefore, an implementation would use an AM for each synchronization event unless the network hardware supports an atomic remote increment operation. The proposed pairwise scheme relies on the fact that only one image can update the remote counter state. Therefore, the state of the counter is known on both target and origin images of synchronization, and the origin image can use RDMA PUT, which is available on most interconnects, to update the target image's number of received notifies. This would result in a potentially more efficient `N/W` synchronization than an AM-based synchronization via eventcounts.

`N/W` are basic primitives. In addition, SSR uses the `Norgs` primitive to notify several origin images of n_p and the `Worgs` primitive to wait for several origin images of w_c. `Norgs(cs,args)` and `Worgs(cs,args)`, used to execute n_p and w_c of a communication event e, take the arguments of the co-space `CS_Neighbor(cs,args)` function, which is e's target image expression $e.image$. Knowing these arguments, `Norgs`/`Worgs` implementation can determine the origin(s) of communication (see Section 6.4) and use `N/W` to perform notify/wait for each origin image.

7.7.2 Code transformation

Figure 7.37 shows high-level pseudocode for SSR code generation. Implementation specific details are not shown. Code generation relies on the fact that notify/wait statements are group-executable, the run-time layer handles guards (see Section 6.4), and the co-space object contains distributed knowledge about the communication topology. For each upwardly synchronizable communication event e, we generate n_p statement for each e's n_p place $place$. If $place.guard_{n_p}$ expression is not symbolically equal to `.true.`, we generate an `IF-THEN` statement to guard the execution of the `Norgs` call, which notifies all origin images of e. Then we insert a call to `W` in the $e.w_pPlace.node$ to wait for a permission (w_p) from the target image to access data. The generated calls are inserted either before or after $place.node.stmt$ according to the value of the $place.whereToInsert$ field.

For each downwardly synchronizable event e, we first capture the values of the $e.image$'s `CS_Neighbor` function arguments in compiler-generated variables $args$ at the place of w_p ($e.w_pPlace.node$, *not* n_c); the arguments are guaranteed to be available there for both group-executable and non-group-executable events by the w_p placement algorithm (see Section 7.5.1). We insert a call to `N` in the $e.n_cPlace.node$ to indicate the completion of the data access (n_c) to the target image. Then, we generate a guard variable $\texttt{w}_\texttt{c}^\texttt{e}\texttt{Guard}$ to match the executions of n_c and w_cs. $\texttt{w}_\texttt{c}^\texttt{e}\texttt{Guard}$ is initialized to `.false.` at procedure entry. Its value becomes `.true.` via the assignment statement in $e.n_cPlace.node$ iff n_p executes . To execute only one w_c, we generate an `IF-THEN` statement that resets $\texttt{w}_\texttt{c}^\texttt{e}\texttt{Guard}$ to `.false.` and executes `Worgs` call to wait for all origin images of e. It is necessary to reset $\texttt{w}_\texttt{c}^\texttt{e}\texttt{Guard}$ because an execution can reach several w_c places; in this respect, $\texttt{w}_\texttt{c}^\texttt{e}\texttt{Guard}$ performs the same role as the n_p guards.

Finally, all reducible barriers are removed from the code. Note that some barriers (around `DO` loops or at procedure entry/exit) might be inserted during reducibility analysis.

procedure $generateCode$
  for each communication event e
    if $e.upwardlySynchronizable$
      // generate permission notifies n_ps
      for each $place \in e.n_pPlaces$
        $args \leftarrow$ arguments of $e.image$ `CS_Neighbor` function
        $stmt \leftarrow$"`call Norgs`$(cs, args)$"
        if $place.guard_{n_p}$ expression $\neq$ `.true`.
          $stmt \leftarrow$"`if` $(place.guard_{n_p})$ `then` $stmt$ `endif`"
        insert $stmt$ in $place.node$
      // generate permission wait w_p
      $stmt \leftarrow$"`call W`$(cs, e.image)$"
      insert $stmt$ in $e.w_pPlace.node$

    if $e.downwardlySynchronizable$
      generate variables $args$ to store $e.image$'s `CS_Neighbor` arguments
      assign $args$ in $e.w_pPlace.node$ // they may not be available in $e.n_cPlace.node$
      // generate completion notify n_c
      $stmt \leftarrow$"`call N`$(cs,$ `CS_Neighbor`$(cs, args))$"
      insert $stmt$ in $e.n_cPlace.node$
      // generate completion waits w_cs
      generate a guard variable $\texttt{w}_\texttt{c}^\texttt{e}\texttt{Guard}$ for e's w_c
      insert statement "$\texttt{w}_\texttt{c}^\texttt{e}\texttt{Guard} = \texttt{.false}$." at procedure entry
      insert statement "$\texttt{w}_\texttt{c}^\texttt{e}\texttt{Guard} = \texttt{.true}$." in $e.n_cPlace.node$
      for each $place \in e.w_cPlaces$
        $stmt \leftarrow$"`if` $(\texttt{w}_\texttt{c}^\texttt{e}\texttt{Guard})$ `then`
            $\texttt{w}_\texttt{c}^\texttt{e}\texttt{Guard} = \texttt{.false}$.
            `call Worgs`$(cs, args)$
          `endif`"
        insert $stmt$ in $place.node$

  // remove reducible barriers
  for each barrier b
    if $b.reducible$
      delete $b.node.stmt$ from the program

Figure 7.37 : Code generation.

7.7.3 Generation of non-blocking PUTs

The task of generating non-blocking PUTs is orthogonal to SSR and should be a part of `cafc`'s alternative code generation strategy described on Page 63, which is not yet supported. We implemented prototype support for non-blocking PUTs generation so that SSR delivers the performance of hand-optimized codes that use non-blocking PUT directives described in Section 3.1.

In our experiments, each remote access is a co-array to co-array copy A, which `cafc` optimizes not to use a temporary for the right hand side (RHS), as discussed in Section 4.1.8. To make such a PUT non-blocking, SSR checks that the SSA name of A's RHS is not redefined before the epoch closing barrier(s). If this is the case, it is safe to make PUT non-blocking and communicate data in-place.

Since `cafc` does not yet have full support for non-blocking PUTs, we implemented limited support as part of SSR only for the case in which each PUT is reachable by only one barrier b_1 and reaches only one barrier b_2, and all PUTs of the epoch can be made non-blocking; when full support for non-blocking PUTs is available, this functionality should be removed from SSR. Then, we insert the `open_nb_put_region` directive at b_1 and the `close_nb_put_region` directive, followed by the `complete_nb_put_region` directive at b_2 to instruct the run-time to issue non-blocking PUTs instead of blocking (see Section 3.1). All such PUTs are completed at b_2.

7.8 Experimental evaluation

We extended `cafc` with prototype support for group, Cartesian, and graph co-spaces, communication analysis, and SSR. As of this writing, `cafc` does not support interprocedural analysis or automatic procedure inlining. Global and group barriers are implemented using MPI barriers.

We tested the effectiveness of the SSR algorithm for Jacobi iteration and the NAS MG and CG benchmarks, described in Section 3.4. We performed our experiments on an Ita-

nium2 cluster with a Myrinet 2000 interconnect (RTC) described in Section 3.3.

We modified the benchmarks to use co-spaces and textual co-space barriers. We denote these versions as XXX-CAF-BARRIER, where the XXX- prefix stands for Jacobi-, MG-, or CG-. The communication subroutines in the NAS MG and CG benchmarks were annotated with the `local` and `synch_context(cs)` hints to compensate for `cafc`'s lack of interprocedural analysis. SSR-optimized versions of XXX-CAF-BARRIER are denoted as XXX-CAF-SSR. We compare the performance and scalability of XXX-CAF-SSR versions with our fastest hand-optimized versions (XXX-CAF-HAND), original MPI versions (XXX-MPI), and two barrier-based versions: XXX-CAF-BARRIER and XXX-CAF-GLOB-BARR. XXX-CAF-GLOB-BARR uses global barriers for synchronization rather than co-space barriers. All versions of each benchmark have the same local computation and differ only in communication and synchronization.

In summary, our experiments show that SSR-optimized versions deliver the performance of hand-optimized CAF versions and roughly the same level of performance as that of MPI versions. In comparison to barrier-based versions, the SSR-optimized versions show noticeably better scalability and deliver higher performance for executions on a large number of processors.

7.8.1 Jacobi iteration

We studied Jacobi iteration decomposed onto a 2D Cartesian processor grid with periodic boundaries. Compared with the original CAF version, using a 2D Cartesian co-space slightly simplified Jacobi-CAF-BARRIER because the Cartesian abstraction logic is hidden inside the run-time layer. The programmer just specifies parameters to the `CS_Create` call, while in the original CAF version, the programmer has to explicitly code the decomposition logic.

The Jacobi-CAF-SSR version has synchronization that is almost identical to that of Jacobi-CAF-HAND version. The only difference is that SSR inserts a barrier before the time-step loop, as explained in Section 7.6.2 (see Figures 7.34 (b) and 7.35 (b)). This

Problem size	32	64
1024^2	11.42%	16.31%
2048^2	6.13%	12.22%
4096^2	2.00%	4.38%
8192^2	1.17%	2.15%
16384^2	1.16%	1.11%

Table 7.1 : Performance improvement of Jacobi-CAF-SSR over Jacobi-CAF-BARRIER for 32- and 64-processor executions.

barrier contributes to the program execution time insignificantly, and the performance of Jacobi-CAF-SSR and Jacobi-CAF-HAND versions is virtually the same.

Table 7.1 shows the run-time improvement of the Jacobi-CAF-SSR version over the Jacobi-CAF-BARRIER version of different problem sizes for 32- and 64-processor executions. As expected, smaller problems benefit more from using faster point-to-point synchronization because the ratio of synchronization time to computation time is higher. The performance gain is larger for 64-processor executions because point-to-point synchronization is asymptotically more efficient than barrier-based synchronization and scales much better.

7.8.2 NAS MG

NAS MG performs computation on several distributed hierarchical grids (see Section 3.4); each grid is identified via a level. The number of processes assigned to each grid depends on the problem size and the total number of processors N, which is always a power of two. NAS MG might decrease the number of processors assigned to compute on a coarser grid to increase the computation to communication ratio in the border exchange `comm3` subroutine. Each grid is decomposed onto a Cartesian communication topology with periodic boundaries. The topology can be 1D, 2D, or 3D depending on the number of processors.

```
subroutine comm3(u,n1,n2,n3,level,cs)
  ...  ! modules and includes
  integer n1, n2, n3, level, axis
  double precision :: u(n1,n2,n3)
  type(Cartesian) cs             ! co-space

  call synch_context(cs)
  if (CS_IsMember(cs)) then          ! single-valued
     if (num_images() .ne. 1) then   ! single-valued guard
        do axis = 1, 3               ! single-valued range
           call barrier(cs)
           ... ! pack data into buffM(1:buff_len,1)
           buffM(1:buff_len,2)[CS_Neighbor(cs,axis,-1)] = buffM(1:buff_len,1)
           ... ! pack data into buffP(1:buff_len,1)
           buffP(1:buff_len,2)[CS_Neighbor(cs,axis,+1)] = buffP(1:buff_len,1)
           call barrier(cs)
           ... ! unpack data from buffM(1:buff_len,2)
           ... ! unpack data from buffP(1:buff_len,2)
        end do
     else
        do axis = 1, 3
           call comm1p(axis,u,n1,n2,n3,level)  ! commfree subroutine
        end do
     endif
  else
     call zero3(u,n1,n2,n3)
  endif
  call synch_context(cs)
end subroutine comm3
```

Figure 7.38 : NAS MG `comm3` boundary exchange subroutine for the MG-CAF-BARRIER version.

In this implementation, a coarser grid has a lower level number. The topology of the coarser grid at level l is either the same as or "nested" in the topology of the next finer grid at level $(l+1)$. For example, the 3D decomposition onto 16 images is $4\times2\times2$ for the level-two (and higher) grid and $2\times2\times2$ for the level-one grid. All 16 processors compute on the finer level-two grid; however, only the even-numbered processors (total 8) along dimension X compute on the coarser level-one grid. Figure 7.40 (disregarding arrows for the moment) shows one of two XY-planes of the images' topology for the level-two grid; in this diagram, darker circles correspond to the processors active in the images' topology for level one. In other words, the topology of images active as part of the coarser grid at level l along axis a may be every other image along a of the encompassing topology of the finer grid level $(l+1)$. In the original MPI version, the topologies are represented via the `nbr(a,d,l)` array of neighbors, where $a = 1, 2, 3$ is the axis, $d = -1, 1$ is the direction,

```
subroutine comm3_ex(u,n1,n2,n3,level,cs)
  ...  ! modules and includes
  integer n1, n2, n3, level, axis
  double precision :: u(n1,n2,n3)
  type(Cartesian) cs             ! co-space

  call synch_context(cs)
  if (num_images() .ne. 1) then             ! single-valued guard
    if (single(cs,level) .le. single(cs,max_ex_l)) then  ! single-valued guard
      if (CS_IsMember(cs)) then             ! single-valued
        do axis = 1, 3                      ! single-valued range
          if (single(cs,do_ex(axis,level))) then    ! single-valued
            call barrier(cs)
            if (give_ex(axis,level)) then   ! non-single-valued
              ... ! pack data into buffM(1:buff_len,1)
              buffM(1:buff_len,2)[CS_Neighbor(cs,axis,-1)] = buffM(1:buff_len,1)
              ... ! pack data into buffP(1:buff_len,1)
              buffP(1:buff_len,2)[CS_Neighbor(cs,axis,+1)] = buffP(1:buff_len,1)
            endif
            call barrier(cs)
            if (take_ex(axis,level)) then   ! non-single-valued
              ... ! unpack data from buffM(1:buff_len,2)
              ... ! unpack data from buffP(1:buff_len,2)
            endif
          end if
        enddo
      end if
    end if
  else
    do axis = 1, 3
      call comm1p_ex(axis,u,n1,n2,n3,level)  ! commfree subroutine
    end do
  endif
  call synch_context(cs)
end subroutine comm3_ex
```

Figure 7.39 : NAS MG `comm3_ex` inter-image extrapolation subroutine for the MG-CAF-BARRIER version.

and l is the level number. The `dead(l)` array specifies whether the image in involved in computation on the grid at level l.

There are two types of communication in NAS MG: a boundary exchange and an inter-processor extrapolation/interpolation between two adjacent grid levels. To exchange boundaries at level l, each image of level l topology executes the `comm3` subroutine. `comm3` exchanges cell boundaries with up to six spatial neighbors. The inter-processor extrapolation subroutine `comm3_ex` communicates data between images of two adjacent grid levels if their communication topologies are *not* identical; Figure 7.40 provides a visualization of this communication pattern along axis X for NAS MG on 16 processors (only

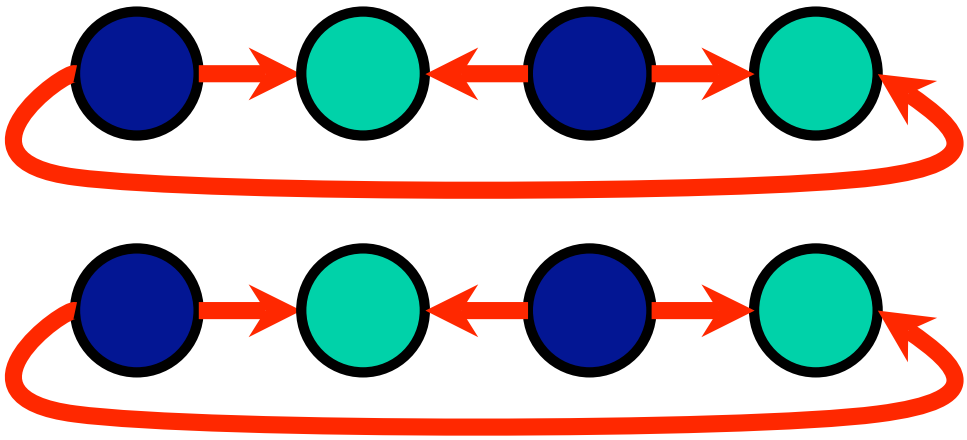


Figure 7.40 : One XY-plane of inter-processor extrapolation communication from coarser grid (level one) to finer grid (level two) in NAS MG on 16 processors.

one XY-plane shown). Let max_ex_lev be the maximum level for which MG performs the extrapolation step because the topologies of these levels are different; this is a single value. An extrapolation communication is performed by all images of the level $(l+1)$ topology. If the level l topology has two-times fewer images along axis a, the members of level l send data to both their a-axis neighbors in level $(l+1)$ topology, which receive the data (see Figure 7.40 for an example). The described two-sided communication is guarded by `give_ex` and `take_ex` arrays in the original MPI version. So as not to execute the barrier unless there is communication in `comm3_ex`, we augmented the barrier-based version with the `do_ex(a,l)` single-valued array that determines whether the extrapolation communication is necessary for axis a on level l.

The MG-MPI version uses two-sided send/receive communication where partner images are determined by the `nbr` array. The `give_ex`, and `take_ex` arrays determine which images participate in an extrapolation step.

The MG-CAF-HAND version is similar to MG-MPI. It utilizes the same `nbr`, `give_ex` and `take_ex` arrays, but uses one-sided PUTs to move data and `notify` and `wait`, guarded by the `give_ex` and `take_ex` arrays in the extrapolation step, for synchronization.

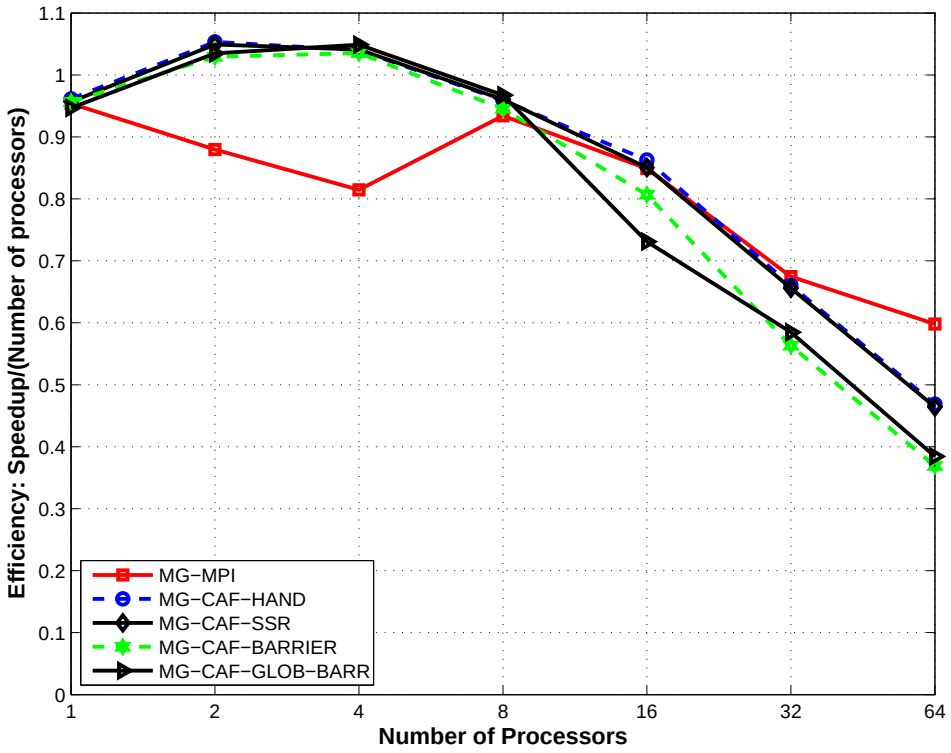


Figure 7.41 : NAS MG class A on an Itanium2 cluster with a Myrinet 2000 interconnect.

MG-CAF-BARRIER declares an array of Cartesian co-spaces `cs(l)`, one co-space cs_l per level l, which are created at the beginning of the program. Each co-space cs_l mimics the information of the `nbr(a,d,l)` array for level l. MG-CAF-BARRIER uses textual co-space barriers and the co-space `isMember(cs)` function (instead of the `dead(l)` array) to determine whether the image is a member of cs_l. The `comm3_ex` subroutine, shown in Figure 7.39, also uses `give_ex` and `take_ex` arrays to guard communication.

MG-CAF-GLOB-BARR is similar to MG-CAF-BARRIER, but uses global barrier for synchronization rather than a co-space barrier.

MG-CAF-SSR is the MG-CAF-BARRIER version optimized by SSR. The SSR optimization applied to `comm3` and `comm3_ex` reduces barriers to point-to-point synchronization. `comm3` is shown in Figure 7.38. Its synchronization is optimally reduced to that of the MG-CAF-HAND version. However, the `give_ex` array guarding communication

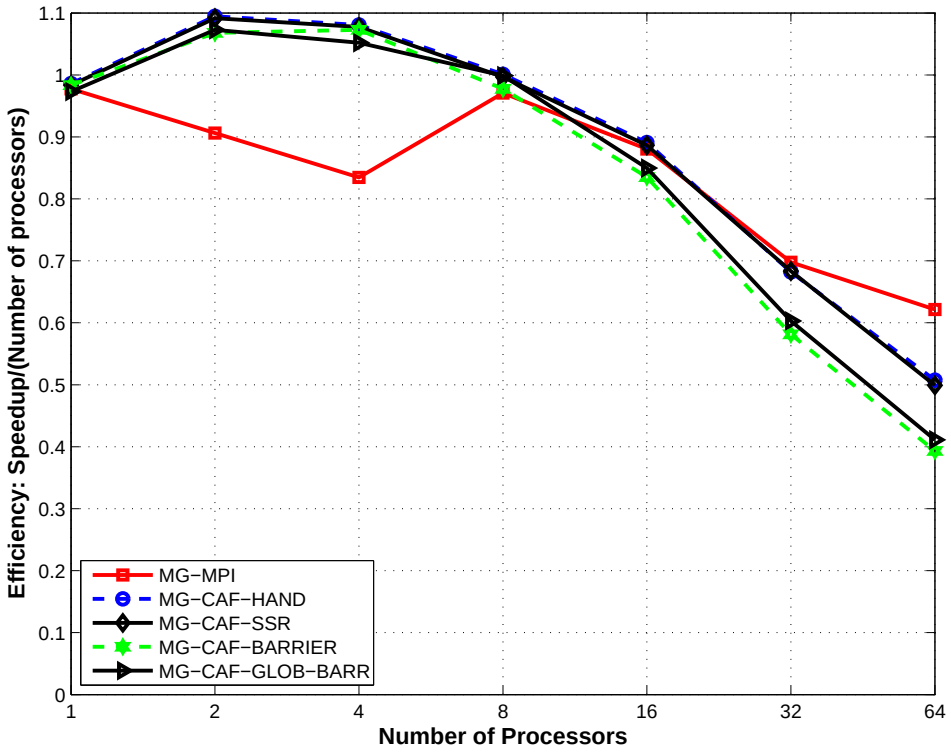


Figure 7.42 : NAS MG class B on an Itanium2 cluster with a Myrinet 2000 interconnect.

in the extrapolation subroutine `comm3_ex` (see Figure 7.39) is not single-valued. Thus, SSR places w_p and n_c outside of the corresponding `IF-THEN` statement. This results in extra notification messages because not all images communicate. In contrast, MG-CAF-HAND uses the optimal number of notification messages because the programmer knows that `give_ex` and `take_ex` arrays guard two-sided communication and can be used to precisely guard `notify` and `wait`. SSR does not have this knowledge and must place w_p and n_c around the `if (give_ex(axis,level))` statement. This induces one extra notification message per image of cs_{l+1}, totaling two times more synchronization messages vs. the hand-coded optimal solution. However, this increase does not cause performance degradation because extra notifications do not contribute to the critical path.

It is possible to make SSR use the optimal number of notifications for this example in two ways. First, an additional set of graph co-spaces can be used to express the extrapola-

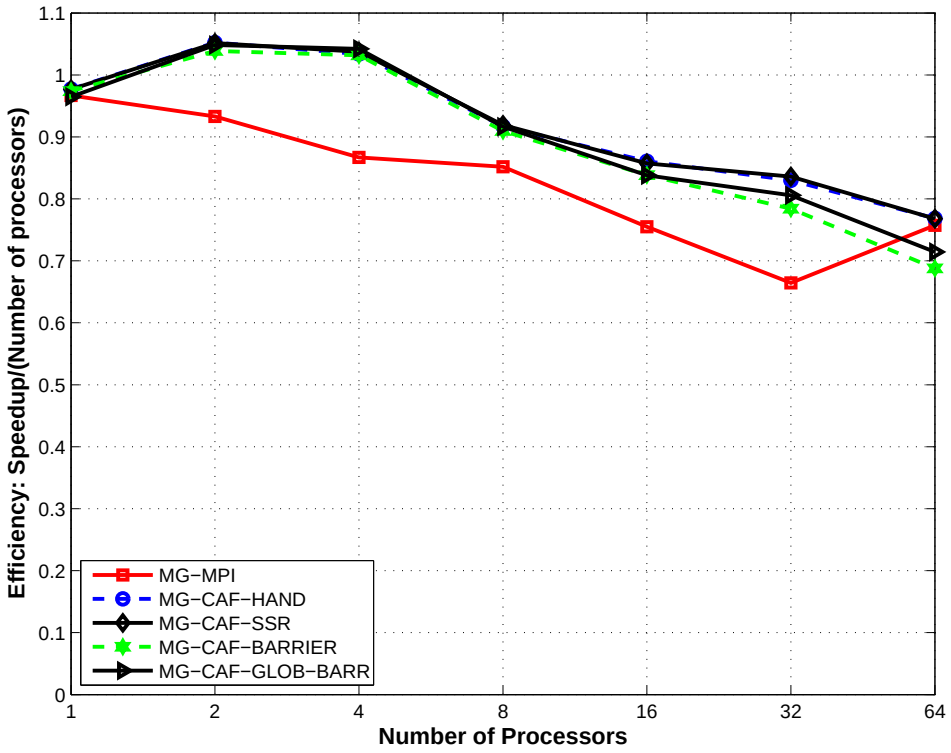


Figure 7.43 : NAS MG class C on an Itanium2 cluster with a Myrinet 2000 interconnect.

tion communication patters. Second, `give3_ex(axis,level)` can be expressed with the co-space `CS_isMember` functions of cs_l and cs_{l+1}; however, this requires extending SSR with a special-case analysis and multi-version code generation (for the case when $cs_l \notin cs_{l+1}$).

The parallel efficiency of NAS MG is shown in Figures 7.41, 7.42, and 7.43 for classes A (256^3 size, 4 iterations), B (256^3 size, 20 iterations), and C (512^3 size, 20 iterations), respectively. The performance of MG-MPI is slightly better for smaller problem sizes and somewhat worse for a larger, class C problem. The performance of MG-CAF-HAND and MG-CAF-SSR is roughly the same despite extra synchronization messages in the extrapolation step. MG-CAF-GLOB-BARR slightly outperforms MG-CAF-BARRIER, which is somewhat surprising because the former uses MPI global barriers, while the latter uses MPI group barriers that synchronize fewer process images. It appears that the MPI developers

```
subroutine transpose_exchange(w,v,send_start,exch_recv_length)
  ... ! modules and includes
  double precision w(na/num_proc_rows+2)
  double precision v(na/num_proc_rows+2)[*]
  integer send_start, exch_recv_length
  integer j

  call synch_context(cs)
  call barrier(cs)
  if (single(cs,l2npcols) .ne. 0) then      ! l2npcols is single-valued
    v(1:exch_recv_length)[CS_Neighbor(cs,1)] =        &
          w(send_start:send_start+exch_recv_length-1)
  else
    do j = 1, exch_recv_length
       v(j) = w(j)
    enddo
  endif
  call barrier(cs)
  call synch_context(cs)
end subroutine transpose_exchange
```

Figure 7.44 : Exchange with the transpose image in CG-CAF-BARRIER version.

```
subroutine scalar_sum_reduction(var)
  ... ! modules and includes
  double precision var
  double precision, save :: buf[*]
  integer i

  call synch_context(cs)
  do i = 1, single(cs,l2npcols)        ! l2npcols is single-valued
    call barrier(cs)
    buf[CS_Neighbor(cs,i+1)] = var     ! (i+1) to skip the exchange_proc
    call barrier(cs)
    var = var + buf
  enddo
  call synch_context(cs)
end subroutine scalar_sum_reduction
```

Figure 7.45 : Group scalar sum reduction for CG-CAF-BARRIER version.

optimized the more commonly used global barrier better than the group barrier. Compared to the faster barrier-based version, MG-CAF-SSR outperforms MG-CAF-GLOB-BARR by 18% for classes A and B, and by 7% for class C on 64 processors.

7.8.3 NAS CG

The NAS CG benchmark has a rather complex communication pattern. The processors are partitioned into groups that perform several types of sum reductions among the members

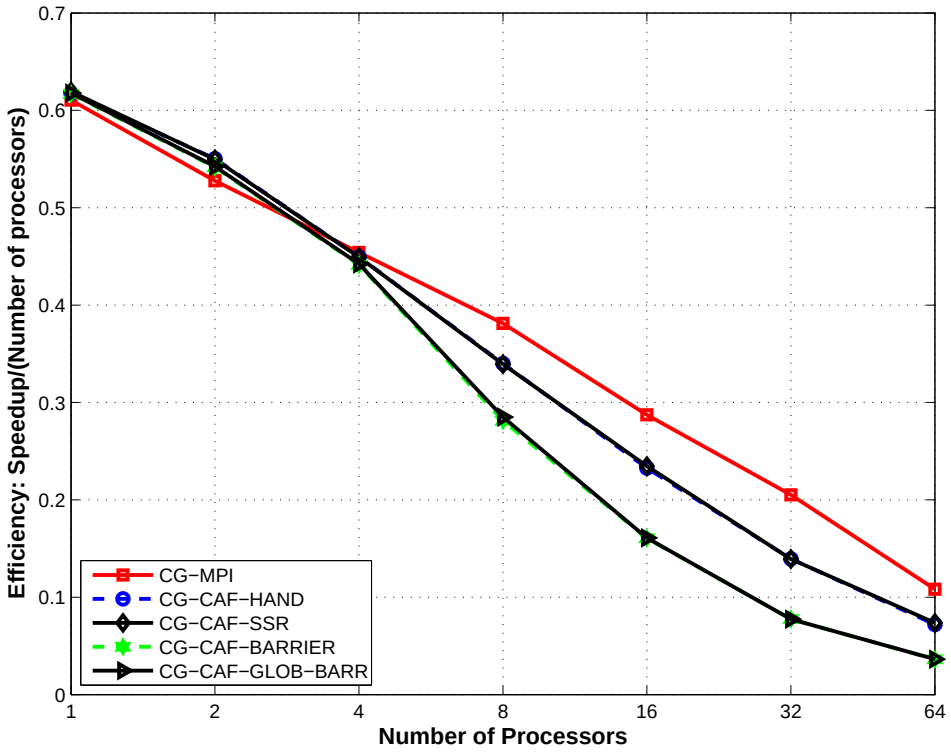


Figure 7.46 : NAS CG class A on an Itanium2 cluster with a Myrinet 2000 interconnect.

of the group. In addition, each image might communicate with the (transpose) exchange image of another group. The CG-MPI version uses the `reduce_exch_proc` array and `exch_proc` scalar variables to represent the communication neighbors for send/receive. The CG-CAF-HAND version mimics the MPI two-sided communication by using PUTs and point-to-point synchronization. The CAF-CG-BARRIER constructs a graph co-space to encapsulate the information of the `reduce_exch_proc` and `exch_proc` variables, exposing it to `cafc`. Figure 7.44 shows the code for the processor exchange. A scalar sum reduction subroutine is shown in Figure 7.45. With the help of the `synch_context(cs)` hints, SSR optimizes the synchronization of all NAS CG communication subroutines to that of the optimal hand-optimized version.

The parallel efficiency of NAS CG is shown in Figures 7.46, 7.47, and 7.48 for classes A (14000 size, 15 iterations), B (75000 size, 75 iterations), and C (150000 size, 75 iterations),

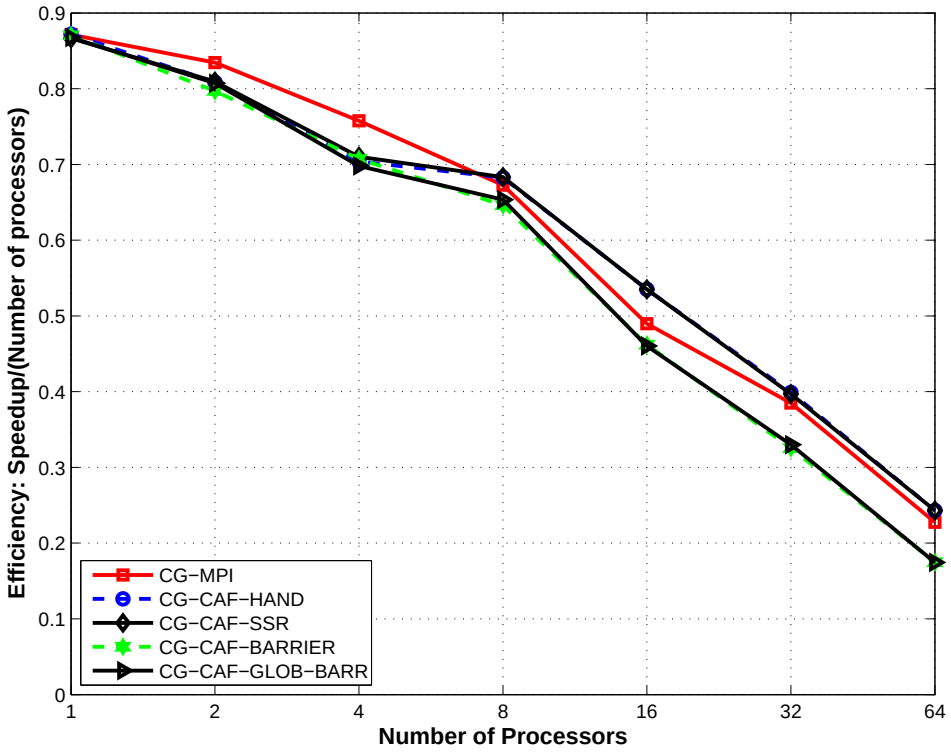


Figure 7.47 : NAS CG class B on an Itanium2 cluster with a Myrinet 2000 interconnect.

respectively. CG-MPI scales better than the optimized CAF versions for class A because MPI uses the eager protocol with implicit buffering (see Chapter 8). It shows slightly inferior performance for larger class B and C problem sizes. SSR optimizes the CG-CAF-BARRIER version into CG-CAF-SSR that has the same point-to-point synchronization as that of the optimal hand-optimized CG-CAF-HAND version. Both CG-CAF-SSR and CG-CAF-HAND versions demonstrate the same performance and scalability.

Co-space barriers in CG-CAF-BARRIER are global barriers because the CG graph co-space includes all process images. As a consequence, the performance of CG-CAF-BARRIER and CG-CAF-GLOB-BARR is the same. The SSR optimization boosts the performance of CG-CAF-BARRIER by 51% for class A, 28% for class B, and 19% for class C on 64 processors compared to the non-optimized CG-CAF-BARRIER version.

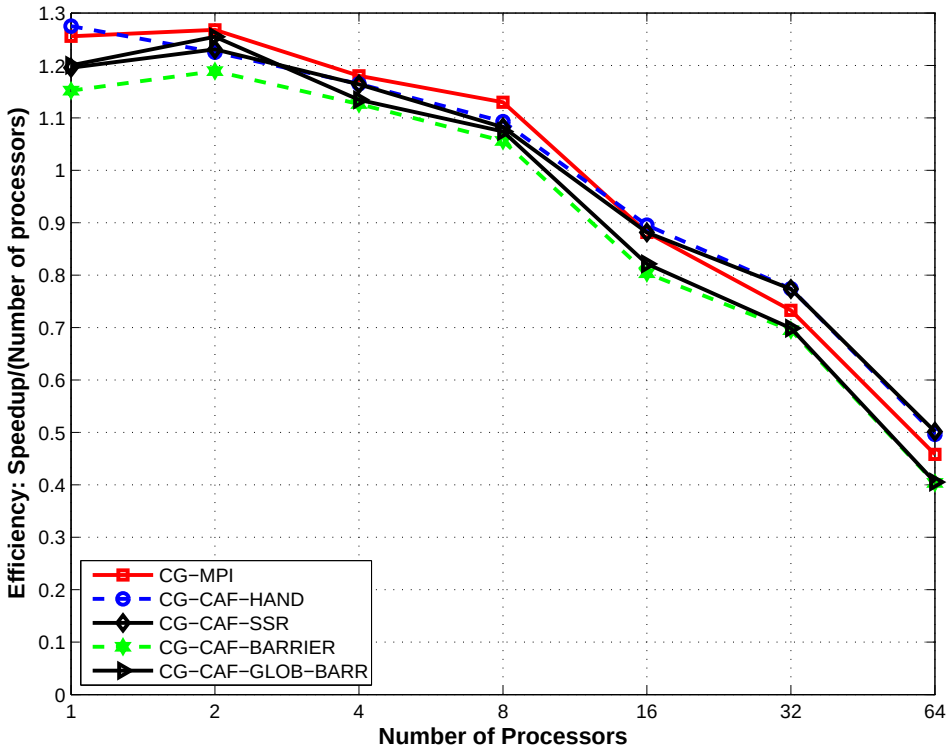


Figure 7.48 : NAS CG class C on an Itanium2 cluster with a Myrinet 2000 interconnect.

7.9 Discussion

The main contribution of Chapters 5, 6, and 7 is the novel technology that makes communication analysis of explicitly-parallel CAF programs possible via a combination of co-spaces, textual co-space barriers, and co-space single values. We identified a subset of this problem that covers nearest-neighbor codes, which include a large class of scientific applications. We devised the procedure-scope SSR transformation that analyzes and optimizes communication patterns typically found in real scientific codes. We extended `cafc` with a prototype support for SSR. Our experiments demonstrate that SSR-optimized codes show significant performance improvement and better scalability than their barrier-based counterparts; in fact, SSR-optimized versions achieve the performance level of their best hand-optimized CAF and MPI counterparts. In our experiments for 64-processor execu-

tions, SSR-optimized codes show performance improvements of up to 16.3% for Jacobi iteration, up to 18% for NAS MG, and up to 51% for NAS CG.

In the future, it would be interesting to consider several promising research directions:

1. **Interprocedural analysis for SSR.** The main shortcoming of procedure-scope SSR is the need for `synch_context(cs)` directives to increase SSR scope beyond one procedure. There are two ways to address this limitation. First, SSR can use procedure inlining; however, inlining is not possible for recursive procedures and might produce a scope with several co-spaces. Second, interprocedural SSR could analyze whether unsynchronized communication (PUTs/GETs) may reach an invocation of a procedure or unsynchronized communication may emerge after an invocation of a procedure s (unsynchronized PUTs/GETs may reach the end of s). Essentially, the interprocedural SSR analyze would collect the information conveyed to the procedure-scope SSR via `synch_context()` directives placed at procedure entry/exit. In addition, interprocedural analysis would reduce the need for `single(cs,exp)` coercion operators because some single values can be inferred from the scope surrounding a procedure invocation.

2. **Unstructured control flow.** We have not observed scientific codes that use unstructured control flow for communication; however, extending SSR to support arbitrary control flow may increase its applicability. A few modifications to the presented SSR version will enable support of unstructured control flow. First, SSR should use constraints-based inference of co-space single values and group-executable statements similar to the analysis presented by Aiken *et al.* [6]. Second, loops should be identified as strongly connected components (SCCs). w_p and n_c placement as well as n_p and w_c movement should be limited by SCC's entry and exit nodes.

3. **Hoisting and vectorization of notify/wait.** It is possible to extend our SSR algorithm to perform hoisting of the permission and/or completion pairs for a PUT/GET executed inside a loop that does not contain a barrier, if arguments of the PUT/GET's

target image expression are loop invariants. The algorithm for determining the initial placement of permission wait and completion notify in Figure 7.21 is a good candidate for such an extension. Hoisting would reduce the amount of SSR-generated point-to-point synchronization for such PUTs/GETs. Additionally, SSR can be extended to perform vectorization of the permission notify and completion wait for a PUT/GET inside a loop that does not execute a barrier, if the PUT/GET's image expression arguments are vectorizable. This would provide for more local computation to overlap permission and completion notifies with. Because we have not observed opportunities for these optimizations in existing codes, we did not pursue further investigation.

4. **Optimization of the communication primitive.** Our SSR algorithm does not change the communication primitive, but it might be beneficial to do so. After SSR analysis, it is possible to use two-sided communication primitives, *e.g.*, non-blocking send/receive, to implement communication; the implementation of these primitives can use additional memory to perform buffering, enhancing asynchrony tolerance and improving performance (see Chapter 8 for a more involved related discussion). Two-sided communication enables compiler-based packing/unpacking of strided communication; this would be the best way to achieve peak efficiency of strided data transfers. In some cases, SSR could convert GETs into PUTs (or vice versa), which has two benefits. First, architectures with RDMA support for PUT, but without RDMA support for GET, would utilize the interconnect hardware more efficiently for codes that use PUTs. Second, push-style (PUTs) communication is usually more efficient than pull-style (GETs) communication in PGAS languages because GET exposes communication latency (unless it is optimized by the compiler, which is hard).

5. **Analysis and optimization of other communication patterns.** It is possible to detect and optimize other communication patterns than analyzable group-

executable/non-group-executable PUTs/GETs. Examples of these patterns (see Section 6.4) include language-level implementation of naive reduction or broadcast, border exchange in generalized block distribution, and finite element codes.

An open question. We do not yet know how to analyze scopes that use communication/synchronization for several co-spaces. Though we have not seen several co-spaces used in one scope, such analysis would benefit codes that do. Without codes to motivate this optimization, we do not think it is worth exploring.

Chapter 8

Multi-version Variables

Many parallel applications send streams of values between processes. Processes that produce values are known as producers; ones that consume values are known as consumers. We refer to such a communication pattern as producer-consumer communication. Scientific producer-consumer codes are, for example, wavefront applications such as the ASCI Sweep3D benchmark, line-sweep applications such as the NAS BT and SP benchmarks, and loosely-synchronous applications.

A successful parallel programming model must provide a convenient way to express common communication patterns such as producer-consumer and deliver high performance at the same time. The two-sided nature of producer-consumer communication is easy to express using the message-passing primitives send and receive. Message-passing implementations of producer-consumer also achieve good performance. Partitioned Global Address Space (PGAS) languages employ the SPMD programming style with one-sided communication to read and write shared data. To achieve high performance producer-consumer communication on clusters, programmers have to explicitly manage multiple communication buffers, pipeline point-to-point synchronization, and use non-blocking communication [32]. In essence, PGAS languages are ill-suited for efficient producer-consumer communication, especially for distributed memory architectures.

We first motivate the need for better producer-consumer communication support in CAF. Next, we briefly summarize a study that we performed for the ASCI Sweep3D benchmark, a parallel wavefront application, to gain a deeper understanding of programmability and performance issues that arise with producer-consumer patterns using one-sided communication. Then, we present the concept of multi-version variables (MVVs) — a solu-

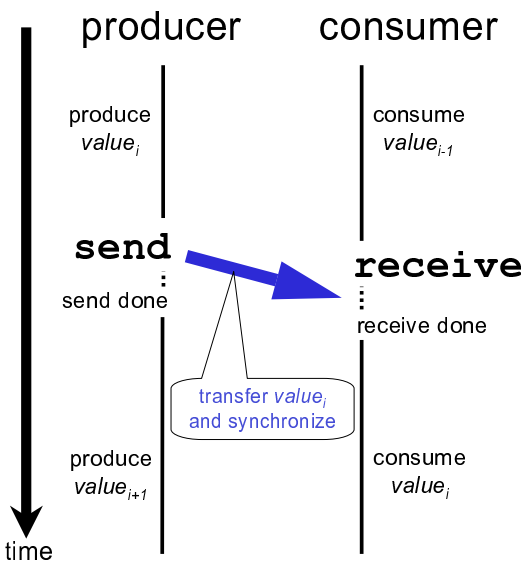


Figure 8.1 : Producer-consumer in MPI using the two-sided send and receive primitives.

tion we devised to simplify development of high performance producer-consumer codes in CAF. Finally, we present an experimental evaluation that studies the utility of MVVs for Sweep3D and the NAS BT and SP benchmarks.

8.1 Motivation

The Message Passing Interface (MPI) offers two-sided communication that is simple and natural for producer-consumer applications. The time diagram in Figure 8.1 shows how data can be transferred from a producer, which only sends data, to a consumer, which only receives data. Using the send and receive primitives is conceptually the simplest way to express producer-consumer communication and will be our “golden” standard to evaluate producer-consumer programmability. As it will be clear from the following discussion, send/receive communication can also deliver high performance by using extra storage to buffer communicated data. The send and receive primitives insulate programmers from the details of buffer management and synchronization, providing simplicity of programming. In PGAS languages, however, programmers are responsible for explicit buffer management and complex synchronization. We first consider two scenarios, called the one-buffer and

multi-buffer schemes, for supporting producer-consumer communication in CAF. Then, we discuss the progress issue with MPI buffering and available MPI send and receive primitives.

To understand why producer-consumer communication is hard to express in PGAS languages, let us consider the case of producer-consumer communication between two processes. The consumer can pull values from the producer using a GET. This would expose full communication latency and lead to performance degradation, unless the compiler can prefetch data ahead of time. However, compiler analysis for prefetching is hard for explicitly-parallel SPMD programs and is unlikely to be effective for many codes. Therefore, we focus on the case where the producer pushes values to the consumer. Process `p` produces a value and transfers it to the consumer process `q` using a PUT to store the value in a shared variable `buffer`. In one-sided communication model, synchronization must be used to signal the consumer that the value is available; or in compiler terminology, to enforce the interprocessor true data dependence. However, the producer cannot PUT a new value into `buffer` unless the consumer finished using the current value stored in `buffer`. To avoid having the producer overwrite a value in `buffer` that is still in use, the consumer must signal the producer when it is save to overwrite `buffer`; or in compiler terminology, to enforce the interprocessor anti-dependence due to `buffer` reuse. Figure 8.2 provides a visualization of this scenario. The dotted lines denote waiting time due to exposed anti-dependence synchronization latency on the producer or due to exposed communication and true dependence synchronization on the consumer.

While it is not possible to avoid the synchronization due to the true data dependence, it is possible to avoid the synchronization due to the anti-dependence if a new memory location is used for every produced value. In reality, memory is limited and must be reused. Let us consider two possible implementations that we call the one-buffer and multi-buffer schemes.

The one-buffer scheme uses only one variable `buffer` to store the values. The synchronization between the producer and consumer must enforce the anti-dependence before

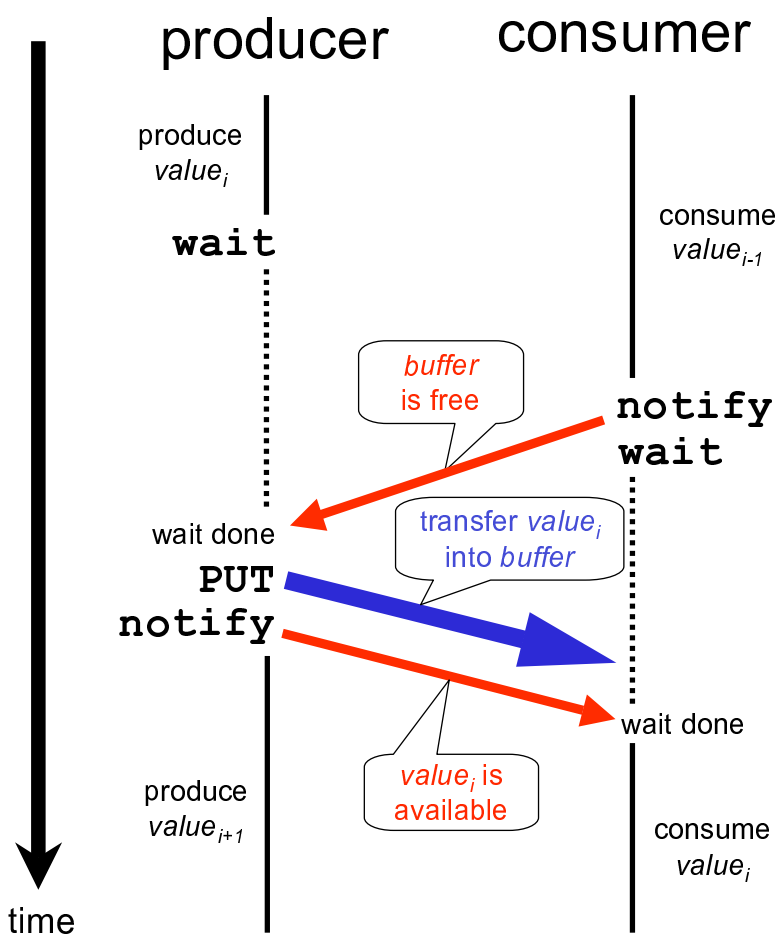


Figure 8.2 : Producer-consumer in CAF using one buffer.

the producer can safely PUT a new value into the consumer's `buffer`; this corresponds to the arrow labeled as "*buffer* is free" in Figure 8.2. This synchronization may delay the producer from transferring a newly produced value to the consumer and computing the next value for two reasons. First, the latency of the synchronization operation is exposed. Second, and more important, the consumer can safely synchronize only when it finished using the current value in `buffer`, which can be past the time when the producer finishes producing $value_i$ and arrives at the `wait` synchronization event. This leads to the producer and consumer "coupling" and a non-asynchrony tolerant program with low performance. Not only the consumer must wait for the producer to deliver a new value (this is unavoidable), but also the producer must wait for the consumer's buffer to become available (as we

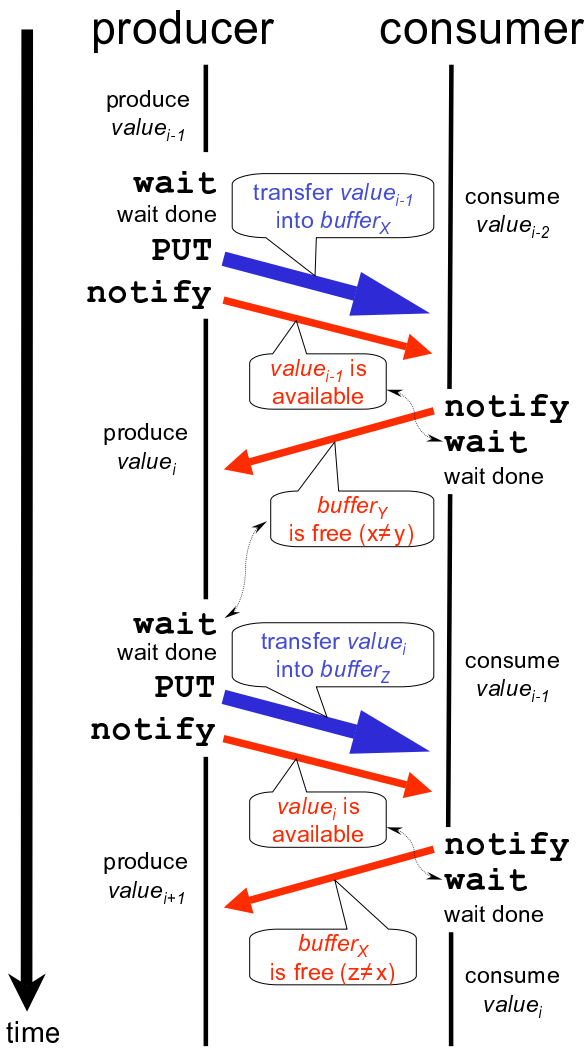


Figure 8.3 : Producer-consumer in CAF using multiple buffers.

shall see, this can be avoided by using several buffers). In other words, any delay in either the producer or consumer causes the delay in the other.

The multi-buffer scheme uses several independent buffers, *e.g.*, elements of an array `buffer(M)`. This allows the producer and consumer to work more independently and results in asynchrony tolerant code. The producer can PUT a new value into a free buffer, *e.g.*, $\texttt{buffer}_\texttt{Z}$, while the consumer uses another instance $\texttt{buffer}_\texttt{X}$, as shown in Figure 8.3. This has the effect of moving the anti-dependence synchronization, *e.g.*, the arrow labeled

as "$buffer_X$ is free" in Figure 8.3, earlier in time, overlapping it with computation on the producer. As soon as the consumer is done using $\texttt{buffer}_\texttt{X}$, it can notify the producer that $\texttt{buffer}_\texttt{X}$ is free; this corresponds to the arrow labeled as "$buffer_X$ is free" in Figure 8.3. The early notification removes the anti-dependence synchronization latency from the critical path.

Implementing producer-consumer communication patterns in PGAS languages using one-sided communication is awkward because programmers must manage synchronization for the true and anti-dependencies in addition to the data movement. With one buffer, the latency of communication and synchronization is exposed. Using multiple buffers may hide the latency; however, this requires enormous programming effort to manage buffers and carefully place the anti-dependence synchronization to remove it from the critical path. In addition, to achieve the best performance, data movement should be non-blocking to overlap communication (PUT) latency with computation on the producer. To summarize, one-sided communication and explicit synchronization of PGAS languages are not suited well for expressing two-sided in nature producer-consumer communication.

In contrast, two-sided communication (*e.g.*, MPI send/receive) offers simpler and more natural programming style for producer-consumer applications. Programmers can simply use MPI send and receive for both data movement and synchronization, as shown in Figure 8.1. Two-sided communication offers implicit synchronization and buffer management with their implementation hidden inside the MPI library. However, since MPI primitives must be general enough to handle arbitrary communication, this can result in suboptimal performance due to extra memory copying/registration/unregistration and exposed synchronization and data transfer latency, especially when MPI uses the rendezvous protocol for large messages (see Section 2.4.1). As we showed in our prior study [32], the performance of multi-buffer code in CAF can exceed that of MPI code.

So far, we have considered producer-consumer communication in which the producer only sends data and the consumer only receives data. A typical communication pattern is the exchange of shadow regions in loosely-synchronous codes such as Jacobi iteration.

Structuring an SPMD program so that processes send data and then receive data might lead to a deadlock. The reason is that the send primitive (`MPI_Send`) is blocking and, in general, needs a matching receive (`MPI_Recv`) to transfer data and unblock. For example, when MPI uses rendezvous protocol, if each process executes send, all processes block waiting for matching receives, which are never executed, leading to a deadlock. There are several ways to avoid the deadlock for such codes; we briefly mention some of them here, while a detailed explanation can be found elsewhere [61]. First, if MPI uses the eager protocol (see Section 2.4.1), `MPI_Send` is usually non-blocking; however, this is not guaranteed by the MPI standard. Second, programmers can order sends and receives, so that some processes execute sends, while others execute receives. This might not be possible or easy for some codes. Third, programmers can use the combined send and receive `MPI_Sendrecv` primitive; however, this does not allow codes in which each process image consecutively performs several sends followed by corresponding receives. Fourth, programmers can use buffered sends `MPI_Bsend`. They must register a buffer before using `MPI_Bsend` and determine the adequate buffer size. Finally, programmers can use non-blocking receive `MPI_Irecv` or non-blocking send `MPI_Isend`, which "register" a receive or send buffer with MPI temporarily, for one receive/send operation; `MPI_Wait` must be used to complete non-blocking operations.

The producer-consumer communication patterns are typical in many applications such as Sweep3D, the NAS benchmarks, Jacobi iteration, and in general, any application that performs shadow region exchange. We argued in Chapter 7 that it is simpler to use barriers for synchronization, while the SSR optimization can replace barriers with more efficient point-to-point synchronization. However, it is not natural to use barriers for synchronization in all cases; *e.g.*, using point-to-point synchronization allows a more natural programming style of wavefront computations such as Sweep3D. Also, SSR is not applicable in all cases, and SSR does not perform buffer management, which is important for hiding latency in producer-consumer communication.

Since producer-consumer communication is difficult to program in CAF and hard to

```
do iq = 1, 8              ! octants
  do mo = 1, mmo          ! angle pipelining loop
    do kk = 1, kb         ! k-plane pipelining loop

      receive from east/west into Phiib    ! recv block I-inflows
      receive from north/south into Phijb  ! recv block J-inflows

      ...
      ! computation that uses and updates Phiib and Phijb
      ...

     send Phiib to east/west               ! send block I-outflows
     send Phijb to north/south             ! send block J-outflows

    enddo
  enddo
enddo
```

Figure 8.4 : Sweep3D kernel pseudocode.

optimize by compiler, we explore extending CAF with a language construct — **multi-version variables** — that simplifies development of high performance producer-consumer codes. Multi-version variables offer the simplicity of MPI two-sided programming style and deliver performance of hand-coded multi-buffer solution. Programmers specify how many versions (buffers) an MVV should have both for correctness (to avoid deadlock and to make progress) and performance. The `commit` primitive is equivalent to send and is used to enqueue a new version to an MVV. The `retrieve` primitive is equivalent to receive and is used to dequeue the next versions from an MVV. Next, we recap our evaluation study of Sweep3D wavefront application, which further motivates the need for MVVs.

8.2 Sweep3D case study

The benchmark code Sweep3D [4] represents the heart of a real Accelerated Strategic Computing Initiative application; see Section 3.4.2 for description. Sweep3D exploits two-dimensional wavefront parallelism on a 2D logical grid of processors, shown in Figure 3.2. Figure 8.4 shows pseudocode representing a high-level view of the Sweep3D kernel.

```
...
if (receiving from I_pred) then
    !  notify I_pred that the local Phiib buffer is ready to accept new data
    call notify(I_pred)
    !  wait for the new data to arrive from the I_pred
    call wait(I_pred)
endif
!  similar for the J-dimension

...
!  computation that uses and updates Phiib and Phijb
...

if (sending to I_succ) then
    !  wait for I_succ notification that its Phiib is ready to accept new data
    call wait(I_succ)
    !  transfer the data to the I_succ using contiguous non-blocking PUT
    start the region of non-blocking PUTs
    Phiib(:,:,:)[I_succ] = Phiib(:,:,:)
    !  notify the I_succ that the new data has been sent
    call notify(I_succ)
    stop and complete the region of non-blocking PUTs
endif
!  similar for the J-dimension
...
```

Figure 8.5 : Sweep3D-1B kernel pseudocode.

8.2.1 Programmability

To investigate the impact of different CAF coding styles, we implemented several CAF versions of the Sweep3D and compared their performance with that of the MPI version — Sweep3D-MPI. The difference among the versions is in the communication and synchronization implementation, while the local computation is similar. Here, we consider only two CAF versions that we developed: Sweep3D-1B, which uses one communication buffer per dimension, and Sweep3D-3B, which uses three communication buffers per dimension. The complete details of the study can be found elsewhere [32].

Sweep3D-1B was developed from the original MPI code by declaring its `Phiib` and `Phijb` arrays as co-arrays and using non-blocking PUT to communicate them in-place. For the I-direction communication, the code is presented in Figure 8.5; `I_pred` and `I_succ` denote the predecessors and successors in the sweep for the I-dimension. For the J-direction communication, the code is similar except that the process image communicates the `Phijb` array with its J-predecessor and with its J-successor.

```
!  start the ‘‘available buffer notification pipeline’’ (only one recv buffer)
call notify(I_pred)
...
if (receiving from I_pred) then
    !  wait for the data from the I_pred
    call wait(I_pred)
endif
!  similar for the J-dimension

...
!  computation that uses and updates Phiib and Phijb
...

if (sending to I_succ) then
    finalize the previous non-blocking PUT to the I_succ
    !  now one of the buffers is free; make it the next receive buffer
endif
if (receiving from I_pred) then
    !  notify the I_pred that there is a buffer available to receive new data
    call notify(I_pred) !  this matches wait(I_succ) from the previous iteration
endif
if (sending to I_succ) then
    !  wait for I_succ notification that it has a buffer ready to accept new data
    call wait(I_succ)
    start the region of non-blocking communication with index phiib_wrk_idx
    !  transmit the new data to the I_succ using non-blocking contiguous PUT
    Phiib(:,:,:,phiib_wrk_idx)[I_succ] = Phiib(:,:,:,phiib_wrk_idx)
    !  notify the I_succ that more data is available
    call notify(I_succ)
    stop the region of non-blocking communication with index phiib_wrk_idx
    advance phiib_wrk_idx for the next stage
endif
!  similar for the J-dimension
...
!  wind down the ‘‘available buffer notification pipeline’’ (only one recv buffer)
call wait(I_succ)
```

Figure 8.6 : Sweep3D-3B kernel pseudocode.

The Sweep3D-1B communication is very similar to that of the MPI version when MPI uses the rendezvous protocol. The data movement statement — assignment to `Phiib` — communicates the same data as the send/receive pair of the MPI version. The `notify` and `wait` provide synchronization analogous to that induced by an MPI send/receive pair. For Sweep3D-1B, there is no data copy from `Phiib` or `Phijb` into an auxiliary communication buffer; the data is delivered directly in-place. In contrast, the MPI version might use additional memory registration/unregistration or extra data copies to/from a communication buffer to move the data, if `Phiib` and `Phijb` are not allocated in registered memory and the interconnect hardware requires communicated data to reside in registered memory.

Sweep3D-3B aims to overlap PUTs with computation on the successor to hide communication latency. It uses additional storage, namely, three instances of `Phiib` and `Phijb` to overlap communication with computation. In the wavefront steady state, one instance of `Phiib` can be used to receive the data from the I-predecessor; at the same time another instance can be used to perform the local computation, while the third instance can be used to hold data being communicated to the I-successor (three-buffer scheme). For shared-memory architectures without hardware support for asynchronous data transfers such as SGI Altix 3000, in which all data transfers are performed with load/store, a two-buffer scheme, in which one buffer is used for local computation and the other is used for a PUT performed by a predecessor, is likely to yield the best performance. To manage the instances as a circular-buffer (to avoid unnecessary copies), we added an extra high order dimension to `Phiib`. Similarly, we use three instances of `Phijb` to enable communication and computation overlap for the wavefront parallelism in the J-direction. The simplified pseudocode for the three-buffer scheme is given in Figure 8.6.

Note that more than three buffers can be used. Our implementation is general and supports an arbitrary number of buffers holding incoming data from the predecessor and outgoing data to the successor. However, we did not encounter a case that using more additional buffers improved performance. On platforms that support non-blocking PUT and `notify`, the code uses non-blocking communication directives [47] (see Section 4.1.8) to overlap communication with local computation, otherwise only blocking PUT is used.

8.2.2 Performance

We evaluated the performance of our CAF and MPI variants of Sweep3D on four platforms described in Section 3.3: an Alpha+Quadrics (Elan3) cluster, an Itanium2+Quadrics (Elan4) cluster, an Itanium2+Myrinet2000 cluster, and an SGI Altix 3000.

For the Sweep3D benchmark, we compare the parallel efficiency of the MPI and CAF versions; the parallel efficiency metric is explained in Section 3.4. We use efficiency rather than speedup or execution time as our comparison metric because it enables us to accurately

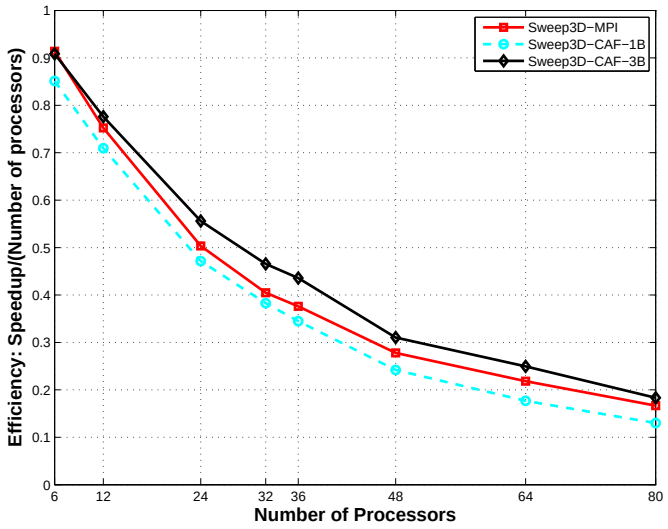


Figure 8.7 : Sweep3D of size 50x50x50 on an Alpha cluster with a Quadrics Elan3 interconnect.

gauge the relative performance of multiple benchmark implementations across the *entire* range of processor counts. Sweep3D-MPI shows the efficiency of the standard MPI version; Sweep3D-1B and Sweep3D-3B stand for the efficiency of the one- and multi-buffer CAF versions. We present results for sizes 50x50x50, 150x150x150, and 300x300x300, with per job memory requirements of 16MB, 434MB, and 3463MB, respectively.

The results for the Alpha cluster with a Quadrics Elan3 interconnect are shown in Figures 8.7, 8.8, and 8.9. The results for the Itanium2 cluster connected with Quadrics Elan4 are presented in Figures 8.10, 8.11, and 8.12. Figures 8.13, 8.14, and 8.15 displays the results for the Itanium2 cluster with a Myrinet 2000 interconnect. Finally, the results for the SGI Altix 3000 machine are given in Figures 8.16, 8.17, and 8.18.

Our results show that for Sweep3D we usually achieve scalability comparable to or better than that of the MPI version on the cluster architectures and outperform MPI by up to 10% on the SGI Altix 3000 with hardware shared memory.

On the Alpha cluster, the Sweep3D-3B version slightly outperforms the MPI version for the 50x50x50 problem size, while MPI outperforms Sweep3D-3B for the 150x150x150

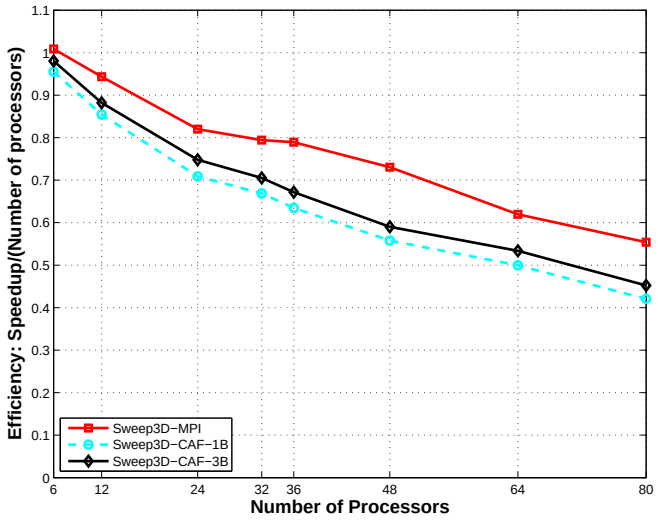


Figure 8.8 : Sweep3D of size 150x150x150 on an Alpha cluster with a Quadrics Elan3 interconnect.

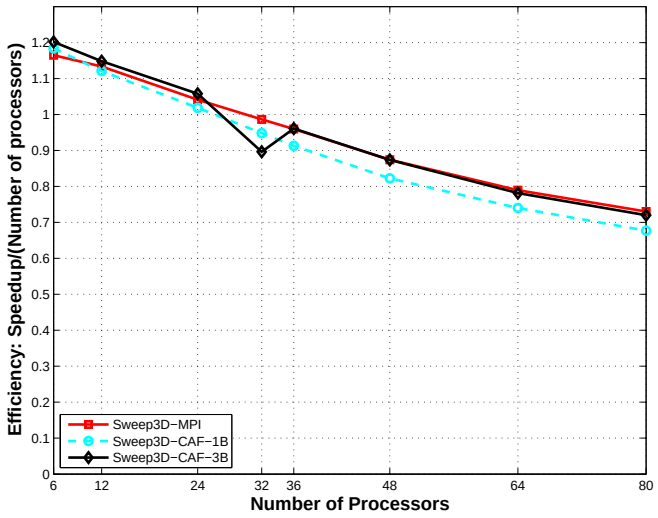


Figure 8.9 : Sweep3D of size 300x300x300 on an Alpha cluster with a Quadrics Elan3 interconnect.

problem size, and they perform comparably for the 300x300x300 problem size[1]. The

[1]We do not have access to the experimental platform to measure why the speedup is superlinear for the 300x300x300 problem size. However, a plausible explanation is that parallel versions have larger cumulative cache size than the one-processor sequential Sweep3D version.

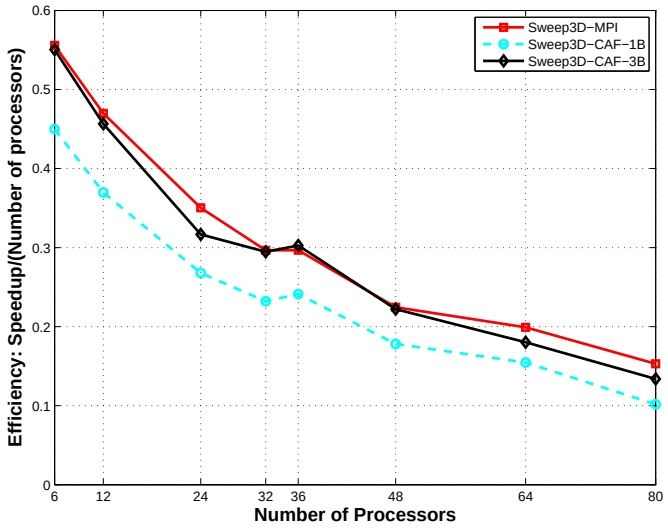


Figure 8.10 : Sweep3D of size 50x50x50 on an Itanium2 cluster with a Quadrics Elan4 interconnect.

Sweep3D-3B version enables better asynchrony tolerance; by using multiple communication buffers, it reduces the wait time of the producer process image for a buffer to become available to transfer data, using a PUT, to the consumer process image. On this platform, the ARMCI implementation of `notify` uses a memory fence and, thus, is blocking. While we can overlap the PUT to the successor with the PUT from the predecessor (both performed as independent RDMA by the NIC, as described in Section 3.2), we cannot overlap the PUT with computation on the producer process image. As expected, the one-buffer version Sweep3D-1B performs worse than the multiple-buffer Sweep3D-3B version and the MPI version because the synchronization and communication latency induced by buffer reuse anti-dependency is on the critical path.

On the Itanium2 cluster with a Quadrics Elan4 interconnect, MPI and Sweep3D-3B outperform Sweep3D-1B for the 50x50x50 problem size because the application is communication bound and Sweep3D-1B exposes the latency of the buffer reuse anti-dependence synchronization. Since the messages for the 50x50x50 problem size are small, MPI uses the eager protocol, performing library-level buffering, that removes the anti-dependence synchronization from the critical path and overlaps it with the computation. Similarly,

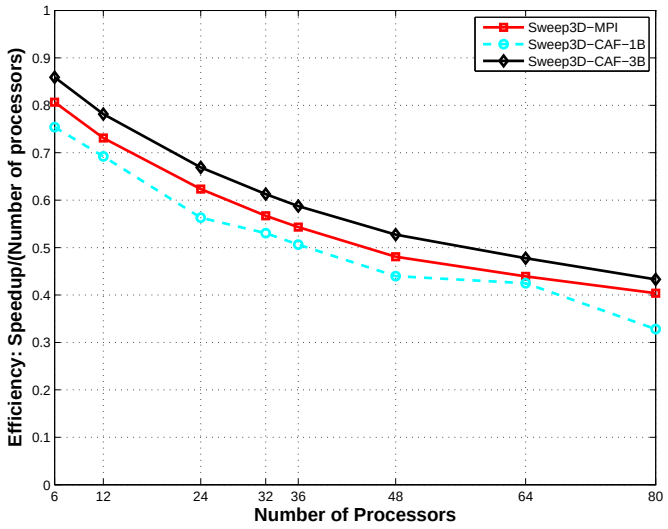


Figure 8.11 : Sweep3D of size 150x150x150 on an Itanium2 cluster with a Quadrics Elan4 interconnect.

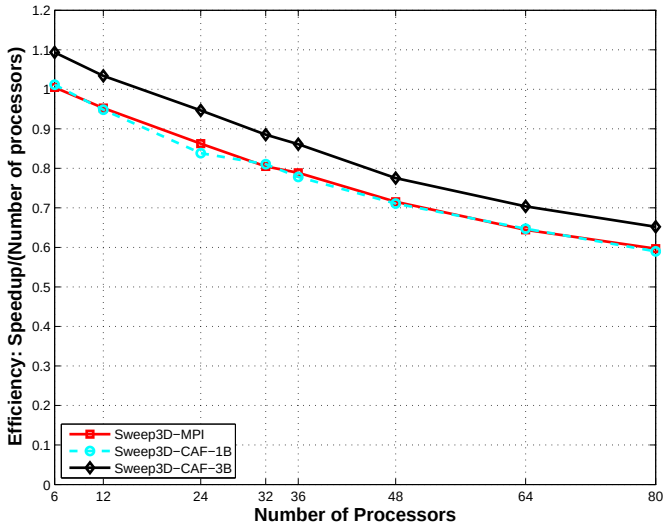


Figure 8.12 : Sweep3D of size 300x300x300 on an Itanium2 cluster with a Quadrics Elan4 interconnect.

Sweep3D-3B hides the synchronization latency by using multiple communication buffers. For this problem size, the performance of MPI and Sweep3D-3B is roughly similar. For a larger 150x150x150 problem size, MPI and Sweep3D-3B also outperform Sweep3D-1B.

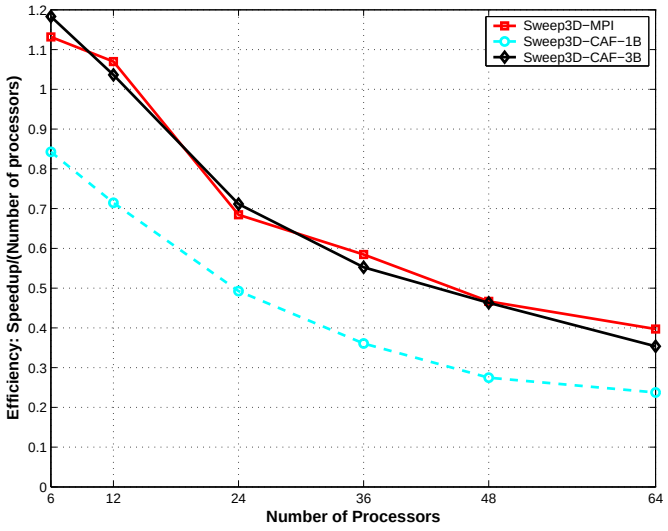


Figure 8.13 : Sweep3D of size 50x50x50 on an Itanium2 cluster with a Myrinet 2000 interconnect.

However, Sweep3D-3B shows noticeably better performance than that of MPI, which uses the eager protocol. We were not able to determine exactly what causes this, but a plausible explanation is that MPI performs extra memory copies while using the eager protocol, polluting the cache and leading to the performance degradation. For a large 300x300x300 problem size[2], the communication message size is large; thus, MPI switches to the rendezvous protocol and shows the performance equivalent to that of Sweep3D-1B. Sweep3D-3B still uses extra communication buffers similar to the MPI's eager protocol and enjoys 10% higher performance due to removing the synchronization from the critical path.

On the Itanium2 cluster with a Myrinet 2000 interconnect, the MPI version outperforms Sweep3D-1B for the 50x50x50 problem size and shows comparable performance for the 150x150x150 and 300x300x300 problem sizes. The Sweep3D-3B version performs comparably to the MPI version for the 50x50x50 problem size and outperforms it for the 150x150x150 and 300x300x300 problem sizes. For Sweep3D, performance is primarily

[2]The speedup of Sweep3D-3B is superlinear because the efficiency is computed relative to a "synthetic" serial execution time of Sweep3D, which was computed as the time of the MPI version on 6 processors multiplied by 6 since the cluster configuration did not allow us to run the sequential Sweep3D on one node.

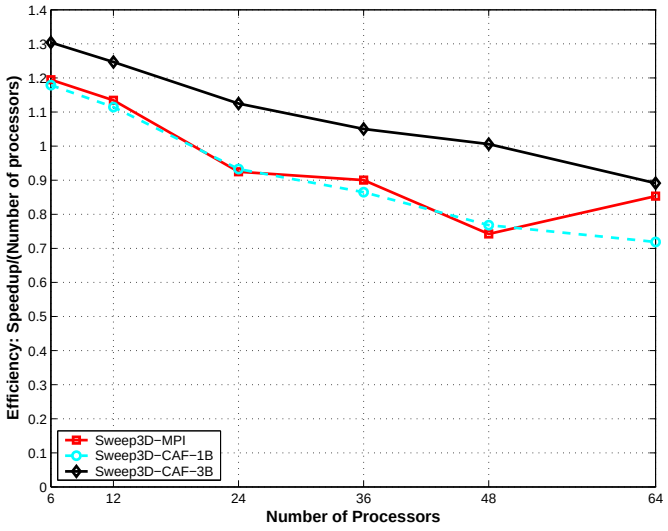


Figure 8.14 : Sweep3D of size 150x150x150 on an Itanium2 cluster with a Myrinet 2000 interconnect.

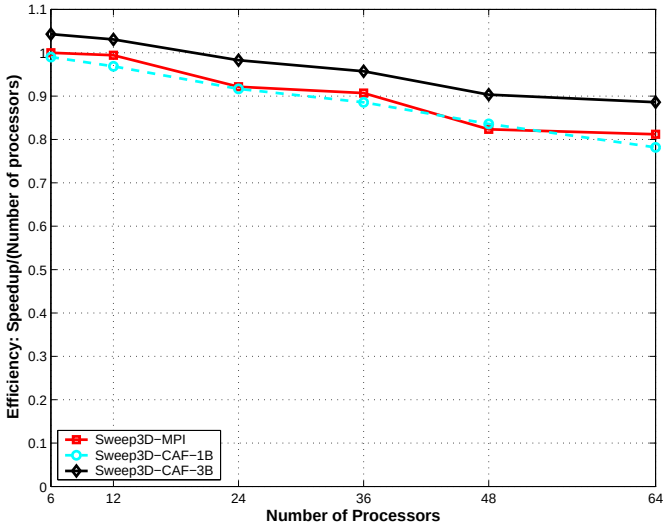


Figure 8.15 : Sweep3D of size 300x300x300 on an Itanium2 cluster with a Myrinet 2000 interconnect.

determined by how quickly the next value of `Phiib` and `Phijb` can be delivered to the remote memory (to the consumer). Using several communication buffers or the eager protocol reduces this latency by removing it from the critical path. The speedups are superlin-

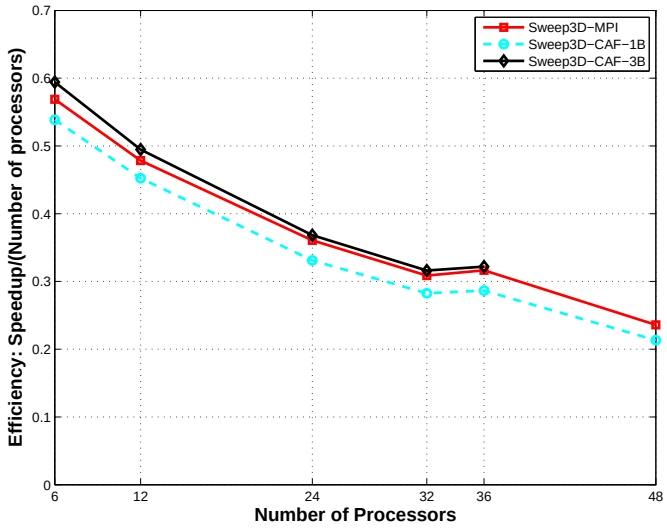


Figure 8.16 : Sweep3D of size 50x50x50 on an SGI Altix 3000.

ear for the 50x50x50 and 150x150x150 problem sizes because the number of L3 data cache misses in sequential Sweep3D version is significantly higher than that in parallel versions for this architecture. Using HPCToolkit [100, 79], we determined that, for the 50x50x50 problem size, the total (per job) number of L3 data cache misses is 5.1 times larger for sequential version compared to that of the 6-processor MPI version; for the 150x150x150 problem size, this number is 2.81 times larger. We attribute this difference to larger combined L3 cache size in parallel execution. For the 300x300x300 problem size, it was not possible to measure the execution time of the serial version because of memory constraints; instead, we use “synthetic” serial execution time computed as the time of 6-processor MPI version multiplied by 6.

An SGI Altix 3000 does not have non-blocking communication; thus, the Sweep3D-3B version uses only two buffers. The MPI version shows better performance than Sweep3D-1B for the 50x50x50 problem size due to using the eager protocol and removing synchronization from the critical path. It looses to Sweep3D-3B because of extra memory copying while Sweep3D-3B communicates data in-place. The Sweep3D-1B and MPI versions perform comparably for the 150x150x150 problem size. For the 300x300x300 problem size,

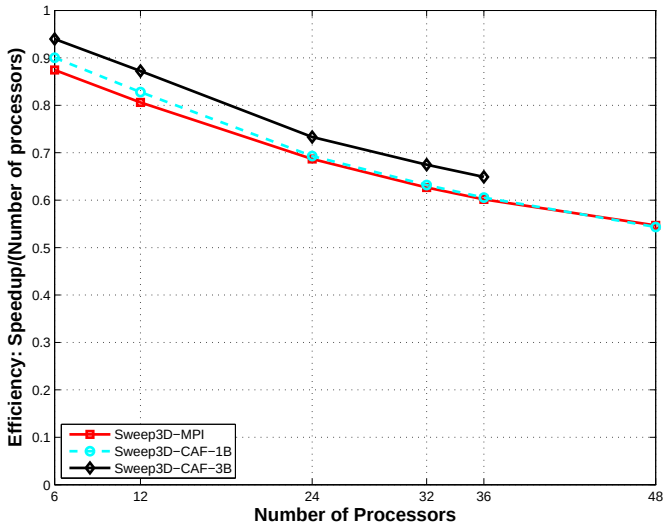


Figure 8.17 : Sweep3D of size 150x150x150 on an SGI Altix 3000.

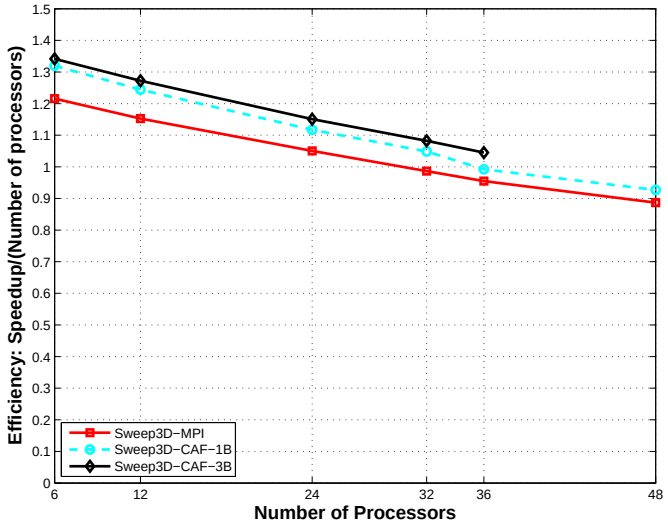


Figure 8.18 : Sweep3D of size 300x300x300 on an SGI Altix 3000.

MPI uses the rendezvous protocol and performs extra memory copying, demonstrating inferior performance comparing to the CAF versions. As expected, Sweep3D-3B shows the best performance for all problem sizes by communicating data in-place without extra memory copies and removing the anti-dependence synchronization off the critical path.

8.3 Language support for multi-version variables

To simplify development of high performance scientific codes with producer-consumer communication, we explore extending CAF with a new language construct called *multi-version variables* (MVVs). An MVV is declared similarly to a regular CAF variable using the `multiversion` attribute. As the name implies, an MVV can store several values; however, the program can read or write only one value — the *current version* — at a time. Values are managed with the semantics of a queue. Conceptually, each MVV has a queue that records the ordering of values in the MVV. A value is enqueued in the order in which it was committed to the MVV. Values are consumed (retrieved) in the order in which they were committed, which is ensured by the queue.

When a producer image `p` commits the first value into an MVV `mvv` located on consumer image `q`[3], we say that `p` establishes a *connection*. The connection state has several *receive* buffers to store unconsumed versions committed by `p` into the MVV `mvv` located on `q`. The ordering of committed values is ensured by `mvv`'s queue since several producers can concurrently commit new values (each commits into its own set of buffers), so the ordering is global rather than per producer. An MVV instance on a process image is guaranteed to store at least N unconsumed (pending) versions *per connection* (per *each* producer image that commits to the MVV). This means that N consecutive commits from the same producer are guaranteed to succeed without blocking the producer. The value of N is specified by the programmer. The default value of N is 1, meaning that there can be one unconsumed buffered value *per connection*, or one commit per producer will succeed without blocking. This suits well for most scientific applications that benefit from using MVVs. There is also a way to specify the number of versions for a particular MVV connection.

Producers commit values using the `commit` primitive and consumers retrieve values using the `retrieve` primitive. When a producer commits a new value and there is an

[3] `p` and `q` can be the same process image.

empty buffer, the value is written into this buffer and the producer continues execution. If there is no free buffer, the producer blocks until a buffer becomes available. When the consumer retrieves the next available version, the consumer's instance of the MVV is updated with the next value in the queue and the buffer occupied by the previous version is freed and made available for the corresponding producer. If there is no available value, the consumer blocks until a value is committed. The consumer can also test whether a new version is available without blocking. Note that producers and consumers can be different images or threads running within an image.

MVVs add limited support for buffered two-sided communication in CAF. The number of buffers is specified by the programmer; however, the run-time layer manages the buffer automatically. As for MPI, the right number of buffer is necessary for algorithmic correctness — to avoid the deadlock and to make progress (see Section 8.1) — and for performance. The `commit` and `retrieve` primitives combine data movement with synchronization. MVVs' two-sided nature makes them a convenient abstraction to express producer-consumer communication in scientific codes. Run-time management of additional buffers enables MVVs to achieve the same level of performance as that of the hand-optimized multi-buffer scheme (see Section 8.2.2), while the programmer is insulated from the details of buffer management, communication, and synchronization.

8.3.1 Declaration

An MVV variable is declared as a regular CAF variable that in addition uses the `multiversion` attribute. An MVV can be of an intrinsic Fortran 95 type or a user-defined type (UDT). If an MVV is of UDT `T`, type `T` or any of its nested UDTs cannot have allocatable, pointer, or multi-version components; in other words, type `T` must have statically known size. An MVV can be a scalar, an explicit shape array, an allocatable, or a component of a UDT `Q`. An MVV can be declared in a module. An MVV can be a subroutine argument and requires an explicit interface.

There are local and co-array (with brackets `[]`) MVVs. The local form simplifies the

```
1) type(T), save, multiversion :: a0, a1(10), ca0[*], ca2(10,10)[*]
2) type(T), allocatable, multiversion :: b0, b2(:,:), cb0[:], cb1(:)[:]
3) type(T), multiversion :: c0, c1(:), c11(N), cc0[*], cc1(:)[*], cc2(N,M)[*]
4) type Q
     type(T), multiversion :: v0
     type(T), multiversion :: v1(100)
     type(T), allocatable, multiversion :: y0
     type(T), allocatable, multiversion :: y2(:,:)
   end type Q
5) type (Q), save :: q0, q2(10,10), cq0[*], cq2(10,10)[*]
6) type (Q), allocatable :: r0, r2(:,:), cr0[:], cr2(:,:)[:]
```

Figure 8.19 : MVV declarations.

development of producer-consumer multithreaded codes when several threads of execution are enabled within one process image with distributed multithreading presented in Chapter 9. Co-array MVVs are used for inter-image producer-consumer communication; however, co-array PUT/GET access is prohibited; instead, `commit` and `retrieve` must be used. Figure 8.19 shows examples of MVV declarations of UDT `T`; the declarations of MVVs of intrinsic types are similar.

Example (1) declares SAVE scalar and explicit shape array and co-array MVVs; (2) declares allocatable array and co-array MVVs; declarations in (3) are used for subroutine parameters; (4) declares type `Q` with multi-versioned components; (5) declares SAVE scalar, explicit shape array, and co-array variables with multi-version components; (6) declares allocatable scalar, array, and co-array variables with multi-version components.

An MVV cannot be COMMON or sequence-associated because the semantics of sequence association are too complex and the feature would not add additional benefits. An MVV cannot have the TARGET or POINTER attributes to avoid aliasing that would complicate the semantics of MVVs. Similar to co-array subroutine parameters, MVVs can be passed in three different ways: using Fortran 77 convention, using Fortran 95 interface to pass an MVV as an MVV-object, or to pass only the current version. If a parameter MVV is passed as an MVV-object, it inherits the multi-version property in the subroutine; otherwise, the parameter becomes a regular Fortran 95 variable.

8.3.2 Operations with MVVs

Variable accesses.

Writes to an MVV `mvv` update the current version. Reads from `mvv` return the current version value. If an MVV is a co-array or a co-array component, it is illegal to remotely access it using the bracket notation. Instead of co-array PUT/GET specified via brackets (`[]`), programmers must use `commit` and `retrieve` to control versions. The restriction is a compile-time check. It avoids unpleasant side effects when a new version is retrieved, gives compiler and runtime more opportunities for optimization, and does not reduce the expressiveness.

Allocation and deallocation.

A local or UDT component MVV with the `allocatable` attribute can be allocated using the `allocate` statement, which is similar to allocation of Fortran 95 allocatable variables. Fortran 95 `deallocate` is used to deallocate the MVV. Note that allocation and deallocation are local operations and do not involve synchronization with other images.

A co-array MVV is allocated in the same way as co-arrays: the `allocate` call is collective among all images of the program and each image must specify the same shape for the MVV. Deallocation is done via the global collective `deallocate` call. Note that both `allocate` and `deallocate` have an implicit global barrier.

Committing values.

A new value can be committed into an MVV `mvv` by using the `commit` operator

```
commit(mvv, val, [live=true])
```

The `mvv` argument denotes an instance of an MVV that can be local, *e.g.*, `mvv` or `a%mvv`, or remote specified via the bracket syntax, *e.g.*, `mvv[p]` or `a[p]%mvv`. The remote `commit` commits value `val` into the instance of an MVV located on the target image

`p`. An MVV can be a scalar, array, or a UDT component, *e.g.*, `mvv[p]`, `mvv(:,:)[p]`, `a[p]%mvv(:)`, `a(i,k)[p]%mvv(:,:)`. The default co-shape of a co-array MVV is that of a co-array `a[*]`. Co-space objects described in Chapter 5 can be used to impose a topology on a co-array MVV.

The value being committed is taken from the `val` argument, whose element type must be the same as that of `mvv`. If `val` is a constant, the committed value is equal to `val` expanded to the size of `mvv`. If `val` is a variable, it can be a co-array, co-array component, Fortran 95 variable, local MVV, or local instance of a co-array MVV (note that a value cannot be read from an MVV instance located in a remote process image). If `val` is an MVV, the current value of the MVV `val` is committed. `val` can also be an expression, *e.g.*, a result of a computation or an array element/section of a variable.

The `live` parameter to `commit` is an optional argument that, if `.false.`, indicates that the value of `val` is not live in the process image performing the `commit` after the commit call. The `live` parameter helps to avoid an unnecessary data copy when a value is committed from an MVV, as explained in Section 8.6.

The order of versions committed into an MVV `mvv` is maintained on the consumer using `mvv`'s queue. If there is only one producer committing values, the values will be consumed in the same order in which they were committed. If several producers interleave `commit`s to `mvv`, two cases are possible. First, if producers do not synchronize between themselves, the order of values in `mvv`'s queue is determined by the hardware and run-time layer timing. If producers synchronize, the order of commits will be that enforced by their synchronization. We discuss these issues in more detail in Section 8.5.

Retrieving values.

A new version can be updated using the `retrieve` operator

```
retrieve(mvv, [var], [image], [ready])
```

The statement updates `mvv` with the next version in the queue. The previous version

value is lost. If there is no available next version, the `retrieve` operator blocks until a new value is committed. The MVV `mvv` must be a local instance; we also discuss extending the MVV concept with remote retrieves in Section 8.8. The optional parameter `var` can be supplied to copy the updated value form `mvv` into `var`; `var` must a local variable. This may save a line of code for the programmers who would have to perform the assignment explicitly. It also allows non-conformant assignments (see below) and provides additional information to the compiler and run-time layer. The optional parameter `image` serves two purposes. First, it is information for the run-time layer to expect a version from the producer image `p`. Second, it is useful for debugging to explicitly indicate that the next retrieved version must be committed from the producer image `image`. When the optional `LOGICAL` parameter `ready` is specified, `retrieve` sets it to `.true.` if there is a value available for retrieval, otherwise it is set to `.false.`; the `retrieve` does not block and does not retrieve a value.

To maintain simple and intuitive semantics of `commit` and `retrieve`, we purposefully limit versions that they accept to only *full-size* versions, whose size is the same as that of the MVV. Partial versions might be useful for some applications, but they complicate the semantics and robustness of the concept. However, we do not require the *shape* of committed and retrieved versions to coincide with the shape of the multi-version variable; only the size must be that of the MVV. The rationale is that some scientific codes such as NAS BT would benefit from non-conformant remote co-array assignments, *e.g.*, `a(1:N,1:M)[p] = b(1:M,1:N)`. In CAF, programmers would use Fortran 95 `RESHAPE` intrinsic or an auxiliary buffer to perform such an assignment.

Note that we do not provide a `retrieve` primitive that retrieves a remote version from another image. While it is possible to extend the concept with such remote retrievals, we have not seen a compelling case where it might be useful. For producer-consumer codes in a distributed environment, it is important to get data to the consumer as fast as possible. PUT or `commit` are better suited for this purpose because the producer can initiate the data transfer as soon as the value is produced and potentially overlap the communication

latency with local computation. GET or remote `retrieve` do not hide the data transfer latency unless optimized by a compiler, *e.g.*, split-phase GET. Such optimization, however, is difficult and unlikely to be effective in the general case.

Buffer tuning.

The default number of buffers to store unconsumed (pending) versions per connection (per each producer committing into the MVV) is defined during the compiler installation or program startup. Most scientific codes require buffering for only one unconsumed version *per connection* for both correctness and high performance. However, different producer-consumer communication patterns might require different number of buffers; moreover different images might require different number of buffers for the same co-array multi-version variable. We provide the ability to fine-tune the number of MVV versions by using `purge` and `mv_set_num_versions` functions.

`purge(mvv)` explicitly deallocates all unused send and unconsumed receive buffers. Note that for most scientific codes that can benefit from using MVVs, this should not be necessary. For example, in nearest-neighbor codes, each image communicates only with a small, fixed subset of neighbor images; therefore, the set of MVV buffers does not grow too large and stabilizes during execution.

`mv_set_num_versions(mvv,K,[image])` instructs the run-time to use at least `K` buffers per connection for an MVV `mvv` (a local instance). If the optional parameter `image` is present, then `K` is the number of receive buffers on `mvv`'s process image (consumer) to store unconsumed versions committed from the producer image `image`. `mv_set_num_versions(mvv,K,[image])` is not a collective call. In fact, the number of versions `K` that a co-array MVV can hold does not need to be the same on every image; it is a local property.

8.4 Examples of multi-version variable usage

In general, MVVs are a good abstraction for a large class of nearest-neighbor scientific codes where each image streams a sequence of values to a fixed set of neighbor images. This class of scientific applications includes wavefront (*e.g.*, Sweep3d), line-sweep (*e.g.*, NAS BT and SP), and loosely-coupled (*e.g.*, Jacobi iteration) codes. MVVs can also be used for intra-node producer-consumer codes if several threads of execution are allowed inside one image (see Chapter 9).

MVVs might not be the best abstraction for irregular codes (*e.g.*, RandomAccess, Spark98) where each image might communicate with all other images in the program, which might result in excessive MVV buffering and degraded performance. MVVs insulate programmers from managing the timing of the anti-dependence synchronization due to buffer reuse. However, if no such synchronization is necessary due to an application algorithm (*e.g.*, if the application can use fixed known number of communication buffers between each synchronization stage), this synchronization might introduce extra overhead. We believe that programmability benefits provided by MVVs in many cases outweigh slight performance loss due to the buffer reuse synchronization. Also, MVVs might not deliver the best performance if there is not enough local computation to overlap the anti-dependence synchronization with.

We now show several real application kernels that can benefit from using MVVs.

8.4.1 Sweep3D

The Sweep3D-3B and even Sweep3D-1B kernels shown in Figures 8.6 and 8.5, respectively, become much simpler when using multi-version `Phiib` and `Phijb` variables as shown on Figure 8.20. When using MVVs, the kernel is simple and intuitive and looks very much like that of the MPI version. At the same time, it can deliver comparable or better performance as shown in Section 8.7.

```
...
if (receiving from I_pred) then
   retrieve(Phiib)
endif
!  similar for the J-dimension
...
!  computation that uses and updates Phiib and Phijb
...
if (sending to I_succ) then
   commit(Phiib[I_succ], Phiib)
endif
!  similar for the J-dimension
...
```

Figure 8.20 : Sweep3D kernel pseudocode with multi-version buffers.

8.4.2 NAS SP and BT forward [xyz]-sweeps

Figure 8.21 shows how the forward sweep along spatial dimension x in NAS SP can be expressed via MVVs; again, the code is very similar to that of two-sided MPI. Sweeps in y- and z-dimensions have similar communication. NAS BT forward sweeps also have similar communication structure.

```
do stage = 1, ncells
   ...
   !  receive the next xf_buff from the x-predecessor in the sweep
   if (stage .ne.  1) then !  first stage
     retrieve(xf_buff)
   endif
   ...
   !  computation that uses values of xf_buff out-of-buffer
   ...
   !  pack xf_buff to send to the x-successor in the sweep
   !  send xf_buff to the x-successor
   if (stage .ne.  ncells) then !  last stage
     commit(xf_buff[x_succ], xf_buff)
   endif
   ...
done
```

Figure 8.21 : NAS SP pseudocode for forward sweep along x dimension expressed via MVVs.

```
do stage = 1, ncells
    ...
    !  receive the next xf_recv_buff(stage) from the x-predecessor
    if (stage .ne.  1) then !  first stage
      call wait(x_pred)
    endif
    ...
    !  computation that uses values of xf_recv_buff(stage) out-of-buffer
    ...
    !  pack xf_send_buff(stage) to send to the x-successor in the sweep
    !  complete previous non-blocking PUT region
    if (stage .ne.  1) then
      complete non-blocking PUT region with index stage-1
    endif
    !  transfer xf_send_buff(stage) to the x-successor
    if (stage .ne.  ncells) then !  last stage
      start non-blocking PUT region with index stage
      xf_recv_buff(...,stage+1)[x_succ] = xf_send_buff(...,stage)
      stop non-blocking PUT region with index stage
      !  notify the x-successor that the buffer has been updated
      call notify(x_succ)
    endif
    ...
done
```

Figure 8.22 : NAS SP pseudocode for forward sweep along x dimension in CAF that uses a buffer per stage.

If the kernel is coded using PUT/GET, the user has to also insert point-to-point synchronization statements, manage several communication buffers, and non-blocking PUT directives to obtain high performance. For instance, it is possible to use a separate communication buffer per stage to avoid the synchronization due to buffer anti-dependence, as shown in Figure 8.22. This is relatively simple kernel to program; however, excessive buffering may increase cache pressure when buffers are large. Alternatively, the three-buffer scheme can be used similar to how it is done for the Sweep3D kernel in Figure 8.6, but to achieve high performance, the programmers would have to code complex synchronization (more complex than shown in Figure 8.22). With MVVs, the synchronization and buffering are hidden from the programmer inside the `commit` and `retrieve` primitives.

8.4.3 NAS SP and BT backward [xyz]-substitutions

Figure 8.23 presents pseudocode for the x-dimension backward substitution stage in NAS BT. Substitutions for y- and z-dimensions are similar. As with the forward sweeps, using MVVs simplifies coding of the backward substitution stages in NAS BT and ST, while also delivering high performance. In addition, this example also demonstrates how MVVs can reduce programmers' effort for writing packing/unpacking code.

```
do stage = ncells, 1, -1
    ...
    if (stage .ne.  ncells) then !  first stage
       retrieve(xb_buff, backsub_info(0:JMAX-1,0:KMAX-1,1:BLOCK_SIZE,cell))
    endif
    ...
    !  intense computation
    ...
    if (stage .ne.  1) then !  last stage
       commit(xb_buff[x_pred], rhs(1:BLOCK_SIZE,0,0:JMAX-1,0:KMAX-1,cell))
    endif
    ...
done
```

Figure 8.23 : NAS BT pseudocode for backward substitution in x dimension.

Using an MVV enables to "reshape" communicated data without making programmers use an auxiliary communication buffer and write the packing/unpacking code in the case when communication could be expressed as an assignment of two non-conformant co-array sections. For example, the following code would be a natural way to express in-place communication in NAS BT, but it is illegal in CAF because the shapes of `rhs` and `backsub_info` references are non-conformant

```
backsub_info(0:JMAX-1,0:KMAX-1,1:BLOCK_SIZE,remote_cell)[x_pred] =
   rhs(1:BLOCK_SIZE,0,0:JMAX-1,0:KMAX-1,cell)}
```

The intent of the assignment is to transfer elements as shown in Figure 8.24. The code would perform fine-grain element accesses and must be optimized by the compiler. It is also possible to use Fortran 95 `RESHAPE` intrinsic.

The sizes of the `rhs` and `backsub_info` sections are the same. The CAF version uses a contiguous 1D buffer `send_buf` to pack data from `rhs` at the source, transfer

```
do j = 0, JMAX-1
    do k = 0, KMAX-1
       do b = 1, BLOCK_SIZE
          backsub_info(j,k,b,remote_cell)[x_pred] = rhs(b,0,j,k,cell)
       enddo
    enddo
enddo
```

Figure 8.24 : Data transfer in x-dimension backward substitution of the NAS BT benchmark.

`send_buf` into a 1D contiguous co-array buffer `recv_buf`, and unpack `recv_buf` into `backsub_info` on the destination. It requires declaring the auxiliary communication buffers and writing packing/unpacking code. Using MVVs relieves programmers from both.

The MVV `xb_buff` for the x-dimension is declared as a 1D allocatable array

```
double precision,allocatable,multiversion::xb_buff(:)
```

The packing/unpacking is done by the run-time layer inside the `commit` and `retrieve` primitives. MVV packing/unpacking code enumerates all elements of a section as a DO-loop nest that accesses the elements in column-major order, which is a typical case. If an alternative element enumeration is desired, the programmer can use DO loops and pack/unpack variables into/from the current version of an MVV.

In addition, all data transfers done by the MVV run-time layer are contiguous. Experiments show [47, 48, 33] that source-level packing/unpacking is necessary to achieve the best communication efficiency for strided transfers on some interconnect. Thus, MVVs naturally take care of user-level packing/unpacking that programmers would have to do manually to compensate for inefficient support for strided communication in most interconnects and one-sided communication libraries.

Finally, with MVVs, the programmers use *local* array subscripts, which is easier than tracking *remote* co-array subscripts of a PUT/GET. In the example above, the programmer needs to know the indices of the remote co-array section `backsub_info`, *e.g.*,

`remote_cell`, which is not equal to the local cell number `cell`. In some cases, maintaining the information about remote co-array shapes locally was a major inconvenience that we noticed during our evaluation studies. When using MVVs, `commit` and `retrieve` use the local value of `cell`.

8.5 Relation of MVVs, GETs/PUTs, and synchronization

The semantics of the `commit` and `retrieve` primitives are not those of PUT/GET. There can be several unconsumed buffered versions in an MVV that are "invisible" to the programmers. The next value in the sequence will become visible only when the consumer executes `retrieve`; in this respect, it is not intuitive to associate PUT with `commit` and GET with `retrieve`. Synchronization statements provide certain guarantees for GET/PUT completion and ordering. However, buffered versions in an MVV are retrieved explicitly by the consumer image and may even be retrieved after several synchronization events between the producer and consumer. This makes it impossible to provide programming model guarantees for the ordering between observing the results of `commit`s (available via `retrieve`s), and PUT/GET and synchronization.

On the other hand, `commit` has more intuitive relation to PUTs/GETs and synchronization; note that `commit` does not make the value visible, only `retrieve` can do it. Since `commit` implies synchronization (similar to unidirectional point-to-point notification) with the `commit`'s target image, PUTs/GETs that were issued before must complete according to the memory consistency model described in Section 3.1.6. The default case is to complete only prior PUTs/GETs issued to the target process image before the committed version is made available for retrieval on the target image; this is equivalent to the semantics of the weaker `notify`. The optional parameter `mode=strict` can be passed to `commit` to complete *all* prior PUTs/GETs. Intuitively, if the programmer uses MVVs to implement synchronization between two images, the semantics of `commit` and `retrieve` are those of `notify` and `wait`: the effects of PUT/GET communication prior to `commit` must be visible to `p` when the value is retrieved.

What is more important for MVVs is the ordering of committed versions. In the case of a single producer, the version ordering is that of a stream. If image `p` executes two `commit`s, the value of the first `commit` is always retrieved before the value of the second. The case of multiple producers is more subtle. If two images `p` and `q` commit values into an instance of an MVV located on image `r`, two cases are possible. First, if there was no synchronization between `p` and `q`'s `commit`s, the order of versions is undefined and determined by the timing in the hardware and run-time layer implementation. Second, if there was a synchronization event, *e.g.*, a barrier or a notify/wait pair that ordered the execution of `commit`s, the value of the first `commit` is guaranteed to be retrieved before the value of the second. These semantics are intuitive with respect to the meaning of synchronization.

8.6 Implementation

We describe a prototype implementation of MVVs based on Active Messages (AM) [122]. If AMs are not available on the target platform, the run-time layer implementation of `commit` and `retrieve` primitives can poll the network emulating AMs. The implementation is conceptually similar to that of the multi-buffer scheme described in Section 8.1; however, it does not use CAF's notify/wait primitives.

8.6.1 An implementation based on Active Messages

Initially, an MVV has buffer only for one version — the current version that can be accessed locally via read and write. When producer image `p` commits the first value to an instance of an MVV located on image `q`, it sends an AM to establish a connection. The AM allocates N receive buffers on `q`, where N is the default number of buffers per connection. These buffers are used to accept values committed from `p`, so that N commits from `p` will succeed without blocking `p`. The reply AM makes the buffer addresses available on `p`. They can be used later to communicate versions from `p` to `q`, *e.g.*, using RDMA, without the need to contact `q`. The connection can also be established by `retrieve` if

the optional parameter `image` is specified and the consumer image executes `retrieve` before the first commit from image `image`. Note that the connection is established without programmer's involvement.

For an MVV `mvv`, assuming that the connection has been established, the `commit` implementation must transfer and enqueue the committed value into `mvv`. Each `mvv` has a queue that records the global order of commits from several producers. The producer knows locally whether there is an available receive buffer on the consumer since the fact that a version committed by the producer has been consumed, thus, one of the corresponding receive buffers becomes free, is communicated to the producer in the implementation of `retrieve` (see below). If there is no available receive buffer, the producer is blocked and waits for a buffer to become available. When there is an available receive buffer, the producer can transfer the value and enqueue it in two ways; which one should be used depends on the version size, communication substrate, and the AM implementation.

In the case of large data size, the value is transferred into the receive buffer, *e.g.*, using RDMA PUT. Next, the producer sends a synchronization AM that enqueues the version on the consumer. In the case of small data size, the value can be transferred with the synchronization AM. This AM enqueues the next MVV version and must copy the value from the AM ephemeral payload memory into the MVV receive buffer, from where it will be retrieved later. An implementation uses an auxiliary send buffer(s) to make data movements non-blocking to overlap it with computation on the producer.

The `retrieve` implementation is straightforward. If there is no available version in the queue to retrieve, `retrieve` blocks[4] and waits for a version to become available. Otherwise, `retrieve` makes the next enqueued value the current version, available for access in the program; the previous version is lost. Next, the run-time layer sends a synchronization AM to the producer containing the address of the free receive buffer indicating that the buffer is ready to accept a new version; for efficient implementation, this message should be non-blocking.

[4] `retrieve` can also poll the network.

The data copy on the consumer is not necessary if the MVV's current version is represented via F90 pointer. Instead of copying the data, the implementation adjusts the address of the F90 array descriptor to point to the buffer containing the next version data. Similarly, if the producer commits a value to an MVV `mvv1` from another MVV `mvv2` and `mvv2`'s current version will be dead after the commit, `mvv2`'s current version F90 pointer is adjusted to point to a free send buffer of `mvv2`; the former current version buffer of `mvv2` becomes a send buffer. Liveness analysis or `commit`'s optional argument `live` set to `.false.` can determine when the committed value is not live.

We now describe several possible extensions to our prototype implementation. The requirement for MVVs to make progress is to have at least N receive buffers per connection. If there is not enough memory for buffering, this is a critical run-time error. However, the run-time layer can implement a smarter buffering scheme rather than aborting the program. For example, if there is an open connection with some producer that has not been active recently and none of the buffers is used, the connection can be closed freeing some memory. Similarly, some unoccupied buffers from open connections can be deallocated freeing the memory. If there are no buffers to deallocate and more memory is needed, the program should be aborted with the resource limit reached message. Note that scientific applications that can benefit from MVVs should not experience excessive buffering.

Another type of adaptation policy is actually to increase the number of buffers per connection. Having more buffers enables to send the buffer-free synchronization AM earlier and remove it from the critical path. Thus, if `commit` does not have a free receive buffer, *e.g.,* because there is not enough computation on the producer to overlap the anti-dependence synchronization latency with, the run-time layer may allocate an additional receive buffer(s) in the hope to hide this latency. The extra buffering should be done carefully not to cause a memory shortage. A good heuristic would be to limit the maximum memory usage per connection. This would enable many versions for small size MVVs — exactly what is necessary to hide the buffer reuse synchronization latency without excessive memory usage.

8.6.2 Prototype implementation in `cafc`

We designed and implemented a prototype compiler and run-time support for MVVs in `cafc`. The runtime uses Pthreads [82] and ARMCI with AMs. `cafc` translator was extended to accept a subset of the MVV specification. This subset includes allocatable MVVs, `allocate`, `commit`, and `retrieve` primitives, which enabled us to compile and evaluate the performance of several parallel codes such as Sweep3D and the NAS SP and BT benchmarks expressed via MVVs. The support for the `multiversion` attribute was not implemented in the Open64/SL front-end; instead an MVV is identified by the `mv_` identifier prefix. The prototype represents an MVV using a F90 pointer and an opaque handle to store the run-time state; this is similar to co-array representation. The implementation allows a fixed number of unconsumed versions per connection. We have yet to find real scientific codes for which smarter buffer management would yield benefits and can justify the development effort. The implementation uses F90 pointer adjustment to reduce memory copies. It also uses non-blocking AMs or PUTs for top efficiency wherever possible. When a version is committed from a large MVV, the prototype uses an additional send buffer(s) to enable non-blocking RDMA PUT. If the `live` argument to `commit` is `.false.`, F90 pointer adjustment is also used on the producer to avoid extra memory copies.

8.7 Experimental evaluation

We used MVVs to implement kernels of three benchmarks: Sweep3D and the dimensional forward and backward sweeps in the NAS BT and SP benchmarks (`[xyz]_solve` subroutines). Using MVVs resulted in much cleaner code compared to the best hand-coded variants in CAF because `commit` and `retrieve` encapsulate the buffering, point-to-point synchronization, and non-blocking communication, which the programmer would otherwise have to code explicitly. The performance of the three codes was measured on an Itanium2 cluster with a Myrinet 2000 interconnect (RTC), described in Section 3.3. It is the only platform where both `cafc` and ARMCI with AM support are available.

8.7.1 Sweep3D

We expected the performance of MVV versions to be comparable to that of the hand-coded multi-buffer one and, therefore, better than that of the MPI version for the cases when the multi-buffer version outperforms MPI. Compared to the hand-optimized multi-buffer, an implementation of the MVV abstraction adds extra overhead. The sources of overhead include extra messages to communicate the control information, higher memory requirements for buffering, and extra memory copies. In hand-optimized programs, such overheads might be avoided because of specific application-level information. For example, hand-coded Sweep3D-3B uses three buffers per communication axis (I/J). When the sweep changes the direction, it changes communication partners, swapping predecessors and successors. Because of the ordering guaranteed by the wavefront, the programmer knows that it is algorithmically safe to reuse the same set of buffers since the former communication partner must have consumed the prior version and its buffers are free and ready to accept new data. However, the MVV run-time does not have such knowledge and uses five buffers per communication axis: the current version buffer and a pair of send and receive buffers per communication direction. It is possible to reduce this number to four by using only one send buffer for both directions because the decision whether to reuse a send buffer is purely local and does not involve synchronization with other images (as opposed to the reuse of receive buffers). Note that Sweep3D and NAS BT and SP communicate several times in the same direction (using the same receive buffer), thus, the increased cache footprint is amortized and might affect the application only when the communication direction changes.

Figures 8.13, 8.14 and 8.15 (see Section 8.2.2) present the parallel efficiency of Sweep3D of the 50x50x50, 150x150x150 and 300x300x300 problem sizes. Sweep3D-3B represents the performance of the multi-buffer version of Sweep3D, the fastest available hand-optimized parallelization. The MVV-based version shows roughly identical performance to that of the Sweep3D-3B version for the 150x150x150 and 300x300x300 problem sizes. It slightly (less than 0.5%) outperforms Sweep3D-3B for the small 50x50x50

problem size because it uses non-blocking AM-based synchronization messages, while the current version of the ARMCI library does not provide truly non-blocking `notify` used in Sweep3D-3B. ARMCI's notify executes a fence before sending the non-blocking notification PUT. Because the `notify` follows the non-blocking PUT immediately in Sweep3D-3B, the PUT essentially becomes blocking, leading to a slight performance degradation for the small problem size. The effect becomes marginal for the larger 150x150x150 and 300x300x300 problem sizes because the communication to computation ratio decreases. Note that both MVV-based and Sweep3D-3B versions show comparable or better performance than that of the MPI version (see discussion in Section 8.2.2).

The MVV-based version uses F90 pointer adjustment on both producers and consumers to avoid extra memory copies and specifies the optional `image` argument to `retrieve`. We also evaluated three other MVV-based versions to measure a potential performance loss due to extra memory copies. The first version performed a data copy only on the producer from the current version buffer into the send buffer. The second version performed a data copy from the receive buffer into the current version buffer on the consumer. The third copied data on both producer and consumer. All MVV-based versions performed roughly identical, and performance fluctuations were statistically insignificant in our experiments; however, extra memory copies may degrade the performance of some codes.

In addition, we evaluated the effect when a consumer, rather than a producer, establishes the connection. This may happen when the consumer executes a `retrieve` with the `image` argument before the producer performs the first `commit`, and has the effect of removing the connection establishment latency from the critical path. For regular codes, the gain is amortized over many communications between the same producer and consumer, thus, knowing the origin of communication has negligible effect.

8.7.2 NAS BT and SP

Figures 8.25, 8.26, and 8.27 present parallel efficiency for executions of NAS BT of A (64^3), B (102^3), and C (162^3) classes, respectively. The results were obtained on an Ita-

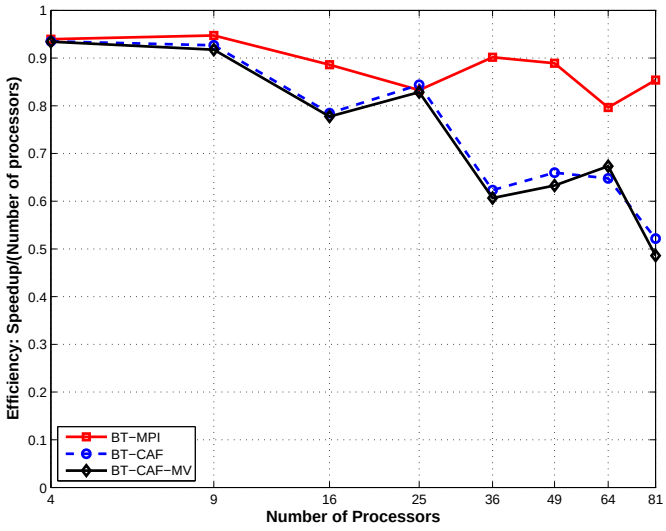


Figure 8.25 : NAS BT class A on an Itanium2 cluster with a Myrinet 2000 interconnect.

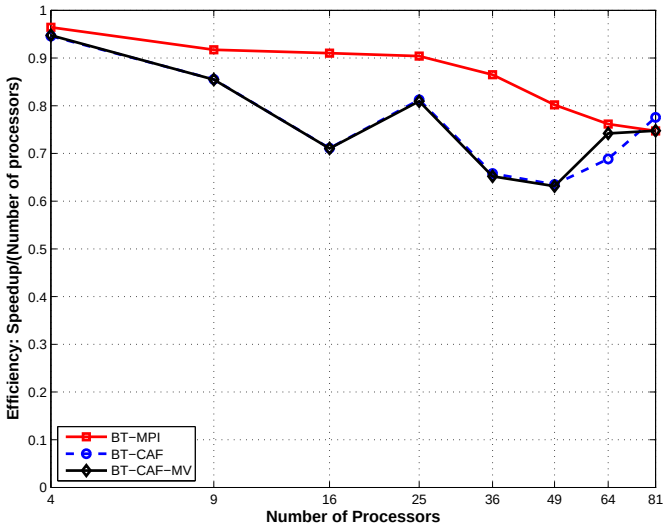


Figure 8.26 : NAS BT class B on an Itanium2 cluster with a Myrinet 2000 interconnect.

nium2 cluster with a Myrinet 2000 interconnect (RTC). BT-MPI stands for the efficiency of the standard MPI version (see Section 3.4). BT-CAF is the efficiency of the best hand-coded CAF version, which uses a different buffer for each stage of the forward sweeps and backward substitutions, thus, avoiding the anti-dependence synchronization due to buffer

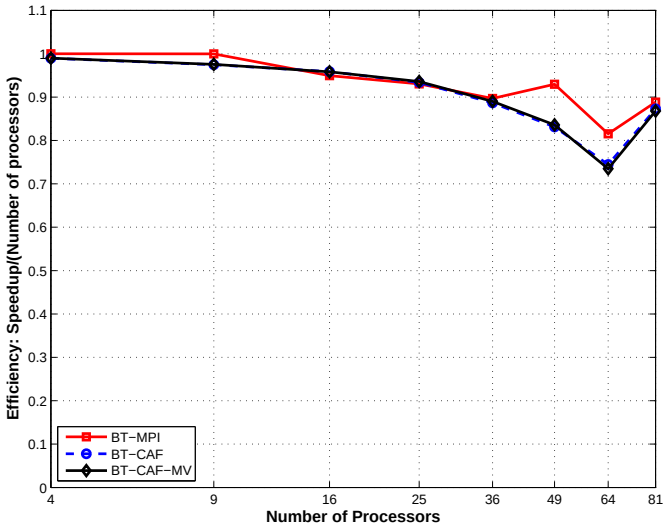


Figure 8.27 : NAS BT class C on an Itanium2 cluster with a Myrinet 2000 interconnect.

reuse altogether. BT-CAF-MV is the efficiency of the version that uses MVVs to implement the forward sweeps and backward substitutions. BT-CAF-MV uses two MVVs per dimension: one to communicate LHS border regions in the forward sweep, the other to communicate RHS borders in forward and backward sweeps.

For all problem sizes, the performance of BT-CAF-MV is roughly equal to that of BT-CAF because (1) the latency of the anti-dependence synchronization in the BT-CAF-MV version is overlapped with computation and does not contribute to the critical path, and (2) even though AM-based non-blocking synchronization in the BT-CAF-MV is more efficient than that of the `notify` implementation, the effect is minor due to significant local computation. The performance of the MPI version is somewhat better than that of both CAF versions[5].

Figures 8.28, 8.29, and 8.30 show parallel efficiency for executions of NAS SP of A

[5]For earlier RTC configuration, the performance of BT-CAF and BT-MPI was almost the same. BT-CAF version, and as a consequence, BT-CAF-MV, showed worse performance after a recent series of RTC software updates. For the purposes of this discussion, comparing the performance of the CAF versions to that of MPI is not that important.

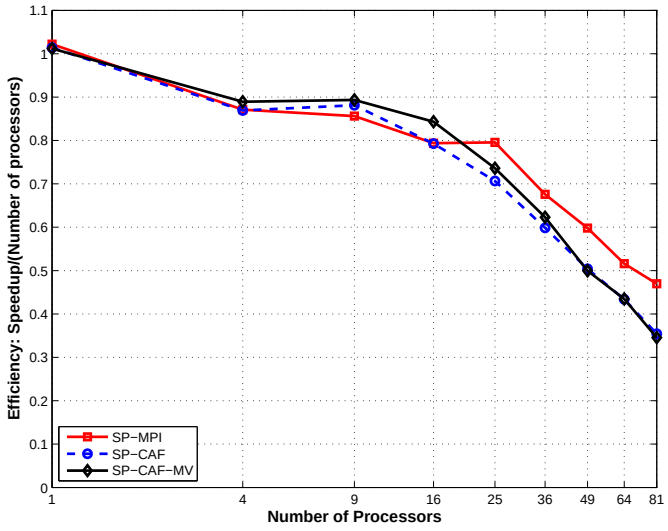


Figure 8.28 : NAS SP class A on an Itanium2 cluster with a Myrinet 2000 interconnect.

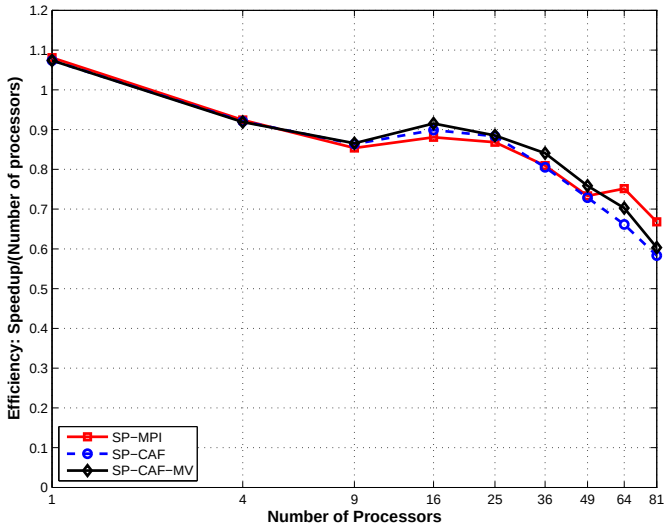


Figure 8.29 : NAS SP class B on an Itanium2 cluster with a Myrinet 2000 interconnect.

(64^3), B (102^3), and C (162^3) classes, respectively. SP-MPI, SP-CAF, and SP-CAF-MV correspond to parallel efficiency of the standard MPI, best hand-optimized CAF, and MVV-based versions. The structure of the forward sweeps and backward substitutions is similar to that of the NAS BT versions. SP-CAF-MV uses one MVV per dimension that replaces

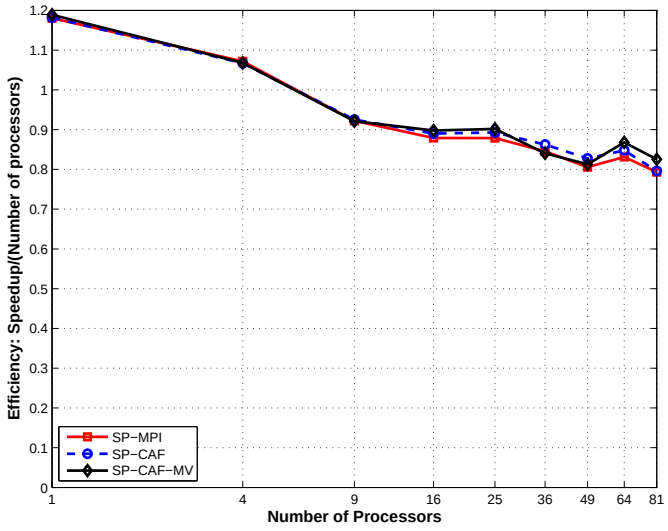


Figure 8.30 : NAS SP class C on an Itanium2 cluster with a Myrinet 2000 interconnect.

the buffer co-array used for communication in forward and backward `[xyz]_solve` sweeps. The `commit` and `retrieve` primitives remove the point-to-point synchronization and non-blocking communication directives from the code as well as allow using the local subscripts (cell indices) to specify parameters of communication (see Section 8.4.3), making the code simpler and more intuitive.

The performance of the SP-CAF-MV version is somewhat better than that of the best hand-coded SP-CAF version for classes A and B because SP-CAF-MV's synchronization messages are non-blocking. For the large, class C, problem size, computation dominates this slight difference, so SP-CAF and SP-CAF-MV have roughly similar performance.

8.8 Discussion

There are many options for simplifying producer-consumer communication in CAF. One can simply use MPI, which is designed for general two-sided communication. MPI might introduce extra memory overhead (copies, registration/unregistration) and, if it uses the rendezvous protocol, expose communication latency, resulting in suboptimal

performance. MVVs essentially encapsulate the multi-buffer communication scheme and can deliver higher performance than MPI. With MVVs, the producer always knows the destination receive buffer on the consumer and using F90 pointer adjustment can avoid extra memory copies, transferring data in-place. Alternatively, the abstraction of a stream/link/channel/pipe is also a good candidate for producer-consumer communication; however, it requires an explicit connection and is likely to exhibit extra memory copies because the destination memory of communication is not known until the value is read from the stream.

MVVs bear resemblance to the Clocked Final (CF) model [106], but programmers are more involved in the process of controlling versions when developing codes using MVVs. First, the number of MVV versions per connection is explicitly defined by the programmer who specifies the minimum number of versions necessary for the application to make progress or for obtaining the best performance. Second, `commit` and `retrieve` explicitly define and control versions. While the CF model guarantees determinacy and deadlock freedom and is a more convenient abstraction for scientific applications, there is not yet a parallel implementation of CF; thus, it is not yet clear whether CF can deliver high performance. In contrast, MVVs are capable of delivering high performance today for a large class of scientific applications with producer-consumer communication, while offering a much simpler programming style compared to CAF without them.

There are several reasons that influenced our decision to make MVVs a language-level rather than library-based abstraction. First, the `multiversion` attribute fits well into the existing type system. The compiler can check the correctness of MVV parameter passing to the `commit` and `retrieve` primitives. The compiler can also use an opaque handle to represent the run-time state of an MVV freeing the programmer from declaring and passing around a special object. Second, the compiler can ensure that accesses to remote MVVs using the bracket notation are prohibited to avoid unexpected side effects when versions are retrieved. Third, `commit` and `retrieve` primitives can be used for MVVs of any user-defined type; a library-based implementation does not permit such

overloading in Fortran 95. Fourth, the compiler and run-time can establish connections on demand, without programmer's involvement, and use F90 pointer adjustment to avoid unnecessary memory copies because the compiler and runtime control how MVV memory is allocated/deallocated. Achieving the same in a library-based implementation would require programmers allocate/deallocate memory specially. To summarize, MVVs are more convenient and expose more information to the compiler and run-time as a language-level abstraction.

Expressing high performance producer-consumer communication in PGAS languages is difficult. MVVs are a language-level abstraction for CAF that both improves programmability and provides more information to the compiler and run-time, which can tailor it to a particular architecture to deliver the best performance. Our research showed that MVVs are applicable to a large set of scientific codes that include wavefront and line-sweep applications; they can also be used in loosely-synchronous applications. MVVs significantly simplify development of wavefront applications, such as Sweep3D, and deliver performance comparable to that of the fastest CAF hand-optimized versions and comparable to or better than that of MPI-based counterparts, especially if MPI uses rendezvous protocol for send/receive communication, on a range of parallel architectures. We counted extra lines of code (LOC) necessary to implement CAF versions of Sweep3D compared to the MPI version. For Sweep3D-1B, this number is 12. For Sweep3D-3B, this number is 70. The MVV-based version has the same communication style as the MPI version and does not introduce extra lines. It is, however, questionable whether LOC is a good metric to estimate programmability gains. The total Sweep3D LOC is 2182; the number of LOC for buffer management, communication, and synchronization is small compared to the computational LOC. Nevertheless, our feeling is that implementing either MPI or MVV-based version is much simple than implementing the Sweep3D-3B version; because it is not the number of lines that matters, but the complexity of reasoning about communication/synchronization — where to insert these lines and how difficult it is to debug the program. Programming Sweep3D-3B is much harder than any other Sweep3D version because the programmer is

responsible for orchestrating the anti-dependence synchronization pipeline (start-up, steady state, and wind-down code), managing buffers in a circular fashion, and using non-blocking PUT directives; this also requires relating events from different loop iterations. Both MPI and MVV-based versions hide this complexity inside the run-time layer. A good analogy here is that programmers should not software pipeline a short loop critical for performance by hand even though it would increase the code size by only a few lines. Also, MVVs greatly simplify coding of line-sweep applications, such as the NAS BT and SP benchmarks, and deliver performance comparable to that of the best hand-optimized MPI and CAF counterparts. Based on our research, we believe that MVVs are a promising extension to CAF.

As scientific community gains more experience using MVVs, it would also be interesting to consider whether MVVs can benefit from extensions such as GET-style remote `retrieve`, commits and retrieves for partial versions, and an adaptive buffer management strategy.

Chapter 9

Toward Distributed Multithreading in Co-array Fortran

The success of a new programming language is imposed by its ubiquitous availability, acceptable programmability, and its ability to deliver high performance on a wide range of computation platforms. The growing popularity of hybrid cluster architectures with multi-core multiprocessor nodes creates a demand for CAF with explicit language-level multithreading that enables one to co-locate computation with data and to exploit hardware threads. This chapter presents *co-functions* (CFs) and *co-subroutines* (CSs), language constructs to support distributed multithreading (DMT) in CAF.

9.1 Motivation

Currently, CAF is an SPMD programming language with only one thread of computation per process image, as shown in Figure 9.1. Under this limitation, even the simple task of efficiently finding the maximum value of a co-array section that is located in a remote image memory and can be concurrently accessed by several process images is problematic. The local image must obtain exclusive access to the remote co-array section, fetch it over the network, and find the maximum (or worse, use remote element-wise accesses to find the maximum if the communication is not vectorized) and, finally, release exclusive access. This code would have very low performance. Alternatively, a computation can be shipped to be co-located with the data that it accesses. On behalf of a requesting image, the remote processor can acquire exclusive access to the co-array section locally, efficiently find the maximum among its local values, release exclusive access and send the result to the requesting processor. However, with only one thread of computation per image, the logic of the program would be much more complicated than it might be. To perform a computation,

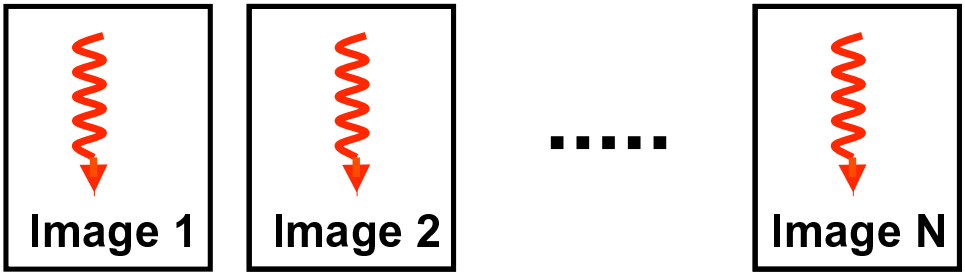


Figure 9.1 : Execution model of classical SPMD CAF.

the remote thread would have to interrupt its own computation and poll for remote requests. Moreover, the question of how often to poll is a difficult problem in itself.

The code to find a remote maximum can be significantly simplified if several threads are available for every image and the language permits one to *spawn* a remote *computation* (or *activity* or *thread*; the three are used interchangeably hereafter). Computations spawned remotely have affinity to the remote memory and enable one to cope with the interconnect latencies for accessing remote data inherent in cluster and NUMA architectures. The current image can spawn a remote activity without changing the logic of the "main" remote thread that would be solely responsible for computation and not for servicing remote requests. While compilers could identify some pieces of code that could be shipped closer to data they access, it is unlikely that the compilers would be able to detect all such computations.

Examples of such data structures are linked lists, trees, graphs, queues, etc.; their parts are located in remote memory. Such data structures are commonly used in parallel search applications and often require complex, *multi-step* accesses due to pointer dereferencing. Such accesses can be compiled into efficient code for parts of data structures located in a node's local memory. However, if a multi-step access is performed to a part of data structure located in another node's memory, the number of network transactions required is typically proportional to the number of dereferencing steps; this is particularly inefficient on cluster architectures. Instead of accessing remote data through multiple levels of indirection over the network, it would be better to declare the code corresponding to a remote access as

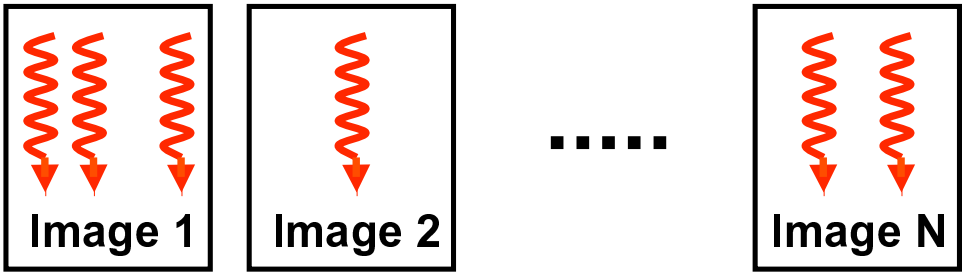


Figure 9.2 : Execution model of CAF with distributed multithreading.

a subroutine and execute it in the remote image address space. In essence, this converts remote data accesses into local accesses since code will now be local to the data structure.

This is another example demonstrating a broader concept of computation shipping. While a compiler could detect some remote accesses and without user intervention "ship" them to the remote processor, it would not be able to detect and optimize all such accesses. Extending CAF with language-level *blocking* and *non-blocking* **co-function** and **co-subroutine** remote calls, analogous to classical remote procedure calls, enhances the expressiveness of the language, makes it easier to use for some codes, and improves performance of codes that heavily manipulate data structures located in a remote memory, as we show in this chapter.

If a computation can be spawned remotely, there is little difficulty in supporting spawning a computation locally as well. Adding the ability to spawn another computation or activity locally makes CAF a multithreaded language, as shown in Figure 9.2. In fact, not only does local multithreading remove a semantic inconsistency, but also it provides an efficient mechanism for fully exploiting multi-core multiprocessor nodes. If several computations within a process image are independent, they can be executed concurrently, utilizing all available processing power of the node, without the need to create an image per CPU core.

The language-level distributed multithreading conceptually changes CAF's execution model. Under this model, co-arrays should be considered as a convenient shortcut to "passively" access remote data. Each image would represent a *locality domain* and physically

own a portion of partitioned global address space. Each image might have several threads executing concurrently. From the language point of view, there is no distinction between computations initiated by local or remote threads. Any thread can spawn or initiate another thread in the local image or in a different remote image. This more general execution model enhances CAF's expressiveness and simplifies programming of certain application classes such as parallel search problems.

Distributed multithreading requires mechanisms to synchronize threads. CAF's critical section [86] is an inadequate, inflexible abstraction. We believe that using *locks* and *condition variables* is a better way to synchronize threads within a process image. For inter-image synchronization, DMT can use tagged barriers, tagged point-to-point synchronization, or eventcounts [99][1].

Below are a few detailed examples that further motivate and clarify the distributed multithreaded execution model of CAF.

9.1.1 Accesses to remote data structures

Without the distributed multithreading support, it is not possible to implement, say, a local linked list in CAF that can be efficiently accessed by remote images on a cluster. Remote insertion or deletion of an element at a certain position requires the ability to search the list efficiently. The remote search might require many network transactions, one for each remote list element. On the other hand, each of these operations can be coded as a subroutine that is spawned inside the image that owns the remote linked list. With this approach, remote insertion, deletion, or search requires only two network messages: to initiate a request and to return the result. This is a significant improvement over the classical solution in CAF. Similar arguments apply to other commonly used data structures that must be accessed efficiently by remote images.

[1] Eventcounts are explored by Cristian Coarfa in his companion thesis [29].

9.1.2 One-sided master-slave programming style

To efficiently utilize all processors, many parallel search algorithms require a fair division of work among processors; this usually amounts to a fair division of an enormous search space. A typical example is the traveling salesman problem (TSP) [104]. In a branch-and-bound TSP implementation, a tree-like search space is huge and the best known cycle length should be used to prune the search space even if the search is done on a massively parallel machine. Pruning creates the necessity for load balancing the processors since some of them might run out of work faster then the others. The problem can be solved using a master-slave paradigm. The master manages the distribution of the search space and the value of the best path length; the clients update and query the master for the best length and ask for more work. This scheme requires a dedicated process to maintain the state and to service requests, which can become a bottleneck on parallel architectures with very large number of processors such as Blue Gene/L [56].

DMT CAF enables an efficient and natural implementation without a dedicated master process. When a processor runs out of work, it grabs a portion of unexplored search space from a different processor. In a simple case, a repository of search space states (*e.g.*, “available for search”/”already explored”/”being explored”) is stored on a single processor, called the *repository processor* R. Other processors spawn remote activities on P to obtain parts of the search space. This could be viewed as a one-sided analog to the classical two-sided master-slave approach. The programmer does not need to structure the computation of the repository processor to periodically poll for and service remote requests; the run-time system takes care of that. The repository processor can also perform useful work by exploring a part of the search space. If the centralized repository becomes a bottleneck, the state can be distributed across several process images. Clearly, distributed multithreading would enable a much simpler implementation of a distributed repository solution compared to that in classical CAF.

Many applications in bioinformatics, *e.g.*, parallel large scale phylogeny reconstruction [49, 34], and other areas that heavily rely on efficient parallel search can benefit from

distributed multithreading. These applications can be programmed in three ways:

1. using a client-server model with one or more process(es) *dedicated* to servicing remote requests,

2. employing one or more extra threads per process image to explicitly service remote requests, or

3. spawning remote threads on demand to access remote state.

In the first scheme, one or more processes do not perform useful work and CPU resources might be wasted if they run on dedicated CPUs. This can happen if a cluster job scheduling software does not allow an asymmetric number of processes per node (as was the case for our experimental platform) or if it is not trivial to start the master as an extra process on one of the client nodes. Both the first and second schemes require the programmer to implement a two-sided communication protocol by encoding requests and marshaling their parameters and return values. In addition, the programmer must explicitly create and terminate threads or maintain a pool of threads to be able to service more than one request at a time. The third approach relieves the programmer from implementing such a protocol. Instead, the logic is implemented via co-functions/co-subroutines (CFs/CSs). Remote computations are created implicitly on demand and implicitly terminated by the run-time when finished. The compiler and run-time marshal the parameters and return values without programmer's involvement. Thus, the third approach provides the most flexible and natural programming style.

9.1.3 Remote asynchronous updates

Some algorithms have shared state that is accessed frequently by many processors. On a distributed memory platform, it is usually beneficial to privatize this state replicating it in every memory for faster access. However, if a process modifies the state, it must propagate the change to other processors. Fortunately, many algorithms, such as parallel search,

have some tolerance to using stale state. The updates need not be synchronized throughout the system, but rather asynchronously propagated to other processors, increasing the asynchrony tolerance of a distributed algorithm.

A typical representative of such asynchronous updates is the propagation of the best cycle length in a parallel branch-and-bound TSP, which is used as the search pruning criteria. Updating the best cycle length by spawning asynchronous remote activities is efficient and natural for the following reasons.

The event of finding a shorter cycle by a processor is completely asynchronous with respect to other processors. If barriers were used to propagate the best cycle length, the application would advance in lock step; this is an oversynchronized solution that is not asynchrony tolerant, and any delay on one processor will cause a delay on the other processors. A non-blocking broadcast cannot be used because it does not protect the best length from concurrent updates by several simultaneous broadcasts. If a co-array is used to store the shortest length on every image, GETs and PUTs on this co-array must be protected by synchronization, *e.g.*, a distributed lock. If the image that found a shorter cycle is to update its length on every other image, it has to acquire the distributed lock, read the remote co-array value, test whether the new value is still smaller (because meanwhile another processor might have found even a shorter cycle and updated the best length), and if so update co-array value in remote memory, and release the lock.

A better solution is to spawn asynchronous remote activities on every image to update the shortest cycle length. Such an activity spawned on image p acquires a lock l_p, local to p, updates the best length, if necessary, and releases l_p. This approach is more efficient because synchronization necessary to protect the best length is local to each process image. The program code is also cleaner because remote operations are logically grouped into a user-defined multi-step operation, which is spawned and executed remotely.

9.1.4 Other applications

DMT enables chains of activity invocations, where an activity a_1 spawns an activity a_2 that, in turn, spawns an activity a_3 and so on. Using chains, global propagation of the shortest tour length might use a logarithmic depth tree-structured distribution pattern. High-Performance Linpack Benchmark [93] uses a source-level non-blocking broadcast, a good candidate to implement using DMT activity chains. Activity chains can also be used to implement *counting networks* [10] and *diffracting trees* [110] on a distributed memory machine.

9.2 DMT design principles

The goal of DMT is to enable co-location of computation with data to reduce exposed interconnect latency and to exploit hardware parallelism within a cluster node. We believe that DMT design must provide a uniform mechanism to support remote and local activities. In this section, we overview run-time support necessary for DMT and our major design decisions.

9.2.1 Concurrent activities within a node

With the increasing number of available CPUs in multi-core multiprocessor cluster nodes, it is essential that the CAF parallel programming model exploit this parallelism. There are several ways to do it. First, programmers could use both CAF and OpenMP. CAF provides locality control, while using OpenMP within a CAF process image on a multiprocessor/multi-core node would enable one to exploit loop-level or task parallelism (via SPMD regions); however, OpenMP is not well suited for recursive divide-and-conquer or nested parallelism. Second, several CAF images can run within one node. Neither of these is flexible enough if an application benefits from different forms of parallelism expressed via several concurrent activities within one CAF image. Several concurrent activities naturally require some sort of thread support from the run-time layer; we assume

that each run-time thread executes only one activity at a time. The questions to answer are whether run-time threads should be cooperative and how many concurrent threads should be allowed.

Cooperative threads have run-until-block semantics. They rely on the threads to explicitly relinquish control once they are at a stopping point. Therefore, the number of concurrently running cooperative threads cannot exceed the number of node's processing elements. For some applications, cooperative multithreading offers some advantages because context switches for cooperative activities happen at well-defined points; moreover, when there is only one run-time thread executing cooperative activities within a process, these activities do not need to be synchronized. However, if activities execute long computations, other activities cannot run and the system becomes non-responsive. In particular, this precludes servicing (responding to) requests from other images. Codes that require responsiveness have to be structured to break a long computation into smaller pieces and yield control in between to attend to other activities. This is a rather strong demand that would make the execution model less appealing to scientific programmers. In addition, the requirement of cooperative multithreading will restrict the compiler's ability to automatically break long computations. We believe that the programming model should not require activities to be cooperative. DMT activities are preemptable; *i.e.*, an activity can be preempted by another activity at an arbitrary point. An implementation, on the other hand, can provide an option to execute a program with the guarantees of cooperative multithreading; which might be especially useful for programs with only one run-time thread executing cooperative activities within a process.

If an operating system (OS) provides support for preemptable threads, *e.g.*, Pthreads, a DMT can be implemented efficiently by exploiting these threads to execute activities. However, operating systems for the largest scalable systems, Blue Gene/L [56] and Cray XT3 [38], currently do not provide support for OS multithreading. With only one thread of execution, it is unlikely that one can implement DMT to deliver high performance. Instead of relying on the operating system to context switch threads, the compiler would have to

insert explicit `yield` instructions in the code that relinquish control to the activity scheduler, which multiplexes activities. How to place `yield` instructions to also deliver high performance is outside the scope of this work. Fortunately, there is a strong indication that thread support will be available on these architectures in the future as they move to multi-core processors with larger number of cores. The HPCS programming languages X10 [69], Chapel [39], and Fortress [7] will require OS threads to efficiently support concurrent activities. ZeptoOS [92], an open-source Linux operating system designed to work on petascale machines such as Blue Gene/L and Cray XT3, is a good candidate to enable OS thread support on these two architectures. Interestingly, the ZeptoOS development team recently announced plans to support function-call shipping for x86-based petascale machines [92] (8/2/06). In the following discussion, we focus on the case when the OS provides support for threads.

Another important issue is how many concurrent computations can/should be active simultaneously. We believe that the programming model should conceptually allow an unbounded number of concurrent activities so as not to restrain the programming style by the inability of the system to make progress when too many activities are blocked. An implementation can execute each activity in a separate OS thread. Unfortunately, the performance degrades when there are too many concurrently running threads. However, most applications do not need many concurrent activities. A pragmatic approximation to the conceptual model is to maintain a *pool of threads* that execute queued pending activities one after another. The minimum size of the pool has to be specified to the run-time by the programmer because it depends on the algorithmic properties of the application and there are no known techniques to determine it automatically in the general case. The number of concurrent activities also affects performance. Too many concurrent activities leads to performance degradation because of context switches and cache contention. Too few might lead to poor responsiveness. Since the performance aspect of a parallel programming model is very important, an implementation is encouraged to provide “knobs”, *e.g.*, thread pool size, to tune the performance of an application.

We believe that any implementation should provide at least two mechanisms to spawn asynchronous activities. The first should enable an unbounded number of concurrent activities, but might not deliver the best performance. The second should enable only a certain number of concurrent activities, *e.g.*, limited by the thread pool size. The implementation should also expose facilities to control the parameters of the thread pool.

9.2.2 Remotely initiated activities

Any distinction between activities initiated locally or remotely should be minimized. Local activities are created by activities running within a process image and can be added for immediate scheduling. Remote activities initiated on process image p are added for scheduling only after p has serviced its interconnect interface incoming queue. p can attend to the network in two ways, using an interrupt-based or polling-based approach. Which approach is used determines the responsiveness or how quickly a remotely initiated activity can be added for execution. We discuss each of these approaches in turn.

The interrupt-based approach is used by the ARMCI communication library. ARMCI uses a dedicated thread, called the server thread, to service the network. Conceptually, the server thread sleeps waiting for a network request to arrive. When a request arrives, the thread is unblocked and processes the request. This mechanism provides good responsiveness because remotely initiated activities can be added for scheduling with little delay. If a node has an unused CPU, the server thread can opportunistically poll the network rather than sleep waiting for a request[2]. In this case, polling decreases the response time, which benefits single threaded programs; however, server thread polling is likely to consume extra CPU cycles in a DMT multithreaded program, limiting its applicability. Because handling of remote requests is asynchronous, hence the name interrupt-based approach, with respect to image's running activities, the server thread can cause undesirable interference, for example, by evicting a portion of running activities' cache due to a data copy.

[2]This is *not* the polling-based approach because the server thread is polling the network interface, not an application thread.

The polling-based approach uses the application's thread(s) to attend to the network at well-defined points. There is no server thread; instead, remote requests are serviced when application code calls either run-time layer functions (implicit polling) or an explicit function, *e.g.*, `poll`. The advantage of this approach is that remote requests are processed at well-known points in the program, thus, localizing the interference caused by processing remote requests to these places. The disadvantage of this approach is that the node may not be responsive to the network requests, for instance, when the application thread(s) is performing a long local computation. For codes that need responsiveness, programmers would have to insert explicit polling instructions in long computations.

If the execution model is restricted to the polling-based approach, the compiler has very limited capabilities to insert polling instructions automatically. Compiler-inserted polling instructions are not "controlled" by the programmer. They have the effect of asynchronous request processing, analogous to the interrupt-based approach. This violates the assumptions of the polling-based approach and nullifies its advantage. Moreover, if the architecture does not have thread support, polling (explicit, implicit, and/or compiler-generated) is the only option to make progress.

The polling-based approach would pose an extra burden on the scientific programmer to insert explicit polling instructions. It also severely restricts the ability of a compiler to automatically insert polling instructions, which is vital for supporting multithreading on architectures without OS thread support. The interrupt-based approach provides better programmability and does not restrict the compiler to automatically insert polling instructions, but might lead to degraded performance due to asynchronous data copying or activity interference. In summary, none of the approaches provides a universally good solution. We believe that the programming model should not sacrifice programmability and should *not* guarantee that remote requests are processed at certain places in the code. Under this assumption, the compiler is also free to insert polling instructions. An implementation may provide support for both interrupt-based and polling-based approaches as well as the ability to explicitly disable/enable processing of remote requests.

9.2.3 Host environment of activities and parameter passing

DMT must provide the ability to co-locate computation with data. In CAF, data resides in a particular process image, specified explicitly by the programmer. The host environment of an activity must naturally be the target process image p where the activity executes. Each activity should be able to access co-arrays on any image as well as global (SAVE, COMMON, MODULE) and heap variables of p.

Programmers should be able to pass values to newly created activities and receive results back. Locally created activities can access these values directly since they are sharing the same process memory. A pragmatic approach to passing parameters to a remote activity is to make a "snapshot", or closure of values, at spawn point and make these values available to the activity. Similarly, when activities return values to the origin image, they must carry the values back and place them in the result variables in the origin image.

9.2.4 Synchronization

Concurrent activities must be able to synchronize. We believe that synchronization primitives must be built on widely-accepted concepts and must not introduce significant overhead. DMT provides different mechanisms for intra-image and inter-image synchronization. Locks and condition variables [82] are good candidates for synchronization of local activities. They are familiar to programmers and, in the absence of contention, do not incur much overhead. We assume standard (release consistency) semantics for locks where a lock executes a local memory fence operation that propagates results of writes into the node's memory making them visible to other threads active on the node.

Process images are synchronized by CAF's barrier and team synchronization primitives called by *one* of the activities running within each image. Inter-image synchronization primitives should be extended to use an additional *tag* parameter, whose interpretation is user-defined. Tags enable several synchronization contexts per process image, executed by different activities. In some sense, tags enable synchronization of particular remote activities, and each image can participate in several such synchronization events. The number

of tags is defined by a particular implementation. It is pragmatic to assume that a tag is represented by a 64-bit integer, so the number of available tags should not be a problem. Locks and tagged inter-image synchronization can be combined to synchronize groups of activities from different images. An alternative approach, explored by Cristian Coarfa [29], is to use eventcounts [99] for both inter-image and intra-image synchronization.

9.2.5 Extensions to the memory consistency model

DMT introduces a few additional rules for CAF's memory consistency model defined in Section 3.1.6.

Locks and condition variables assume the standard release consistency. Accesses to variables declared with the `volatile` attribute have the same semantics as in Fortran 95 [5]. These two conditions allow the compiler to perform standard sequential optimization in between synchronization points to deliver high scalar performance. DMT should provide a *local* memory fence operation that propagates results of writes into the node's memory making them visible to other threads active on the node, to be able to write language-level primitives for intra-image synchronization. The DMT lock release operation has an implicit local memory fence to make the writes of the thread leaving a critical section visible to other node's threads.

An open question.

It is not entirely clear what the memory consistency model should be for remotely-spawned activities, especially for chains of remotely-spawned activities. There is not enough experience with using distributed multithreading in scientific codes to define the exact model yet. We describe two candidates.

The first is more intuitive for the programmer, but may cause performance degradation. A spawned activity A "knows" the execution history of the origin activity O, which includes local co-array reads/writes, co-array PUTs/GETs, synchronization calls, and completed activities performed by O prior to spawning A. Intuitively, A should observe the

```
a = 1
a[2] = 2
a[3] = 3
call foo()[2]
x1 = b
```

(a) code on image 1

```
cosubroutine foo
  x2 = a[1]
  y2 = a
  z2 = a[3]
  call bar()[3]
  return
```

(b) `foo` on image 2

```
cosubroutine bar
  x3 = a[1]
  y3 = a[2]
  z3 = a
  b[1] = 10
  return
```

(b) `bar` on image 3

Figure 9.3 : Activity chain (note that `[]` denote remote co-array references).

results of these operations when it starts. Likewise, when A returns, O should observe the results of A's execution up to A's return point. Pseudocode in Figure 9.3 shows a scenario for a chain of activities initiated using blocking spawn. Image 1 spawns an activity `foo` on image 2. In turn, `foo` spawns an activity `bar` on image 3. `a` and `b` are co-arrays; `x1`, `x2`, `x3`, `y1`, `y2`, `z2`, and `z3` are local variables. The value of `x2` should be 1; the value of `y2` should be 2; but what should be the value of `z2`? If `foo` spawns an activity `bar` on image 3, and `bar` reads the value of `a`, this value should be 3. Thus, `x3` should be 1, `y3` should be 2, `z3` should be 3. Implementation techniques that ensure that `z3` is equal 3, but `z2` is equal to something else, and offer performance gain are unlikely to exist. Due to this observation, the value of `z2` must be 2. Finally, the value of `x1` on image 1 should be 10 after `foo` returns.

One feasible implementation strategy is to execute a memory fence right before spawning (`call` in Figure 9.3) a new activity and right before a spawned activity returns (`return` in Figure 9.3). The memory fence in a multithreaded environment completes outstanding memory operations issued by the activity rather than by all activities running within the image. The overhead of memory fence operations might be high for fine-grain activities. While a CAF compiler could possibly uncover opportunities to eliminate or weaken fences in some cases, an implementation may also support hints to indicate that a memory fence is not necessary.

Our second memory consistency model is less intuitive for the programmer, but allows

```
cosubroutine foo(a, n)
  integer, intent(IN) :: n
  integer :: a(n)[*]
  ...
end cosubroutine foo
```

(a) co-subroutine `foo`

```
cofunction bar(i)
  integer, intent(IN) :: i
  double precision bar
  ...
end cofunction bar
```

(b) co-function `bar`

Figure 9.4 : Examples of CS and CF declarations.

an efficient implementation that can potentially hide all communication latency. Under this model, an activity spawned in image `p` can observe only results of PUTs issued by the spawner to `p` prior to spawning the activity (similar to the semantics of the weak notify). This guarantees only that the value of `y2` is 2; `x2` and `x3` need not be 1.

9.3 Language support for DMT in CAF

This section defines a small set of language extensions to add distributed multithreading to CAF.

9.3.1 Language constructs

- The **`cosubroutine`** (CS) and **`cofunction`** (CF) keywords are used to declare a subroutine or a function that can be spawned. Figure 9.4 shows two declaration examples. The `intent` attribute [5] specifies the intended usage of a CS/CF (CS for short) dummy argument. `intent(IN)` indicates that CS must not change the value of the argument. `intent(OUT)` means that CS must not use the argument before it is defined. `intent(INOUT)` argument may be used to communicate information to CS and return information. If the intent attribute is not specified, it is `intent(INOUT)`.

- The **`call`**, **`spawn`** and **`ship`** keywords are used to spawn CS/CF remotely or lo-

cally. `call` is used for blocking spawning; it is a "syntactic sugar" for `spawn` that is used for non-blocking spawning. `spawn` returns an explicit handle used to await the completion of the activity later. `ship` is used for non-blocking spawn without a return value (note that CFs cannot be `ship`-ed because they always return a value). By analogy to local and remote co-array accesses, `call`, `spawn`, `ship`, or CF invocation *without* `[]` (brackets) indicate than an activity is initiated locally, as shown in Figure 9.6. `call`, `spawn`, `ship`, or CF invocation *with* `[]` (brackets) indicate than an activity is initiated remotely in the process image specified in the `[]`, as shown in Figure 9.7.

- The **`reply`** keyword-statement is used by the CS/CF to return values to the spawner and to enable the spawner to continue execution, if the spawner is blocked waiting for the spawnee's reply. Note that `reply` is not equivalent to `return`; the spawnee might proceed execution after `reply` until it returns (executing a `return`). Figure 9.5 shows an example of `reply` usage.

```
cosubroutine foo(int a)
  integer, intent(INOUT) :: a
  a = a + 1
  ! return the value of a to the spawner image
  reply
  ! perform some additional work
  call bar(a)
  ...
end cosubroutine foo
```

Figure 9.5 : Using a `reply` to return `INOUT` and `OUT` parameters to the spawner.

- The **`await_reply`**`(handle)` construct is used to await the completion of the activity initiated via a non-blocking `spawn` that returned handle `handle`. Figures 9.6 (3) and 9.7 (3) show the usage of `await_reply`.

- The **`sync`** keyword-statement completes co-subroutines that were spawned with `ship`, as shown in Figures 9.6 (4) and 9.7 (4).

- The **`get_id()`** intrinsic returns the activity ID, a unique number within the process image index for each activity, constant for the lifetime of the activity.

- The **`get_spawner_image()`** intrinsic returns the process image index from which the activity was initiated.

- The **`get_spawner_id()`** intrinsic returns the ID of the activity that spawned the current activity.

- Type **`type(CAFMutex)`** declares a mutex object for intra-image synchronization (see Section 9.2.4). **`caf_lock`**`(mutex)` and **`caf_unlock`**`(mutex)` acquire and release `mutex`.

- Type **`type(CAFCond)`** declares a condition variable for intra-image synchronization (see Section 9.2.4). **`caf_cond_wait`**`(cond)` puts the activity to sleep on a condition variable `cond`. **`caf_cond_signal`**`(cond)` wakes up an activity sleeping on the condition variable `cond`; **`caf_cond_broadcast`**`(cond)` wakes up all such activities.

- The **`local_memory_fence()`** intrinsic flushes all writes of the current activity to memory. It can be used to write codes with data races that require stronger ordering guarantees, *e.g.*, for custom intra-image synchronization.

- The **`yield`** keyword-statement instructs the run-time to yield the execution of the current activity.

- The **`poll`** keyword-statement instructs the run-time to process remotely initiated activities.

1	`call foo(...)`	`foo` executes in the context of the local image and the spawner is blocked until `foo` replies
2	`a = bar(...)*i + 5`	`bar` executes in the context of the local image; the spawner is blocked until `bar` replies at which point the statement is computed

3	`handle = spawn foo(...)`
	`...`
	`call await_reply(handle)`
		`foo` executes in the context of the local image; the spawner continues execution and must block in `await_reply` until `foo` replies

4	`ship foo(...)`
	`...`
	`sync`
		`foo` executes in the context of the local image; the spawner continues execution right away and never waits for `reply`; `sync` blocks until such spawns complete (by explicit reply or return)

5	`a = bar1(...)*i +`
	`  bar2(...)  +`
	`  bar3(...)  + 7`
		`bar1`, `bar2`, and `bar3` execute *concurrently* in the context of the local image; the spawner is blocked until all three reply at which point the statement is computed

Figure 9.6 : Locally initiated activities (no `[]` after CF/CS; the spawnee image is the same as the spawner image).

9.3.2 DMT semantics

Declaration

The `cosubroutine` and `cofunction` keywords are necessary for the declaration of a CS/CF to enable separate compilation. They indicate to a compiler that the program unit requires special calling convention and parameter handling. Figure 9.4 shows two declaration examples.

1	`call foo(...)[p]`	`foo` executes in the context of the spawnee's image `p`; the spawner is blocked till `foo` replies
2	`a = bar(...)[p]*i+7`	`bar` executes in the context of the spawnee's image `p`; the spawner is blocked until `bar` replies at which point the statement is computed

3	`handle = spawn foo(..)[p]`
	`...`
	`call await_reply(handle)`
		`foo` executes in the context of the spawnee's image `p`; the spawner continues execution and blocks in `await_reply` until `foo` replies

4	`ship foo(...)[p]`
	`...`
	`sync`
		`foo` executes in the context of the spawnee's image `p`; the spawner continues execution right away and never waits for `reply`; `sync` blocks until such spawns complete (by explicit reply or return)

5	`a = bar1(...)*i  +`
	`  bar2(...)[p] +`
	`  bar3(...)[q] + 7`
		`bar1`, `bar2`, and `bar3` execute *concurrently* in the context of their images: local, `p` and `q` respectively; the spawner is blocked until all three reply at which point the RHS expression is computed

Figure 9.7 : Remotely initiated activities (the spawnee image is specified in `[]`).

Execution and execution context

A CS/CF spawned on image `p` is said to be executed in the context of image `p`. There is little difference between locally and remotely-spawned CSs. A CS behaves as if it were a thread running in image `p`.

A CS can access co-array data and private data of image `p`, *e.g.*, `COMMON`, `SAVE`, `MODULE`, and heap variables. The co-array local part for a CS is the one that resides in the

```
cofunction bar()
  integer bar
  ...
  bar = ...
end cofunction bar
```

(a) co-function

```
cosubroutine bar_sub(bar_res)
  integer, intent(OUT) :: bar_res
  ...
  bar_res = ...
end cosubroutine bar_sub
```

(b) equivalent co-subroutine

Figure 9.8 : Conversion of a co-function into the equivalent co-subroutine.

spawnee image `p` memory. A CS can access co-arrays on other images. CSs can participate in intra- and inter-image synchronization. CSs can allocate private variables as well as co-arrays; however, remotely-spawned CSs cannot allocate/deallocate parameters or return pointers. CAF intrinsic functions are computed relative to image `p`. A CS can call subroutines/functions and can also spawn CSs. CSs can perform arbitrarily long computations and block in synchronization[3] or I/O.

CS/CF co-space

A co-array declaration or allocation defines a co-shape used to compute the target process image index of a remote co-array access. On the contrary, a CS declaration does not define any co-shape. Instead, a CS/CF has an implicit 1D co-shape `[*]`, which corresponds to process image indices in the range $[1,$ `num_images()`$]$. The target process image index of a `call`, `spawn`, `ship`, or co-function invocation can be specified as an integer number i, $i \in [1,$ `num_images()`$]$, or using the interface functions of `CAF_WORLD`, group, Cartesian, or graph co-spaces (see Chapter 5).

Blocking spawning and semantics of `call`

A `call` used to initiate a blocking CS is just "syntactic sugar" for a non-blocking `spawn` completed right after it was initiated as shown in Figure 9.9.

[3]Note that a deadlock is possible, *e.g.*, the spawnee tries to acquire a lock held by the spawner and the spawner is blocked in `call` waiting for the spawnee to `reply`. DMT does not prevent deadlocks; it is programmers' responsibility.

```
call foo(...)[p]
```

(a) blocking spawn

```
tmpHandle = spawn foo(...)[p]
call await_reply(tmpHandle)
```

(b) equivalent non-blocking spawn

Figure 9.9 : Blocking and equivalent non-blocking CS spawn.

Figures 9.6 (1,2) and 9.7 (1,2) present the blocking style of spawning local and remote activities. If no brackets are present, a CS/CF is spawned in the local image; if brackets are present, a CS/CF is spawned in the image defined by the expression in the brackets. Even though the statement in Figure 9.6 (1) looks like a regular Fortran 95 subroutine call, its semantics are different as follows from the discussion below. Co-functions, Figures 9.6 (2) and 9.7 (2), are always blocking because their return values are necessary to execute the statement. A co-function `bar` is converted into an equivalent co-subroutine `bar_sub` by passing the return value as an extra argument `bar_res` and rewriting all assignments to `bar` as assignments to `bar_res`; Figure 9.8 shows an example of such a conversion. After `bar_sub` replies, the value returned in the `bar_res` variable is used in computation in the place of the corresponding function call.

Using these two transformation, all blocking CF and CS spawns can be converted into the form shown in Figures 9.6 (3) and 9.7 (3), which is the focus of our further discussion.

Semantics of non-blocking `spawn`

The `spawn` construct creates a spawnee activity that runs concurrently with the spawner. The spawner does not block. The spawner represents the spawnee via a *spawn handle*, *e.g.*, `type(DMTActivity)::handle`. When the spawner executes a `await_reply(handle)`, it blocks and waits for a reply from the spawnee; after the reply is received, the spawner continues execution. The return `INOUT` and `OUT` parameters become available to the spawner at the point of `await_reply`.

Statements with several co-function calls, *e.g.*, in Figures 9.6 (5) and 9.7 (5), are of particular interest. All three CFs can be executed concurrently, perhaps in different process

images. The order of evaluation is not defined and full control over side effects is left to the programmer. To clarify the semantics, let us consider the statement in Figure 9.7 (5). It is transformed into the following equivalent piece of code, assuming CFs bar_i, $i \in [1,3]$, were converted into CSs bar_i_sub. The CSs execute concurrently and all three intermediate temporaries are available after the last `await_reply`.

```
h1 = spawn bar1_sub(..., tmp1)
h2 = spawn bar2_sub(..., tmp2)[p]
h3 = spawn bar3_sub(..., tmp3)[q]
call await_reply(h1)
call await_reply(h2)
call await_reply(h3)
a = tmp1*i + tmp2 + tmp3 + 7
```

Fortran 95 allows but does not mandate short-circuit evaluation of boolean expressions [5]. Similarly, DMT allows but does not mandate short-circuit evaluation of boolean expressions containing CF invocations; the choice is left to a particular implementation.

Semantics of `reply`

The `reply` keyword-statement is a "remote return" that unblocks the spawner and returns the values of `INOUT` and `OUT` parameters to the spawner. If the spawner if blocked in a `await_reply` statement, `reply` allows it to continue execution. Only one `reply` is allowed; execution of more than one `reply` in the same invocation of CS/CF is a critical run-time error. If a CS/CF returns and no `reply` has been explicitly issued, the run-time layer deliver an implicit `reply` to the spawner. At the spawner side, all state associated with the activity is deleted when the matching `await_reply` executes. The spawnee activity can; however, continue execution until it returns.

Only CSs/CFs can execute `reply`. Calling `reply` in a regular Fortran 95 subroutine/function results in a compile-time error.

Termination of concurrent activities

An activity terminates upon execution of an explicit `return` or when it reaches the last statement. If it has not executed a `reply`, the run-time sends an implicit `reply` to the

spawner. All state associated with a terminated activity on the spawnee's image is deallocated. Fortran `stop` [5] statement aborts the entire program and terminates the process image; as a result, all activities within the image are terminated.

Parameter passing

Parameter passing conventions for locally- and remotely-spawned activities are slightly different, but we believe that this is necessary to reduce overhead of spawning for local activities. Locally-spawned activities run in the same address space as the spawner. In this case, parameters are passed using the same rules as for Fortran 95 subroutine parameter passing; there are no restrictions. However, remote activities cannot access (non-co-array) arguments directly. When `spawn` executes, the spawner marshals `IN` and `INOUT` non-co-array parameters by packing them into a buffer and transferring them to the target process image, where the run-time layer makes them available for the spawnee. Upon execution of a `reply`, the spawnee marshals `INOUT` and `OUT` non-co-array parameters and transfers them to the spawner's image, where the run-time layer unpacks them into the proper variables. The lifetime of a remotely-spawned activity's arguments is the duration of the activity. To marshal parameters, it is necessary to know their sizes; therefore, there are additional restrictions on the types of arguments passed to remotely-spawned activities.

Non-co-array arguments of a remote CS are allowed to be local scalars, arrays, and Fortran 95 pointers of primitive and user-defined types as well as subroutine/CS/CF pointers. Parameters of user-defined types with an allocatable or pointer fields (either in the parameter type itself or one of its field types) are passed by "shallow" copy. Otherwise, they would require transmitting of all data reachable by the parameter pointer components to the remote process image. This would degrade performance, increase the number of side effects, and complicate the implementation. Most importantly, it would defeat the purpose of shipping computation closer to data if the computation must drag all data linked with it to the remote memory. Because parameter packing/unpacking and communication incurs overhead, it is recommended that programmers do not pass large actual arguments to CSs.

However, this is purely the programmer's decision. One could even implement communication, packing/unpacking of strided communication, communication aggregation, etc. using CSs that transfer data as parameters.

Marshaling requires knowing parameter sizes and shapes. The size and shape of scalar arguments are always known. In Fortran 95, the shape of array parameters is defined by their declarations with dimensions declared in terms of specification expressions [5], *e.g.*, `integer a(N+1)`, where `N` is also a parameter to the subroutine or a global variable[4]. In CAF, global variables referenced in specification expressions are private to each image and may have different values on different images. One may not use global variables in specification expressions of a CS/CF; this is a compile-time check. If a value of a global variable needs to be used, it can always be passed as an extra argument to CS/CF. With this restriction, all array and co-array shapes are properly dimensioned when evaluated on the spawner or spawnee. If a parameter is passed by Fortran 95 pointer, the caller passes a dope-vector that specifies the shape information necessary for marshaling. If Fortran 95 pointer points to a strided memory section, it is likely that an implementation would compact the section, transmit the contiguous message over the network, and adjust the Fortran 95 pointer on the spawnee to point to the contiguous memory section corresponding to the parameter data. DMT cannot support marshaling of parameters with unknown sizes, *e.g.*, array arguments with an implicit bound.

Co-array arguments are passed differently than Fortran 95 variables. A co-array exists in the spawnee's image, so no co-array data is transferred. Instead, the co-array parameter becomes local to the spawnee's image, as if the argument was passed in the context of the remote image (see Section 9.4).

Semantics of `ship`

The `reply` construct incurs extra overhead if the corresponding spawn is remote; thus, if the reply is not semantically necessary, it should be avoided. One example is a CS that

[4]Here, we also assume that variables of a host subroutine are "global" to the spawnee.

updates a remote counter and returns nothing to the spawner. Another example is a CS that we developed for a fine-grain implementation of the RandomAccess HPC Challenge benchmark [1]. It performs remote XOR updates of random memory locations; we present pseudocode and description in Section 3.4.3. Sending a `reply` message for each XOR update would effectively reduce the interconnect bandwidth available for sending updates by half. The `ship` operator shown in Figures 9.6 (4) and 9.7 (4) is designed for codes that spawn many non-blocking remote computations, for which individual replies are not necessary. A shipped co-subroutine is not allowed to return a value. Thus, it can have only `IN` parameters (a compile time check).

The spawner of a co-subroutine `foo` does not block and does not have any language-level state (*e.g.*, `handle`) to check whether `foo` has completed. A `ship`-ed CS may complete by either returning or executing a `reply` statement. The `sync` keyword-statement waits for the completion of activities spawned via `ship`.

Semantics of `sync`

`ship`-ed activities belong to a *ship-epoch*, or epoch for short. An epoch is defined by execution of successive `sync`s. Each `ship`-ed activity belongs to only one epoch. Each epoch belongs to a *ship-context*. By analogy to Cilk [63], a subroutine/CS/CF invocation implicitly creates a ship-context; the subroutine return implicitly completes (`sync`) all incomplete activities `ship`-ed from the context and destroys the ship-context. We adopted these semantics to provide Cilk-like activity invocations for local, and remote, activities. This benefits programmability of codes with recursive parallelism, and the programmer does not have to think about explicit completions of ship-contexts. A unique program-level ship-context is live for the duration of the program. `sync` completes all `ship`-ed activities within the current epoch and starts a new epoch. It is *not* a collective call.

Enforcing an implicit `sync` at the end of a (co-)subroutine restricts how a ship-epoch is defined. Some codes do not need to know at all whether `ship`-ed activities have completed because they can obtain this information from the algorithmic properties of the applica-

tion. For instance, one would need to `ship` a chain of activities to implement counting networks [10]. However, the result does not need to be returned to the original spawner through the "reversed" multi-hop chain; it can be sent directly to the original spawner O if O's index is passed along the chain. If the subroutine/co-subroutine `return` enforces the completion of these `ship`-ed activities, it exposes the synchronization latency. We suggest to use a `nosync` directive at the start of a (co-)subroutine to not create a new ship-context. Activities `ship`-ed from such a subroutine belong to the current ship-epoch and ship-context.

To give finer control over the management of `ship`-ed activities, it might be useful to have a directional `sync(p)`, which completes all `ship`s destined to `p`. Another useful extension might be to support explicit ship-epochs, *e.g.*, using `e = start_ship_epoch()` and `end_ship_epoch(e)`. `sync(e)` completes all `ship`s from the epoch with the ID `e`.

9.4 DMT implementation and experience

We designed and implemented prototype support[5] for DMT and evaluated function shipping for several codes such as TSP and RandomAccess. The DMT prototype also supports the run-time aggregation of compiler-recognized fine-grain activities (see Section 9.5.2). Currently non-supported features include: co-functions (instead, a programmer can use an equivalent co-subroutine with the extra `OUT` argument), co-subroutine parameters of user-defined types, subroutine pointers and parameters passed by Fortran 95 pointer, support for implicit ship-contexts (there is only one program-level ship-context), and easy-to-implement features such as `get_id()`[6], `get_spawner_image()`,

[5] There is no front-end support for `call`, `spawn`, and `ship`. We specify them via a function call, *e.g.*, `handle=spawns(foo(args),target)`, where `foo` is actually a co-subroutine, represented as a co-function whose return value is ignored.

[6] For example, for Pthreads, it is possible to attach a context to the thread running the activity and retrieve the context later.

`get_spawner_id()`, `yield`, and `poll` that we did not need for our experiments. The implementation uses ARMCI with Global Process Calls (see Section 3.2.1), which are referred to as Active Messages (AM), to execute computation in remote process and the Linux Pthreads [82] library to support intra-image multithreading. Currently, `cafc` with DMT is available only for an Itanium2 cluster with a Myrinet 2000 interconnect. This is the only platform where both `cafc` and ARMCI with AM support are available today. We now briefly discuss the major implementation decisions and our experience with DMT.

9.4.1 Implementation overview

DMT uses Pthreads threads to execute activities. We refer to these threads as run-time threads. Run-time threads are synchronized via Pthreads mutexes and condition variables. Mutexes and and condition variables are implemented using the corresponding Pthreads primitives.

The `cafc` compiler supports marshaling of `IN`, `INOUT`, and `OUT` scalar, array, and co-array parameters. For each original co-subroutine, `cafc` creates a stub function S_1 and a subroutine S_2. The spawner calls S_1 at the spawn site to marshal arguments and to invoke an AM that sends the activity for execution in the target image[7]. The AM invokes S_2, which executes in the context of the spawnee image and performs the same computation as the original co-subroutine.

Each `ship`-ed activity has a context on the spawnee. All other activities have contexts on both the spawner and spawnee. A context is a data structure representing the run-time state of the activity. The spawner context is created when the activity is spawned; the execution of the AM handler on the spawnee creates the spawnee context. The spawner context has information about the activity ID, spawn type, spawnee image index, arguments, S_2's address, reply state, activity ship-parameters (ship-context and ship-epoch), and other implementation specific details. The spawnee context has information about the activity ID,

[7]We focused mainly on remote function shipping and did not optimize parameter passing for locally-spawned activities; passing parameters locally does not require marshaling.

spawn type, spawner image index, spawner's context address, arguments, S_2's address, reply and termination states, activity ship-parameters, and other implementation-specific details.

Function S_1 has the argument list of the original co-subroutine extended with two parameters: the spawnee image number `p` and the type of spawn `t`. It performs the following steps. S_1 computes the sizes (rounded up according to the arguments' alignments) of `IN`, `OUT`, and `INOUT` non-co-array arguments to know the size of the argument buffer. S_1 instructs the run-time to create a spawner context and to allocate the argument buffer. S_1 instructs the run-time layer to pack the arguments into the buffer one after another since Fortran 95 passes scalar and array arguments by-address and their sizes, intents, and types are known. The run-time layer uses padding to start each argument with an offset that is multiple of the argument's natural alignment; so the size of the argument buffer may be slightly larger than the total size of all arguments.

Co-array arguments require different handling. `cafc` converts each co-array parameter into two: the co-array handle H and the address of the local part L. The spawner uses L to compute the local co-array address that is valid in the target's image address space. This is possible since every image has the co-array start addresses for every other image. H is a pointer to the co-array run-time descriptor data structure residing in the spawner's memory. To locate the co-array descriptor in the spawnee's memory, `cafc` uses a co-array ID `id` that is a unique number for each co-array in the program. Therefore, a parameter co-array is represented in the argument buffer via two fields: its local part remote address and `id`.

Finally, S_1 instructs the run-time layer to initiate a non-blocking AM that transfers the information necessary to start the activity in the target process image. This information includes the relevant part of the spawner context: the activity ID, spawn type, spawner's context address, S_2's address, ship-parameters, and arguments. The information about arguments includes the values of `IN` and `INOUT` arguments as well as argument offsets in the argument buffer and their sizes necessary to unmarshal the arguments. The spawner's context address is used to located the activity spawner context on the spawner when the

activity replies (alternatively, one could attach state to a Pthread). S_1 returns the address of the spawner context as the spawn handle used to complete the activity later.

Subroutine S_2 has the argument list of the original co-subroutine extended with the spawnee context handle parameter `sch`. S_2 executes the same code as the original co-subroutine. The run-time layer uses `sch` to associate the spawnee context with the activity. Through `sch`, the run-time layer can find the spawnee context, *e.g.*, to perform a `reply` operation. The AM handler creates the activity spawnee context. It copies the information from the AM payload, which does not exist after the AM handler returns, into the spawnee context. In addition, it creates a vector v of pointers to represent the addresses of S_2's arguments; the pointers are in the same order as the arguments of S_2. The AM handler computes the elements of v by calculating the addresses of `IN`, `INOUT`, and `OUT` arguments, which reside in the spawnee context argument buffer. It also determines the address of the co-array descriptor for each co-array argument using the co-array ID. Finally, the AM handler enqueues the activity for execution and returns.

An activity (spawnee context) waits in the activity ready queue Q until it is dequeued and executed by one of the run-time layer threads. The thread executes the activity by calling S_2. The arguments of S_2 are the addresses from v; Fortran 95 passes scalar and array arguments by-reference. The activity terminates when S_2 returns.

9.4.2 Spawn types

DMT enables the programmer to specify how to execute a co-subroutine to deliver best performance. There are three spawn modes: the AM-style mode (AM-mode), the thread pool mode (Pool-mode), and the thread mode (Thread-mode).

The `call_am`, `spawn_am`, and `ship_am` constructs spawn an activity in AM-mode. AM-mode means that it is possible to execute the activity by any run-time thread, even a thread that is already executing a different non-AM-mode activity. Typically, a run-time thread executes only one activity a; however, an AM-mode activity b can "preempt" a and run to completion. When a run-time thread has two user computations, b must be restricted.

An AM-mode activity should execute fast and should not block its run-time thread by sleeping on a condition variable, participating in an inter-image synchronization, or waiting for a spawned activity. If it acquires locks, it must not acquire the same lock(s) that have been acquired and are still held by the preempted user computation that the thread was running, not to cause a deadlock. An AM-mode activity is atomic with respect to other activities; if a run-time thread executing an AM-mode activity is preempted, it does not execute another activity until it finishes the current one. AM-mode activities are useful to control the number of concurrent run-time threads as discussed below; some of our implementations of the RandomAccess benchmark use this mode to perform remote updates efficiently (see Section 9.6.3).

The `call`, `spawn`, and `ship` constructs spawn an activity in Pool-mode. The DMT runtime maintains a pool of threads to execute Pool-mode activities. The programmer can control the size of the pool either by setting an environment variable `DMT_THREAD_POOL_SIZE` or by calling `dmt_set_pool_size(num_threads)`. Each process image can have different number of threads in its pool. Pool-mode is less restrictive than AM-mode; an activity is free to perform arbitrarily long computations, block, and participate in intra- and inter-image synchronization. The run-time thread running a Pool-mode activity is taken from the thread pool for the lifetime of the activity and cannot be preempted to execute another activity, except an AM-mode activity. The number of *concurrent* Pool-mode activities cannot exceed the pool size. There may be queued pending activities waiting to be executed. It is the programmer's responsibility to setup an adequate thread pool size to accommodate the concurrency needs of the application. Too small of a thread pool may lead to a resource deadlock due to the lack of a run-time thread to execute a pending activity a essential for the system to make progress, *e.g.*, if other activities, holding the pool threads, are blocked waiting for a's actions. Too large thread pool may lead to performance degradation due to activity interference and context switching overhead. Unbounded number of threads can be obtained by either dynamically calling `dmt_set_pool_size(num_threads)` or using Thread-mode spawns.

The `call_thread`, `spawn_thread`, and `ship_thread` constructs spawn an activity in Thread-mode. DMT creates a new run-time thread to execute the activity. This thread is active for the lifetime of the activity. Creating a new run-time thread is expensive; so Thread-mode is most useful for long-lasting activities. Thread-mode bypasses the resource deadlock possible with Pool-mode activities because the number of run-time threads is practically unlimited.

The programmer can use AM-mode, Pool-mode, and Thread-mode functionality to specify the best way to execute activities. While the compiler may be able to determine the best mode in some cases, *e.g.*, to convert a local `call` into a Fortran `call` (see Section 9.5.3), it cannot do so in all cases.

9.4.3 `ship` and `sync` support

`ship` and `sync` are used for activities that do not return any state to the spawner. For example, our fine-grain implementation of the RandomAccess benchmark, which performs many remote fine-grain XOR updates of random memory locations, benefits from this. The spawner does not need to know that a particular update has completed; it only needs to know when all updates have completed. In fact, if an explicit acknowledgment (reply) were sent to the spawner for each update, this would significantly reduce the effective interconnect bandwidth. `sync` is used to complete all activities of the ship-epoch at once. The implementation challenge is to support `sync` semantics without sending individual replies and to maintain as little state as possible to track potentially out-of-order completion of `ship`-ed activities. We present our solution.

Each `ship`-ed activity is identified by two ship-parameters: ship-context[8] C and ship-epoch E_C. They are assigned by the *spawner* when the activity is `ship`-ed and "inherited" on the spawnee. The spawner assigns each epoch of C an epoch-ID from a monotonically increasing sequence. `sync` closes the active ship-epoch of C and starts a new one. Note that another activity can initiate `ship`-ed activities in the new epoch while `ship`-ed

[8] As of this writing, our prototype implementation of DMT supports only one program-level ship-context.

activities of a previous epoch have not yet completed. The spawner maintains a set of counters $I_{\langle C,E_C\rangle,p}$, where $p \in [1, \texttt{num\_images}()]$; $I_{\langle C,E_C\rangle,p}$ is equal to the number of activities initiated to p in ship-context C and ship-epoch E_C — $\langle C, E_C\rangle$.

The *first* activity of $\langle C, E_C\rangle$ `ship`-ed by q to p, creates the epoch E_C context on p that has a counter $F_{\langle C,E_C\rangle,q}$ equal to the number of completed[9] (finished) $\langle C, E_C\rangle$ activities on p that were `ship`-ed by q to p.

When q's activity a executes `sync`, it closes E_C and waits for completion of all $I_{\langle C,E_C\rangle,p}$ $\langle C, E_C\rangle$ activities, $p \in [1, \texttt{num\_images}()]$. a sends an *epoch-completed* AM, carrying the value of $I_{\langle C,E_C\rangle,p}$, to each image p, if $I_{\langle C,E_C\rangle,p} > 0$, and waits for the *activities-completed* AM replies. When p receives the epoch-completed AM from q and $F_{\langle C,E_C\rangle,q} = I_{\langle C,E_C\rangle,p}$, it sends the activities-completed AM reply to q. a is blocked until it receives all activities-completed AM replies.

9.4.4 Support of dynamically linked libraries

For an SPMD program, the static addresses of all subroutines and functions are the same on all process images. However, dynamically linked libraries present a minor engineering difficulty since they can be loaded into different address ranges on different images. If a dynamically linked library subroutine S needs to be invoked in a remote process, an implementation can use S's handle to find S's address in the remote process. The current DMT implementation does not support dynamically linked libraries.

9.4.5 Polling

Run-time layer polling should be avoided when executing a multithreaded application to not waste CPU resources that can be used to perform useful computation. During our experimentation, we encountered two cases when run-time layer polling degraded performance.

[9] An activity is completed if it either replied or terminated.

We disabled the ARMCI server thread polling because it was consuming a noticeable fraction of CPU resources, slowing down useful computation. In our experiments, the server thread executes a blocking `gm_receive` Myrinet call and becomes active for a brief period of time only when a network request arrives.

The MPICH barrier implementation for Myrinet, which `cafc` runtime uses to implement `sync_all`, polls the network. While this is not a problem for single-threaded SPMD programs, it degrades performance of multithreaded codes that use MPICH barriers. Alternative implementations of the barrier primitive are necessary for multithreaded codes.

9.4.6 Number of concurrent threads

The number of threads concurrently sharing a CPU affects the application's performance. For most scientific codes, one thread per physical CPU would probably deliver the best performance. However, one thread per process image limits the responsiveness of DMT because remote activities are executed only when the network is serviced. This can delay the propagation of asynchronous events, *e.g.*, the best cycle length in TSP, or cause a resource deadlock if the application requires two or more concurrent threads to make progress. This might require a compiler to insert `poll` instructions automatically or would make programmers restructure code to insert explicit `poll` statements. The optimal number of concurrent run-time threads to achieve the best performance with reasonable programming effort is application-specific; moreover, it can differ for different run-time phases of the application. Three spawn modes, `dmt_set_pool_size(num_threads)`, and `poll` allow the programmer to tune the application. For example, our fastest implementation of the RandomAccess benchmark (see Section 9.6.3), assuming it runs on a single processor node, uses AM-mode `ship_am` to apply remote XOR updates and sets the thread pool size to zero. This enables it to have exactly one run-time thread per CPU. This thread executes both code generating XOR updates and remotely `ship`-ed activities to apply XOR updates, maintaining a balance between update generation and application. We discuss several RandomAccess implementations in detail in Section 9.6.3.

9.4.7 Activity interference

In a multithreaded application, threads compete for cache/TLB, which can result in cache/TLB contention and thrashing. If an activity a performing computation is preempted by another activity b that accesses a large array, execution of b might evict a large portion of cache, degrading a's performance. In the current implementation of DMT, it is the responsibility of the programmer to reduce/avoid such problems. For example, the programmer can synchronize activities, *e.g.*, via a condition variable, to control when interfering activities are executed rather than rely on DMT.

Alternatively, a DMT implementation can support cooperative activities. The programmer controls exactly when activities relinquish control, so cache-sensitive computations are not unexpectedly preempted.

Ideally, the underlying thread library should provide the capability to suspend threads, *e.g.*, by changing their priorities. It would then be possible to extend DMT with `suspend_activities_on_pe()` and `resume_activities_on_pe()`. The first statement reserves a processing element, *e.g.*, a CPU core, for the current activity and allows it to run uninterrupted by other activities until other activities are resumed by the second statement. None of the currently available thread libraries supports this functionality.

Our DMT implementation revealed that the way activities are scheduled affects performance. Our prototype implementation favors the activities spawned remotely in AM-mode over local computation. If too many remote activities are waiting on node `p`, `p` stops accepting new requests because there are no available network buffers or memory. Meanwhile, other nodes keep retransmitting requests targeted on `p` wasting CPU cycles that could have been productively spent. In the current implementation of DMT, when an activity executes spawn, it first processes AM-mode pending activities initiated by other process images. This frees the resources faster and limits the generation rate of new activities in the system. Clearly, scenarios exist for which this simple heuristic would not deliver the best performance because of load imbalance. For top performance, the programmer is still responsible

for load-balancing the application properly, especially for not swamping a node with a lot of activities.

9.5 Compiler and run-time optimizations for function shipping

DMT enables explicit support for computation shipping and multithreading. However, the compiler and run-time layer can detect some optimization opportunities without user intervention. We outline a few promising directions.

9.5.1 Compiler recognition and aggregation of remote operations

The compiler can detect snippets of code heavily accessing remote data. If profitable, it can convert them into remotely-spawned co-subroutines, sparing programmers the effort to write such co-subroutines. In particular, the compiler can detect regions of code with a lot of fine-grain remote updates or compiler-recognizable remote operations, *e.g.*, XOR updates `a(i)[p]=XOR(a(i)[p],v)` in RandomAccess, *without* intervening synchronization. Since each remote fine-grain operation causes a network message, it is profitable to aggregate several of them into one coarser-grain network message that is expanded into several operations on the target image. A synchronization event completes all buffered remote operations.

The following piece of code is another example:

```
do i = 1, N
  a(i)[p] = a(i)[p] + 1
end do
```

The operations are fine-grain remote activities that can be aggregated (similar to communication vectorization) into a coarser-grain remote activity. The code can also be expressed in the vector form `a(1:N)[p]=a(1:N)[p]+1` and converted into a co-subroutine, if profitable.

9.5.2 Remote fine-grain operations/accesses aggregation

For irregular codes, vectorization and compile time aggregation might not be possible. However, the compiler can detect regions of code with a lot of unsynchronized fine-grain remote operations/accesses, *e.g.*, XOR updates in RandomAccess (see Section 9.6.3). These operations can be *aggregated* to increase the granularity of network messages. The compiler can instruct the run-time layer to aggregate these updates/operations to buffer them and deliver when the buffer gets full or a synchronization event happens.

It is also possible to perform aggregation of fine-grain remote operations by the compiler, rather than by the run-time layer. The compiler, knowing that the next segment of the code performs many fine-grain operations, can request buffers from run-time layer and generate aggregation code in the translated program, avoiding a function call to the run-time layer.

It is important to bundle the requests of similar type together, effectively compressing the remote request operation code such as +, -, /, *, XOR, PUT. When decoding an aggregated message, the run-time executes operations in a loop and applies them using the operation code and arguments.

Our DMT prototype implements run-time aggregation of fine-grain compiler-recognized remote operations: +,-,/,*,XOR, and PUT, which we used to experiment with fine-grain RandomAccess (see Section 9.6.3). Compiler support for automatic recognition of fine-grain operations was not implemented; instead our prototype implementation provides a special function that we used to indicate fine-grain operations to the runtime-layer: `call caf_op_i8(opcode,location,value,dest)`, where `opcode` is the code `op` of a fine-grain operation `x[dest]=x[dest] op value`, location is a co-array element of `integer(8)` type, `value` is the second operand, and `dest` is the target image.

9.5.3 Optimization of spawn

With the help of interprocedural analysis, the compiler may replace local `call`/`spawn`/`ship` or CF invocation with a regular Fortran 95/CAF `call` or function

invocation, avoiding the overhead of spawning the activity. For instance, if `spawn` is blocking and the spawnee does not have an explicit `reply` in its body (`reply` in this case coincides with the `return`), it is safe to replace `spawn` with Fortran 95 `call` because the spawner is blocked until the spawnee returns and the spawnee does not exist beyond the `call` point.

9.6 Experimental evaluation

We evaluate DMT using three codes: a micro-benchmark that computes the maximum value of a co-array section to show potential performance gains by shipping computation closer to data; a branch-and-bound implementation of a TSP solver to evaluate the benefits for parallel search applications; and several versions of the RandomAccess benchmark to stress the implementation and reveal limitations. All results are obtained on an Itanium2 cluster with a Myrinet 2000 interconnect (RTC) described in Section 3.3.

9.6.1 Maximum of a remote co-array section

Figure 9.10 (log-log-scale) presents the results for three codes that find a maximum value of a co-array section. Each curve *local*, *GET*, and *CF*, presents the normalized time to find the maximum value of a contiguous co-array section of N double precision numbers as a function of N. The curves are normalized to the *local* time, so *local* is a constant line 1.0. The *local* line corresponds to the time to compute the maximum locally: `res=mymaxval(a,n)`. The *GET* line shows the time to compute the maximum of a remote co-array section done by fetching the section and performing the computation locally: `res=mymaxval((a(1:n)[p]))`. It uses a temporary buffer to store the off-processor data used in local computation. The *CF* line presents the time to compute the maximum using a co-function to ship the `mymaxval` computation to data, rather than the data to

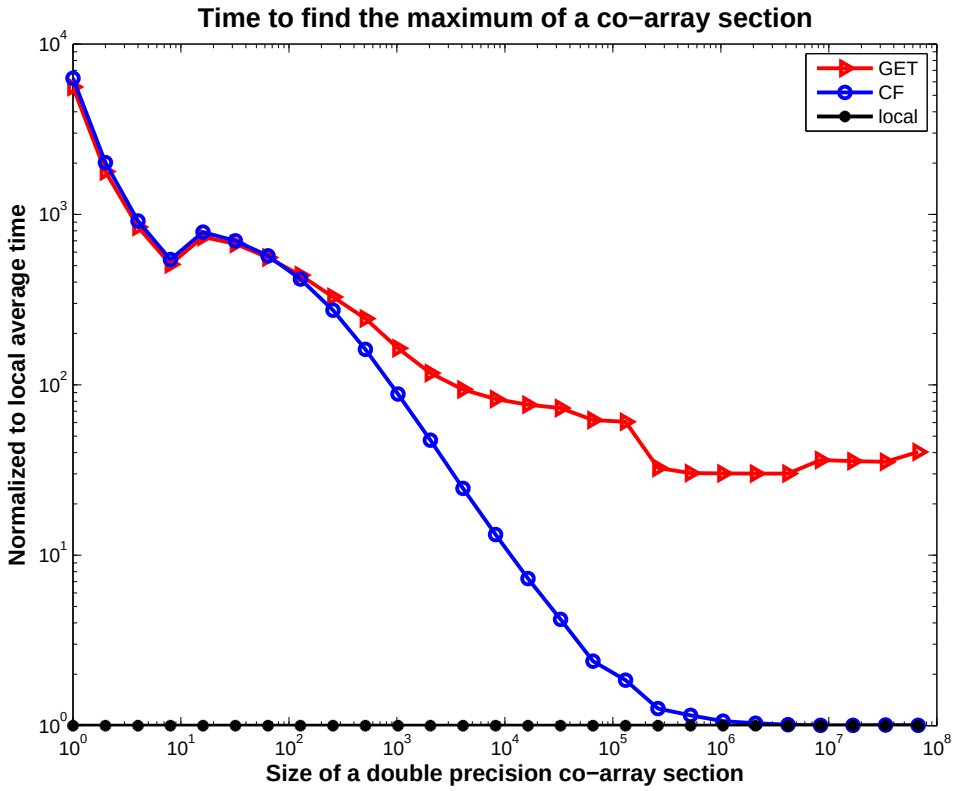


Figure 9.10 : Normalized (to local) time to find a maximum value of a co-array section.

computation: `res=cmymaxval(a,n)[p]`. The co-function is shown below.

```
cofunction cmymaxval(a, n)
  integer, intent(IN) :: n
  double precision :: a(n)[*]
  double precision :: cmymaxval, mymaxval

  cmymaxval = mymaxval(a, n)
end cofunction cmymaxval
```

The versions to compare are *GET* and *CF*; *local* is shown for completeness. For this reason, Figure 9.11 displays the same data normalized to the time of the *CF* version. Note that both local and remote computations are not interrupted by other computations. All versions use `mymaxval` to compute the maximum instead of Fortran 95 `maxval` intrinsic because implementation of the `maxval` intrinsic by the Intel Fortran compiler v9.0 delivered very poor performance.

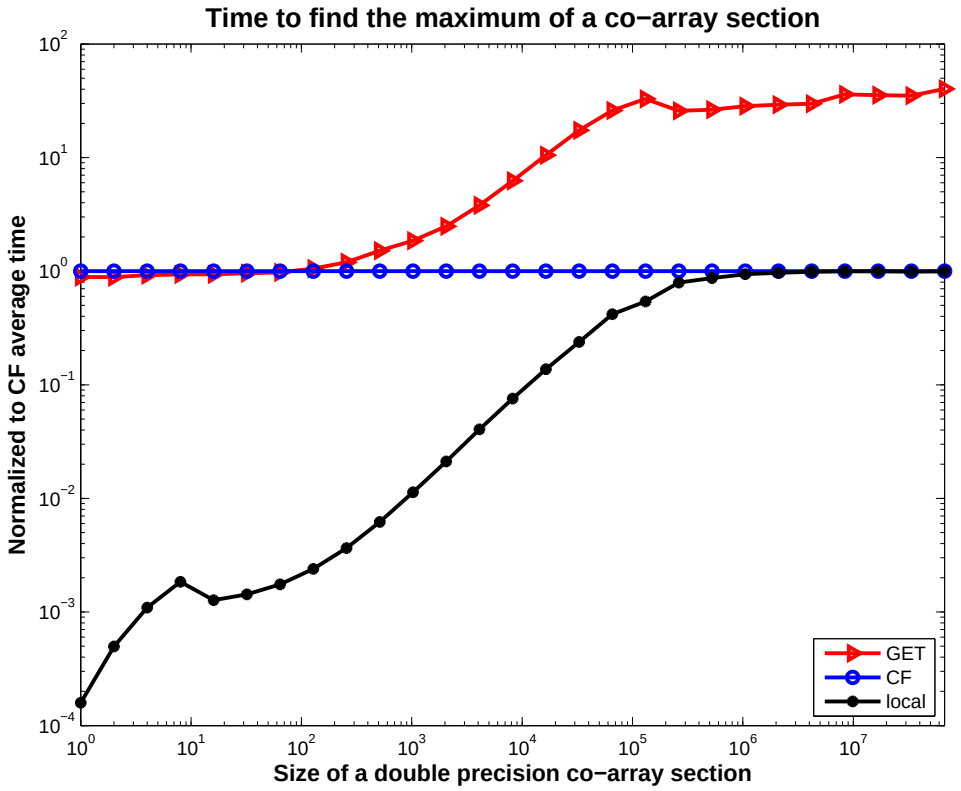


Figure 9.11 : Normalized (to CF) time to find a maximum value of a co-array section.

CF shows slightly lower, but comparable to *GET* performance for the array section size of up to 64 double precision numbers. The reason is that ARMCI GET is somewhat cheaper than heavier-weight ARMCI AM, and DMT adds more overhead because of parameter marshaling and extra bookkeeping. For array sections larger than 64, *CF* clearly outperforms *GET* by shipping computation to image p. For array sections of 256K double precision values, the performance of *CF* is within 25% of that of *local* and becomes almost identical for larger sections because computation dominates communication in *CF*. The *GET* version is 46% slower for 1024 section size and the performance gap gets much wider for larger section sizes because *GET* fetches a large amount of data over the network.

These results demonstrate the potential for function shipping on the example of a contiguous co-array section. It is expected that for more complex data structures, such as

linked lists, trees, queues, etc., function shipping would yield higher performance benefits when accessing remote portions of such data structures by hiding communication latency.

9.6.2 Traveling Salesman Problem

An important motivation for DMT is to simplify the development of parallel search applications without sacrificing performance. Many such applications are programmed using the master-slave paradigm. More efficient implementations (*e.g.*, using threads and polling) are possible, but they are harder to develop. To test whether it can be done easier with DMT without performance loss, we implemented a version of a parallel TSP solver that uses a branch-and-bound algorithm [104]. The algorithm finds the exact shortest cycle by performing the exhaustive search. To make it parallel, the search tree is cut at some level creating subproblems identified by a unique path prefix. The length of the shortest cycle found so far is used as the pruning criteria. Each image has its private copy of the shortest length that is eventually updated with the global best value. We implemented MPI and CAF versions of TSP; both use the same code to solve subproblems locally, but differ in how the subproblems are obtained and how the best length value is updated.

The MPI version uses a master-slave scheme. The master is responsible for generating subproblem prefixes, maintaining the global best length, and servicing requests from the clients. The clients obtain the best length and a subproblem from the master, solve the subproblem locally, and, if necessary, update the best length on the master. The master process does not perform "useful" computation (solve subproblems). To better utilize all available CPUs, it is necessary for the master process to share a processor with a client process. If a parallel machine's job scheduling subsystem does not allow an asymmetric number of processes per node, as is the case for our RTC cluster, such sharing is problematic. However, the application can be rewritten to run the master thread in one of the process images.

The CAF version does not reserve a process to be the master. It implements a centralized repository solution, which can be thought of as a one-sided master-slave scheme. One image is the repository image. It maintains the search space state and en-

```
cosubroutine getSubproblemFromRepository(prefix)
  ...
  integer, intent(out) :: prefix(1:n_cities)

  ! lock the search tree data structure
  call caf_lock(treeLock)
  ! execute local code to generate prefix
  call getSubproblem(prefix)
  ! unlock the search tree data structure
  call caf_unlock(treeLock)
end cosubroutine
```

Figure 9.12 : Co-subroutine to obtain a new subproblem path prefix.

sures its consistency. All images contact the repository process image `repository` when they run out of work and obtain new subproblem prefixes via spawning a CS `getSubproblemFromRepository`.

```
call getSubproblemFromRepository(prefix)[repository]
```

Figure 9.12 shows the `getSubproblemFromRepository` co-subroutine. The repository image application thread is doing useful work — solving subproblems. Repository image DMT helper threads, which are hidden from the programmer, execute activities that request new subproblems. These activities use a lock for mutually exclusive access to the search space data structures. When an image finds a shorter cycle, it propagates its length to all other images by spawning the CS `updateBestLength`, shown in Figure 9.13, per image using `ship` as shown in Figure 9.14[10].

Our MPI and CAF versions execute the same code to generate a new subproblem. However, in CAF, this logic is declared as a co-subroutine that returns the prefix to the spawning image. A second difference is how the shortest cycle length propagates to other images. In the MPI version, the best length is sent to the master first, then other images get the updated

[10]We could also introduce a `multiship` construct that would ship the same computation to several images; this is analogous to broadcast, but executes code rather than just communicates data.

```
cosubroutine updateBestLength(length)
  ...
  integer, intent(in) :: length

  ! lock the best length
  call caf_lock(bestLengthLock)
  ! update the best length
  if (length < bestLength) then
    bestLength = length
  end if
  ! unlock the best length
  call caf_unlock(bestLengthLock)
end cosubroutine
```

Figure 9.13 : Co-subroutine to update the best length.

```
! update best length on all images
if (myLength < newLength) then
  do i = 1, num_images()
    ship updateBestLength(newLength)
  end do
end if
```

Figure 9.14 : Code to update the best length.

value when they request a new subproblem. In DMT, when an image finds a shorter cycle, it spawns asynchronous activities to update the best length on all other images and continues execution without waiting for these activities to complete. The updates are propagated directly to other images bypassing the repository image[11]. Thus, the CAF version has the advantage that the pruning criteria is propagated faster throughout the system.

The parallel efficiency of two TSP instances is presented in Figures 9.15 and 9.16. Both instances find the shortest cycle for 18-city cliques where locations of the cities were randomly generated. The *TSP-MPI-1pe* and *TSP-MPI-2pe* curves show the performance of the MPI version runs with one and two processes per dual-processor node, respectively. *TSP-CAF-1pe* and *TSP-CAF-2pe* stand for the performance of the DMT CAF version with

[11] It is also possible to propagate the update in a tree-like fashion using $O(log(\texttt{num\_images}()))$ steps.

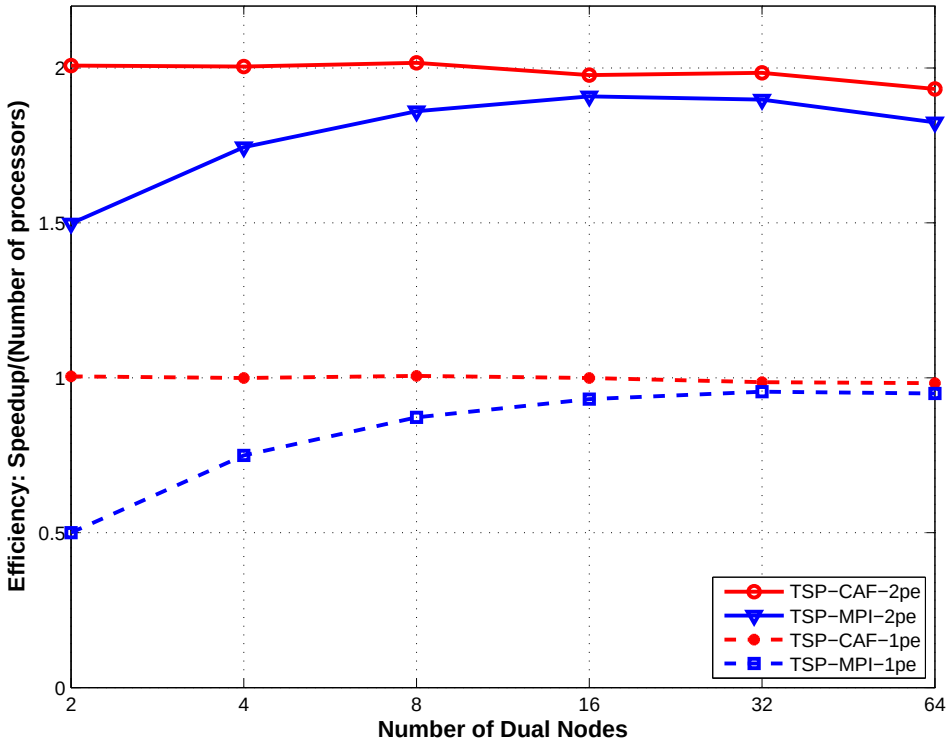


Figure 9.15 : Traveling salesman problem for 18 cities (seed=1).

one and two subproblem solving threads, called solvers, per node. For *TSP-CAF-2pe*, the second solver is an activity spawned (`ship_thread`) locally that executes the same code as the first solver. The second solver was necessary to utilize both CPUs of a dual-processor node because the available ARMCI implementation does not allow running two process images per node with a Myrinet 2000 adapter. The performance of versions with two solving threads per node is almost two times faster than that of the single-process-per-node versions for both MPI and CAF.

The performance of *TSP-CAF-1pe* is double than that of *TSP-MPI-1pe* for two nodes because *TSP-MPI-1pe* dedicates one node to be the master. *TSP-MPI-2pe* version has four processes on two nodes: one master and three workers, so its performance is only 25% lower than that of *TSP-CAF-2pe*, which uses all four CPUs. As the number of nodes increases, the performance of the MPI versions approaches that of the CAF version because

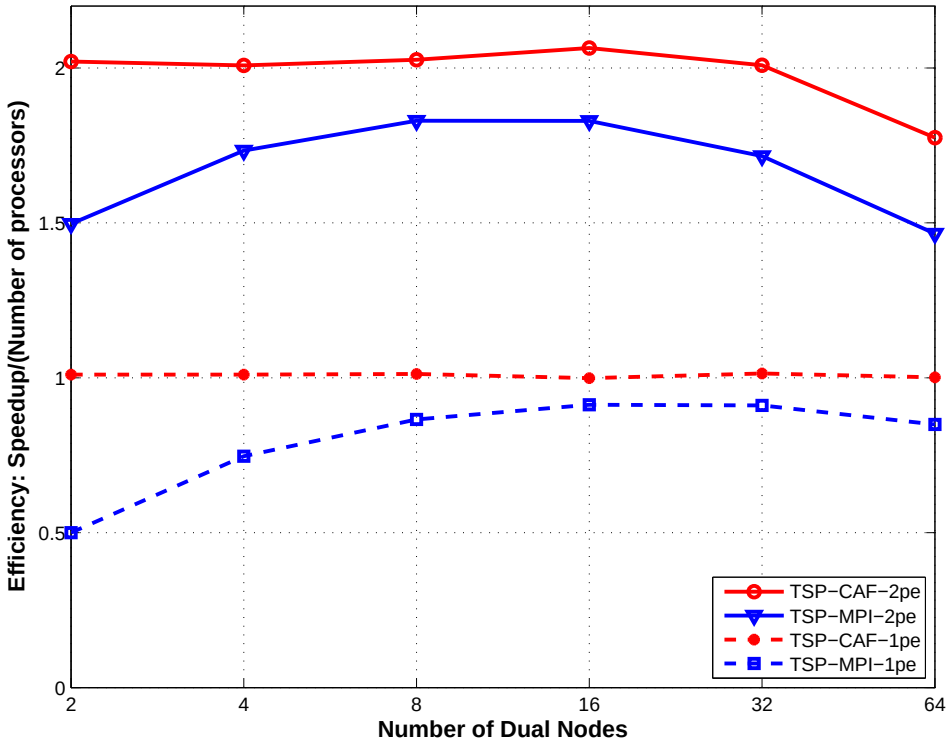


Figure 9.16 : Traveling salesman problem for 18 cities (seed=2).

the fraction of wasted CPU resources decreases. It is hard to quantitatively measure the contribution of faster shortest cycle length propagation for CAF versions. However, we did notice that the fraction of wasted CPU time due to the reserved master process is not exactly $\frac{1}{N}$, where N is the number of MPI processes, but slightly higher.

9.6.3 RandomAccess

The RandomAccess benchmark [1][12] presents a challenge for every existing architecture and programming model. It updates random locations in a huge table (see Section 3.4.3). The table is equally distributed among the nodes of a parallel machine. Each node generates a set of updates to random locations in the table. Each update involves performing an XOR

[12]We used the Table Toy Benchmark (08/01/97 version).

to adjust the value of a random location in the table. Errors in up to 1% of the table entries due to data races are allowed.

Rather than performing individual updates, the MPI version uses a bucketing strategy to bundle several updates that must be delivered to the same destination process. The code uses a set of buckets, one bucket for each MPI process. When a bucket is full, the process participates in an `MPI_AlltoAll` exchange, receiving updates from all other processes to apply to the local portion of the table. This process continues until all updates by all nodes are done.

We implemented several versions of RandomAccess using classic CAF and DMT CAF to stress our DMT implementation and to get a deeper understanding of performance issues. We compared the performance of our versions with that of the MPI bucketed version [1], the reference standard for the RandomAccess benchmark. All versions are either fine-grain or bucketed-based; we did not implement more sophisticated algorithms that perform aggregation and routing of updates.

Figure 9.17 shows the weak scaling in billion (10^9) of updates per second per dual-processor node of different CAF and MPI versions on the RTC cluster. The main table size is 512MB per node, the bucket size is 4KB per destination.

MPI bucketed versions

RA-MPI-1pe and *RA-MPI-2pe* are the MPI bucketed versions with one and two processes per node, respectively. They are used as the baselines for the comparison. *RA-MPI-2pe* shows two times better performance over *RA-MPI-1pe* for one node because it uses both CPUs of a dual-processor node. *RA-MPI-2pe* uses the table size of 256MB per process, totaling 512MB per node. *RA-MPI-2pe* scales worse than *RA-MPI-1pe* and does not achieve twice as high performance. The reason that neither MPI version scales well is the use of `MPI_AlltoAll` to exchange cached updates. While the RandomAccess random number generator is reasonably uniform, it does not fill limited-size buckets equally. This results in slightly larger than necessary data transfers because `MPI_AlltoAll` exchanges full

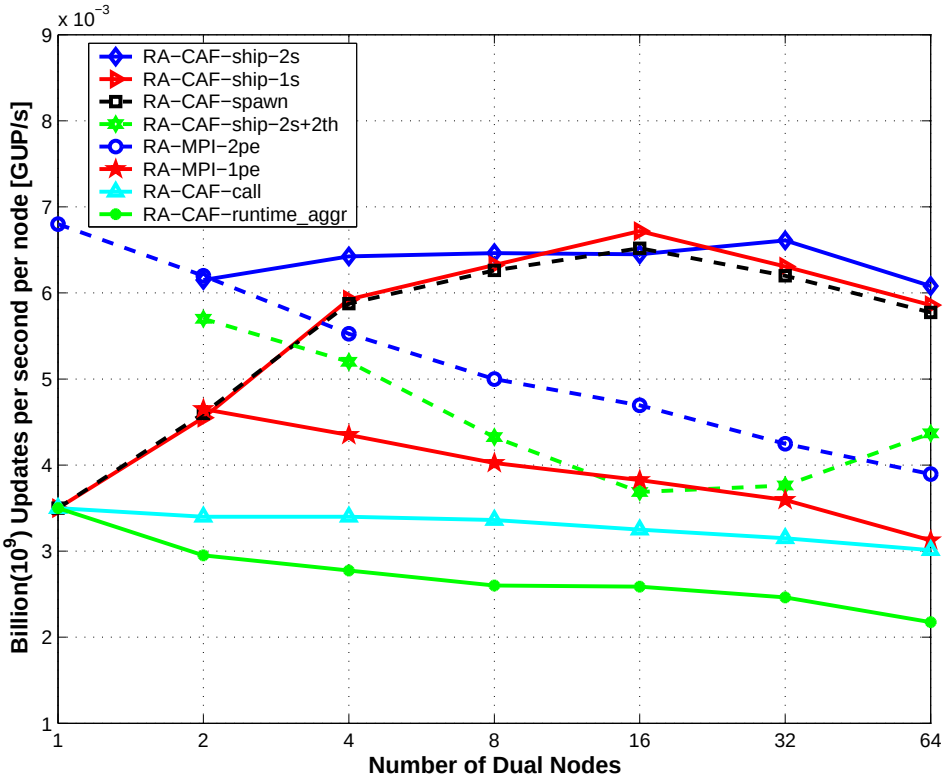


Figure 9.17 : RandomAccess with 512MB per node table and 4KB bucket size.

size buckets and some of them might not be filled entirely. Further, `MPI_AlltoAll`, being a collective call, enforces lock-step parallel exchanges; this precludes some processes waiting in `MPI_AlltoAll` for others to generate more updates for the next round. Consequently, running on twice the number of CPUs, *RA-MPI-2pe* scales worse than *RA-MPI-1pe* does.

Fine-grain CAF versions

CAF's one-sided model enables us to express a fine-grain version (see Section 3.4.3), in which each generated update is not cached locally but applied right away, *e.g.*,

```
table(loc)[p] = XOR(table(loc)[p], v)
```

`cafc` without DMT support compiles this code to a GET, XOR, and PUT, resulting in two exposed interconnect latencies per update. As expected, the performance is poor, and we do not show it in Figure 9.17. `cafc` with DMT support can compile this code to ship (`ship_am`) the XOR operation to the remote image. However, the performance is still poor, though slightly better than that of the version in classical CAF, because the run-time overhead for performing fine-grain remote operations is high. For this reason, we do not show the performance of this version in Figure 9.17.

Run-time aggregation in DMT CAF

`cafc` with DMT could recognize the remote XOR operation and instruct the run-time to aggregate it. Compiler support for the recognition of such operations is not yet implemented; however, we manually replaced the remote XOR update with a special run-time function to evaluate the performance of run-time aggregation. The run-time aggregates such operations into buffers, one per destination, to increase the granularity of the remote activity and to avoid sending many small network messages. This aggregation is analogous to how the MPI bucketed version caches XOR updates, but is done automatically by run-time layer without the need to modify the source code. The aggregation process stores the operation code (XOR) and arguments of each operation (its address in the remote memory and its XOR-value). It also compresses the operation code by bundling operations with the same code issued consecutively and storing the operation code only once per bundle[13] (see Section 9.5.2).

RA-CAF-runtime_aggr shows the performance of the fine-grain run-time-layer-aggregated version in Figure 9.17. It is roughly two times worse than that of *RA-MPI-1pe* version. However, it yields several orders of magnitude performance improvement compared to the non-aggregated fine-grain versions. Therefore, run-time aggregation may still be a valid technique for complex irregular codes where application-level aggregation is algorithmically hard. The performance degradation is caused by the overhead of a function

[13]This scheme is analogous to run-length encoding [124].

```
cosubroutine xorBucket(table, numUpdates, locations, values)
  ...
  ! the main table, tableSize is the size of the local part
  integer(8) :: table(0:tableSize-1)[*]
  ! number of updates to apply
  integer, intent(in) :: numUpdates
  ! locations in the table to update
  integer, intent(in) :: locations(numUpdates)
  ! values to XOR to the updated table locations
  integer(8), intent(in) :: values(numUpdates)

  do i = 1, numUpdates
    table(locations(i)) = XOR(table(locations(i)),values(i))
  end do
end cosubroutine
```

Figure 9.18 : Co-subroutine to apply XOR updates.

call, one for each update, to perform the run-time-layer aggregation. The execution of remote operations does not incur a function call per update. Instead, an aggregated packet is decoded on the destination and each XOR update is applied in a loop using the operation code and arguments. To avoid the aggregation function calls, `cafc` could ask the run-time layer to provide buffers and then generate code to perform the aggregation in the source code. However, we do not believe this strategy to be sufficiently general to justify a non-trivial implementation in `cafc`.

CAF version with blocking `spawn`

The other RA-CAF versions use buckets to cache remote XOR updates similar to that of the MPI bucketed version. When a bucket to a particular destination is full, the activity shown in Figure 9.18 is spawned to apply cached XOR updates in the remote image. The entire bucket is passed as an argument to the co-subroutine performing remote updates. An activity is initiated only when a bucket is completely full (except the very last bundle), which differentiates these versions from MPI, where partial buckets might be transferred. RA-CAF versions do not use collective communication and may achieve much better asynchrony

tolerance than MPI versions using `MPI_AlltoAll`. Local updates are applied as soon as they are generated. Each CAF version has thread(s) generating updates and thread(s) applying updates. Properly tuning the number of threads is key to good performance.

RA-CAF-call uses one thread per image to generate updates and blocking spawn to apply remote updates using `xorBucket`. Its performance is inferior to the other bucketed RA-CAF versions because the blocking spawn exposes not only the network latency to spawn the activity but also the latency to execute the activity in the target image. The latter latency includes not only the time to apply XOR updates, but also the time waiting to be scheduled: activities from all other nodes compete for execution scheduling within the target image. While waiting for the blocking spawn to complete, the local application thread does not perform useful computation such as generating new XOR updates, initiating other remote activities, or applying updates received from other images. Because of this wasted time, *RA-CAF-call* performs poorly.

CAF version with non-blocking `spawn`

RA-CAF-spawn hides the exposed latency by using non-blocking AM-mode spawns (`spawn_am`) to apply remote updates. When a bucket b_p for image `p` becomes full for the first time, a non-blocking activity is spawned to apply b_p updates in `p`. Meanwhile, the current image continues to generate updates. It can reuse b_p because spawn copies b_p values (table locations and XOR values) into a run-time layer buffer. *RA-CAF-spawn* is coded using one spawn handle per destination, which enables one in flight spawn per target image. When b_p becomes full again, the image waits for the completion of the previous non-blocking spawn destined to `p` (to reuse the spawn handle variables), which is likely to be completed by this time. Thus, the *RA-CAF-spawn* version hides the latency of `spawn` by overlapping it with computation and enjoys much better performance than *RA-CAF-call*. In fact, the performance is almost twice as good as that of the *RA-MPI-2pe* version because non-blocking spawns enable better asynchrony tolerance — each image generates updates more independently from what the other images are doing.

The DMT runtime is configured to have two run-time threads, an application thread and a pool thread, providing exactly one thread per CPU. Whenever the application thread executes `spawn_am` and there are pending activities from the other images, it helps the pool thread to process these activities; this maintains the balance between the speeds of generation and application of XOR updates. The performance of the benchmark is primarily bound by the TLB performance because each update is likely to cause a TLB miss. Having two threads applying XOR updates enables to utilize both TLB units of the node.

While delivering good performance, *RA-CAF-spawn* version is harder to implement than *RA-CAF-call* because the programmer must explicitly manage the spawn handles.

CAF versions with `ship`

RA-CAF-ship-1s is similar to *RA-CAF-call*, but uses AM-mode ship (`ship_am`) to apply remote updates. The programmer gets both high performance of the *RA-CAF-spawn* version and simplicity of the *RA-CAF-call* version. The performance of *RA-CAF-ship-1s* is slightly higher than that of *RA-CAF-spawn* because (1) `ship` is a non-blocking spawn without reply and (2) more than one activity can be spawned on the target image from the same origin image; in comparison, *RA-CAF-spawn* can spawn only one activity because it reuses handle variables. At the same time, *RA-CAF-ship-1s* is as simple to code as *RA-CAF-call* because the programmer does not manage explicit `spawn` handles.

While the performance of *RA-CAF-spawn* and *RA-CAF-ship-1s* exceeds that of *RA-MPI-2pe* for runs on four and more nodes, it is lower for one- and two-node runs. The reason is that both CAF versions have only one application thread per node generating XOR updates, while *RA-MPI-2pe* has two such threads. For small number of nodes, the amount of work done by application threads is greater than that done by pool threads applying XOR updates. In fact, the pool thread does not perform any work for executions on one node since the application thread applies local updates and there are no remote updates. Thus, the pool thread is underutilized resulting in lower overall performance.

Since the current ARMCI implementation does not allow two CAF process images per

cluster node for a Myrinet 2000 interconnect, we created the *RA-CAF-ship-2s* version to evaluate the effect of having two application threads. *RA-CAF-ship-2s* is a version based on *RA-CAF-ship-1s*, but it runs two application threads, called solvers, generating (and applying) XOR updates. Each solver executes half the total XOR updates per image and uses its own set of buckets. The second solver is spawned locally using `ship_thread`. We set the thread pool size to be zero to have exactly one run-time thread per CPU. The solver threads spawn activities in AM-mode (`ship_am`) and execute them.

The performance of *RA-CAF-ship-2s* is virtually the same as that of the *RA-MPI-2pe* version on two nodes and exceeds it for runs on larger number of nodes. *RA-CAF-ship-2s* shows a bit better performance than that of *RA-CAF-spawn* and *RA-CAF-ship-1s* at the expense of slightly more complicated code to have the second application thread, which would not be necessary with the two-image-per-node configuration.

To estimate the effect of "wrong" number of threads per image, we evaluated the *RA-CAF-ship-2s+2th* version based on *RA-CAF-ship-2s*. *RA-CAF-ship-2s+2th* runs four run-time threads per node: two solver threads and two pool threads. The performance of *RA-CAF-ship-2s+2th* is much worse than *RA-CAF-ship-2s* because the application threads that generate XOR updates compete for CPUs with the pool threads that only execute remote activities; our understanding is that this results in *bursty*, and overall lower, update generation speed.

Note that *RA-MPI-2pe* uses 256MB tables per process (512MB per node), while all other versions use 512MB tables per process. However, this did not give much advantage to *RA-MPI-2pe* because the TLB performance for random updates of 512MB and 256MB tables is roughly the same.

The presented RA-CAF versions were used to evaluate the DMT prototype implementation. DMT enabled a simpler implementation of the RandomAccess benchmark and demonstrated better performance than MPI for up to 128-CPU runs. However, the bucket-based approach might not scale well on very large scale clusters, such as IBM Blue Gene/L with 128K processors and small memory nodes, because the bucket size would decrease as

the number of processors increases. This would result in a lot of small network messages carrying remote updates. A different, software-routing algorithm is necessary to deliver the best RandomAccess performance on large systems.

9.7 Discussion

We argued for the need of function shipping and multithreading in the CAF parallel programming model and described potential applications of distributed multithreading in scientific codes. We evaluated DMT design principles and presented a DMT specification for CAF. Our prototype implementation of DMT in `cafc` enabled us to evaluate the benefits of function shipping for programmability and performance.

DMT improves programmability of applications that benefit from asynchronous activities. DMT makes one-sided access to remote parts of complex data structures practical without the need to implement a two-sided master-slave scheme; this directly benefits programmability and may benefit performance of parallel search applications. Our branch-and-bound TSP implementation in DMT CAF is simpler than a master-slave message-passing implementation in MPI. The simplicity comes from not having to implement a two-sided protocol when using DMT; instead, the programmer can use co-functions to execute asynchronous remote activities. This is more intuitive than message passing and very much resembles using regular function calls. DMT-based TSP demonstrates better performance because the MPI implementation dedicates a processor to be the master, which does not perform useful computation.

Our micro-benchmark to compute the maximum value of a co-array section enabled us to quantify the performance gain due to co-locating computation with data. As expected, the benefit increases as the size of a remote co-array section gets larger. For large sections, it is up to 40 times faster to ship computation and get the result back than fetch data and obtain the result locally. We expect this benefit to be even higher for more complex data structures such as remote linked lists, queues, etc.

Performance and scalability of DMT-based RandomAccess exceeds that of the MPI

bucketed version due to better asynchrony tolerance. MPI version uses `MPI_AllToAll` reduction, which delays processes to wait for the slowest, resulting in poor scalability and low performance. DMT-based version uses asynchronous remote activities and does not synchronize with other processes. However, it is important to configure DMT to use the right number of run-time threads and to load-balance the application to obtain best performance. Currently, DMT leaves these tasks to the programmer; however, it provides three types of spawns and the ability to control the number of run-time threads.

It would be interesting to consider whether it is possible to use DMT to perform automatic load-balancing on distributed memory machines for a large class of applications. Another promising research direction is to improve OS thread support to enable applications, rather than the OS, to schedule threads; this will provide better control over scheduling of concurrent activities and user-defined scheduling policies in DMT and other emerging multithreaded languages. Finally, it would be interesting to investigate in detail possible compiler optimizations for function shipping and local multithreading; for example, selecting the most appropriate spawn type, aggregating fine-grain remote activities, and scheduling activities to reduce interference (*e.g.*, scheduling concurrently CPU-bound and memory-bound activities within a multi-core multiprocessor node).

Chapter 10

Conclusions and Future Directions

The quest to find a parallel programming model that is ubiquitous, expressive, easy to use, and capable of delivering high performance is a difficult one. The Message Passing Interface (MPI) remains the *de facto* parallel programming model today despite a huge effort to find alternatives that are easier to use. In this dissertation, we principally explored the design and implementation of Co-array Fortran (CAF) as a representative of the emerging Partitioned Global Address Space (PGAS) languages, which also include Unified Parallel C (UPC) and Titanium.

10.1 Contributions

The primary contributions of this dissertation include:

- design and implementation of `cafc`, the first multi-platform CAF compiler for distributed and shared-memory machines (joint work with Cristian Coarfa),
- performance studies to evaluate the CAF and UPC programming models (joint work with Cristian Coarfa),
- design, implementation, and evaluation of new language features for CAF, including communication topologies, multi-version variables, and distributed multithreading,
- a novel technique to analyze explicitly-parallel SPMD programs that facilitates optimization, and
- a synchronization strength reduction transformation for automatically replacing barrier-based synchronization with more efficient point-to-point synchronization.

Our joint studies show that CAF programs can achieve the same level of performance and scalability as equivalent MPI codes; however, development of high performance codes in CAF is as difficult as when using MPI.

In this dissertation, I show that extending CAF with language-level communication topologies, multi-version variables, and distributed multithreading will increase programmers' productivity by simplifying the development of high performance codes.

10.1.1 Design, implementation, and performance evaluation of `cafc`

We designed and implemented `cafc`, the first multi-platform CAF compiler for distributed and shared-memory architectures. `cafc` is a widely portable source-to-source translator. By performing source-to-source translation, `cafc` can leverage the best Fortran 95 compiler available on the target architecture to compile translated programs, and the ARMCI and GASNet communication libraries to support systems with a range of interconnect fabrics, including Myrinet, Quadrics, and shared memory.

We ported many parallel benchmarks into CAF and performed extensive evaluation studies [30, 47, 48, 31, 32, 33] to investigate the quality of the CAF programming model and its ability to deliver high performance. An important result of our studies is that CAF codes compiled with `cafc` can match the performance and scalability of their MPI counterparts. We identified three classes of performance impediments that initially precluded CAF codes from achieving the same level of performance and scalability as that of their MPI counterparts. They include scalar performance of a translated program, communication efficiency, and synchronization.

Scalar performance. We found that source-to-source translation of co-arrays introduces apparent aliasing in the translated program due to `cafc`'s representation of co-arrays via implicit shape arrays. This hinders the platform's Fortran 95 compiler to efficiently optimize code accessing local co-array data. We developed a *procedure splitting transformation* that converts each procedure s referencing `COMMON` and `SAVE` co-array local data

into two subroutines s_1 and s_2. s_1 resembles s, but instead of performing computation, it calls s_2 and passes the co-arrays as arguments. s_2 performs the original computation in which each `COMMON` and `SAVE` co-array reference is converted into a reference to the corresponding co-array parameter. In `cafc`, co-array arguments are represented via explicit shape subroutine dummy arguments, which do not alias in Fortran 95. As a result, the lack of aliasing among `COMMON` and `SAVE` co-arrays, their bounds and contiguity are conveyed to the Fortran 95 compiler. The procedure splitting transformation implemented in `cafc` enables a translated program to achieve the same level of scalar performance as an equivalent Fortran 95 program that uses `COMMON` and `SAVE` variables.

Communication efficiency. Our experiments showed that it is imperative to vectorize and/or aggregate communication on distributed memory machines to deliver performance and scalability; without coarse-grain communication, the performance is abysmal on cluster architectures. For strided data transfers, it is also important to pack the data at the source and unpack it on the destination to achieve the best communication efficiency. Fortunately, CAF enables source-level communication vectorization, aggregation, and packing/unpacking. With CAF, one can get high performance *today* rather than wait for a mature implementation of a vectorizing CAF compiler. However, automatic compiler transformations such as communication vectorization and aggregation, studied by Cristian Coarfa [29], will be important to broaden the class of CAF programs that can achieve high performance and to improve the performance portability of CAF programs across a range of architectures.

Synchronization. The burden that PGAS languages impose on programmers is the need to synchronize shared one-sided data access. We observed that using barriers for synchronization was much simpler than using point-to-point synchronization, which is painstaking and error-prone. However, point-to-point synchronization may provide much better scalability; we observed up to a 51% performance improvement for the NAS CG benchmark (14000 size) for a 64-processor execution.

We observed that using extra communication buffers can remove from the critical path anti-dependence synchronization due to buffer reuse. This yielded up to a 12% performance improvement for the ASCI Sweep3D benchmark (150x150x150 size) as compared to the standard MPI version. However, coding such multi-buffer solutions is difficult due to the need for explicit buffer management and complex point-to-point synchronization.

CAF and UPC. We also compared CAF with UPC and found that it is easier to match MPI's performance with CAF for regular scientific codes. We attribute this to the more explicit nature of communication in CAF and language-level support for multi-dimensional arrays.

10.1.2 Enhanced language, compiler, and runtime technology for CAF

Co-spaces: communication topologies for CAF. We found that CAF's multi-dimensional co-shape is not convenient and expressive enough to be useful for organizing parallel computation. It does not provide support for process groups, group communication topologies, nor expression of communication partners relative to the process image. Instead, programmers often use Fortran 95 arrays and integer arithmetic to represent communication partners. Such ad hoc methods of structuring parallel computation render CAF impenetrable to compiler analysis.

We explored replacing CAF's multi-dimensional co-shapes with more expressive communication topologies, called co-spaces, such as group, Cartesian, and graph. They simplify programming by providing convenient abstractions for organizing parallel computations. Group co-space enables support for process groups as well as remapping process image indices. Cartesian or graph co-spaces are used to impose a Cartesian or graph communication topology on a group; they provide functionality to systematically specify the targets of communication and point-to-point synchronization. These abstractions, in turn, expose the structure of communication to the compiler, facilitating compiler analysis and optimization.

Communication analysis. We devised a novel technology for analyzing *explicitly-parallel* CAF programs suitable for a large class of scientific applications with structured communication. When parallel computation is expressed via a combination of a co-space, textual co-space barriers, and co-space single-valued expressions, the CAF compiler can infer communication patterns from explicitly-parallel code. As of this writing, communication analysis is limited to a procedure scope with structured control flow. Our work focuses on two patterns that are common for nearest-neighbor scientific codes. The first pattern is a group-executable PUT/GET in which the target image is expressed via a co-space interface neighbor function with co-space single-valued arguments. The second is a non-group-executable PUT/GET with the target image expressed via a co-space interface neighbor function with co-space single-valued arguments. Knowing the communication pattern for each process image of the co-space enables determination of the origin image(s) of communication locally. This is a fundamental enabling analysis for powerful communication and synchronization optimizations such as synchronization strength reduction.

Synchronization strength reduction. We developed a procedure-scope synchronization strength reduction (SSR) optimization that replaces textual co-space barriers with asymptotically more efficient point-to-point synchronization where legal and profitable. This transformation is both difficult and error-prone for application developers to exploit manually at the source code level. SSR optimizes the communication patterns inferred by our analysis of communication partners. As of this writing, it operates on a procedure scope with a single co-space and textual co-space barriers for synchronization. To extend SSR's applicability to real codes, we use compiler hints to compensate for the lack of interprocedural analysis. Understanding communication structure enables the CAF compiler to convert barrier-based synchronization into more efficient form. We investigated the conversion of textual co-space barriers into point-to-point synchronization. SSR-optimized programs are more asynchrony tolerant and show better scalability and higher performance than their barrier-based counterparts.

We implemented prototype support for SSR in `cafc`. SSR-optimized Jacobi iteration, NAS MG, and NAS CG benchmarks show performance comparable to that of our fastest hand-optimized versions that use point-to-point synchronization. Compared to their barrier-based counterparts, they demonstrate noticeable performance improvements. For 64-processor executions on an Itanium2 cluster with a Myrinet 2000 interconnect, we observed run-time improvements of 16% for a 2D Jacobi iteration of 1024^2 size, 18% for NAS MG classes A and B, and 51% for NAS CG class A. In our prior studies, we observed similar benefits from using point-to-point synchronization instead of barriers on other parallel platforms and for other benchmarks as well.

Multi-version variables. Many scientific codes such as wavefront, line-sweep, and loosely-coupled parallel applications exhibit the producer-consumer communication pattern, in which the producer(s) sends a stream of values to the consumer(s). Expressing high performance producer-consumer communication in PGAS languages is difficult. The programmer has to explicitly manage several communication buffers, orchestrate complex point-to-point synchronization (to hide the latency of anti-dependence synchronization due to buffer reuse), and use non-blocking communication.

We explored extending CAF with multi-version variables (MVVs), a language-level abstraction we devised to simplify the development of high performance codes with producer-consumer communication. An MVV can store more than one value. Only one value can be accessed at a time; others are queued by the runtime. A producer commits new values into an MVV and a consumer retrieves them. MVVs offer limited support for two-sided communication in CAF, which is a natural choice when developing producer-consumer codes. MVVs simplify program development by insulating the programmer from the details of buffer management, complex point-to-point synchronization, and non-blocking communication.

MVVs are the right abstraction for codes in which each process communicates streams of values to a small subset of processors. MVVs might not be the best abstraction for

codes in which each process communicates data to a lot of processes, which might cause excessive MVV buffering. While MVVs insulate the programmer from managing the anti-dependence synchronization, sometimes no such synchronization is necessary because it is enforced elsewhere in the application. However, we believe that programmability benefits of the MVV abstraction outweigh slight performance losses due to unnecessary anti-dependence synchronization in this case.

We extended CAF with prototype support for MVVs. MVVs significantly simplify development of wavefront applications such as Sweep3D, and MVV-based codes deliver performance comparable to that of the fastest CAF multi-buffer hand-optimized versions, up to 39% better than that of CAF one-buffer versions, and comparable to or better (up to 12%) than that of their MPI counterparts on a range of parallel architectures. MVVs greatly simplify coding of line-sweep applications, such as the NAS BT and SP benchmarks, and deliver performance comparable to that of the best hand-optimized MPI and CAF versions.

Distributed multithreading. Distributed memory is necessary for the scalability of massively parallel systems [80]. Systems in which memory is co-located with processors continue to dominate the architecture landscape. The nodes of these distributed memory architectures are also becoming parallel, *e.g.*, multi-core multiprocessors. Distributed multithreading (DMT) is based on the concepts of function shipping and multithreading, which provide two benefits. First, DMT enables co-locating computation with data. Second, it enables exploiting hardware threads available within a node. DMT uses *co-subroutines* and *co-functions* to co-locate computation with data and to enable local and remote asynchronous activities. Using DMT to co-locate computation with data is an effective way of avoiding exposed latency, especially when performing complex operations on remote data structures. In addition, concurrent activities running within a node would enable utilizing available hardware parallelism.

We presented design principles behind multithreading in an SPMD language and provided the DMT specification for CAF, featuring blocking and non-blocking activities that

can be spawned remotely or locally. We extended `cafc` with prototype support for DMT. We developed a micro-benchmark to compute the maximum value of a co-array section to quantify the performance gain due to co-locating computation with data. In our experiments on an Itanium2 cluster with a Myrinet 2000 interconnect, we observed that, for large sections, it is up to 40 times faster to ship computation and get the result back than fetch data and obtain the result locally; for accesses to more complex remote data structures, this benefit is likely to be much higher. We developed several fine-grain and bucketed versions of the RandomAccess benchmark to gain a better understanding for DMT design. Our experimentation revealed that it is necessary to use a pool of OS threads to execute activities, rather than to spawn each activity in a separate OS thread, to deliver best performance; it is also necessary to allow programmers to control the thread pool to tune the runtime for the application's concurrency needs. Better asynchrony tolerance allowed the performance of a DMT-based implementation of bucketed RandomAccess to exceed that of the standard MPI version, which uses `MPI_AlltoAll` to exchange remote updates.

We found that DMT improves programmability of applications that benefit from asynchronous activities. We experimented with a branch-and-bound traveling salesman problem (TSP), which we selected as representative of parallel search applications. We found that the DMT-based CAF version is simpler than a master-slave message-passing implementation in MPI. The simplicity comes from not having to implement a two-sided protocol when using DMT; instead, the programmer can use co-functions to execute asynchronous remote activities. DMT-based TSP demonstrates better performance, because, in our experiments, the MPI implementation dedicates a processor to be the master, and this mater processor does not perform useful computation.

10.2 Future Directions

New technology and infrastructure developed in this dissertation will enable us to investigate a set of interesting ideas in the future. We outline a few promising research directions.

Extending CAF analysis and communication/synchronization optimization. To develop scalable, high performance explicitly-parallel programs, programmers must use efficient communication and orchestrate complex point-to-point synchronization, which is difficult. Barriers are the simplest synchronization mechanism to use in PGAS languages. Thus, the role of the compiler is to enable application developers to use barriers for synchronization, while optimizing communication and synchronization into a more efficient form delivering performance and scalability. SSR is an example of such an optimization.

As of this writing, our novel CAF analysis and SSR are limited to a procedure scope with single co-space and structured control flow. It is possible to extend the analysis to handle arbitrary control flow (see discussion in Section 7.9). There is also a good indication that interprocedural analysis can be developed to eliminate the necessity of hints for SSR. Such analysis would include: (1) detecting whether a procedure may access local or remote co-arrays or perform synchronization in any invocation; (2) propagation of single values across procedure calls; (3) propagation of unsynchronized PUT/GET across procedure boundaries. It is still an open question whether an analysis can be developed to analyze scopes where communication/synchronization is done for multiple co-spaces.

In addition to SSR, our CAF analysis technology enables a set of promising communication and synchronization optimizations. SSR does not change the communication primitive. Doing so will enable conversion of one-sided PUT/GET communication into two-sided send and receive. Such two-sided communication can be buffered, and would enable us to automatically generate more asynchrony tolerant code, since buffering can move anti-dependence synchronization off the critical path, and packing/unpacking of strided communication. Conversion of GET into PUT will enable us to utilize interconnect RDMA capabilities, when accessing remote data via PUTs, for architectures with RDMA support for PUTs, but not for GETs. The push (PUTs) strategy would also enable us to hide exposed latency inherent to the pull (GETs) strategy as well as to tile producer-consumer loop nests to entirely hide communication latency.

Finally, our SSR algorithm is not based on array section dependence analysis. Devel-

oping such an analysis, which must also include remote co-array sections, might improve the precision of our CAF analysis and SSR; however, we have not yet seen opportunities that would benefit from such analysis in the limited set of codes we have studied.

Enhancing multi-version variables and beyond. Producer-consumer communication is typical in many scientific codes; however, it is difficult to develop scalable, high performance producer-consumer applications in PGAS languages. We offer MVVs as a pragmatic and convenient way to simplify development of high-performance producer-consumer codes in CAF.

It would be interesting to consider whether multi-version variables can benefit from extensions such as GET-style remote `retrieve`, the `commit` and `retrieve` primitives of partial MVV versions, and an adaptive buffer management strategy.

It is worth investigating the stream abstraction as an alternative to MVVs, especially for codes that stream values of unequal size. While streams are a more general abstraction than MVVs, they would require the programmer to establish explicit connections. For streams, it would also be harder to optimize unnecessary memory copies, which MVVs achieve via adjusting an F90 pointer.

The clocked final model (CF) [106] is another more general alternative to MVVs that does not require the programmer to specify the number of buffers and explicitly manage `commit`s and `retrieve`s. It would be interesting to investigate whether it is possible to develop sophisticated compiler and runtime technology to optimize CF-based scientific codes to deliver as high performance as that of using MVVs on a range of parallel architectures.

Improving thread support in programming languages. Co-locating computation with data and utilizing intra-node parallelism is essential to fully utilize hardware capabilities of modern parallel architectures. While experimenting with distributed multithreading, we discovered that operating systems do not provide adequate support for precisely controlling multithreading for high performance codes. A promising research direction is to work

with OS developers to develop an efficient, flexible, and portable threading system that enables applications, rather than the OS, to schedule threads. This would enable us to extend a multithreaded programming model with user-defined scheduling policies that best accommodate the concurrency needs of the application, as well as compiler analysis and optimization to appropriately mix & schedule concurrent computations. Better run-time support would also be necessary to enable massive (millions of threads) multithreading within a node.

Finally, it is worth investigating whether a programming model can provide convenient abstractions for efficient work-sharing that can be optimized for automatic load-balancing in the presence of distributed memory.

Bibliography

[1] HPC Challenge Benchmarks. http://www.hpcchallenge.org.

[2] Sisal language tutorial. http://tamanoir.ece.uci.edu/projects/sisal/sisaltutorial/01.Introduction.html.

[3] The SUIF compiler system. http://suif.stanford.edu/suif.

[4] Accelerated Strategic Computing Initiative. The ASCI Sweep3D benchmark code. http://www.llnl.gov/asci_benchmarks/asci/limited/sweep3d/asci_sweep3d.html, 1995.

[5] J. C. Adams, W. S. Brainerd, J. T. Martin, B. T. Smith, and J. L. Wagener. *Fortran 95 handbook: complete ISO/ANSI reference*. The MIT Press, Cambridge, MA, 1997.

[6] A. Aiken and D. Gay. Barrier inference. In *Proceedings of the 25th Annual ACM SIGPLAN Symposium on Principles of Programming Languages*, pages 342–354, San Diego, CA, Jan. 1998.

[7] E. Allen, D. Chase, J. Hallett, V. Luchangco, J.-W. Maessen, S. Ryu, G. L. S. Jr., and SamTobin-Hochstadt. The Fortress language specification, v1.0α. http://research.sun.com/projects/plrg/fortress.pdf, Sept. 2006.

[8] R. Allen and K. Kennedy. *Optimizing Compilers for Modern Architectures*. Morgan Kaufmann Publishers, San Fransisco, CA, 2002.

[9] ANSI. *Myrinet-on-VME Protocol Specification (ANSI/VITA 26-1998)*. American National Standard Institute, Washington, DC, 1998.

[10] J. Aspnes, M. Herlihy, and N. Shavit. Counting networks. *Journal of the ACM*, 41(5):1020–1048, Sept. 1994.

[11] R. Bagrodia and V. Austel. UC user manual, v1.4. `http://pcl.cs.ucla.edu/projects/uc/uc-manual.ps`.

[12] D. Bailey, T. Harris, W. Saphir, R. van der Wijngaart, A. Woo, and M. Yarrow. The NAS parallel benchmarks, v2.0. Technical Report NAS-95-020, NASA Ames Research Center, Moffett Field, CA, Dec. 1995.

[13] A. D. Birrell and B. J. Nelson. Implementing remote procedure calls. *ACM Transactions on Computer Systems*, pages 39–59, Feb. 1984.

[14] W. Blume *et al*. Polaris: The next generation in parallelizing compilers,. In *Proceedings of the 7th Workshop on Languages and Compilers for Parallel Computing*, Ithaca, NY, Aug. 1994.

[15] R. Blumofe and C. Leiserson. Scheduling multithreaded computations by work stealing. In *Proceedings of the 35th Annual Symposium on Foundations of Computer Science*, pages 356–368, Santa Fe, NM, Nov. 1994.

[16] O. A. R. Board. The OpenMP Application Program Interface (API) v2.5. `http://www.openmp.org/drupal/mp-documents/spec25.pdf`, May 2005.

[17] D. Bonachea. GASNet specification, v1.1. Technical Report CSD-02-1207, University of California at Berkeley, Berkeley, CA, Oct. 2002.

[18] D. Bonachea. Proposal for extending the UPC memory copy library functions and supporting extensions to GASNet, v1.0. Technical Report LBNL-56495, Lawrence Berkeley National Laboratory, Berkeley, CA, Oct. 2004.

[19] Z. Bozkus, L. Meadows, S. Nakamoto, V. Schuster, and M. Young. PGHPF – an optimizing High Performance Fortran compiler for distributed memory machines. *Scientific Programming*, 6(1):29–40, 1997.

[20] D. C. Cann. The optimizing SISAL compiler. Technical Report UCRL-MA-110080, Lawrence Livermore National Laboratory, Livermore, CA, Apr. 1992.

[21] F. Cantonnet, Y. Yao, S. Annareddy, A. Mohamed, and T. El-Ghazawi. Performance monitoring and evaluation of a UPC implementation on a NUMA architecture. In *Proceedings of the International Parallel and Distributed Processing Symposium*, Nice, France, Apr. 2003.

[22] B. L. Chamberlain, S. J. Deitz, and L. Snyder. A comparative study of the NAS MG benchmark across parallel languages and architectures. In *Proceedings of Supercomputing*, Dallas, TX, Nov. 2000.

[23] B. Chapman, P. Mehrotra, and H. P. Zima. Enhancing OpenMP with features for locality control. In *Proceedings of the 8th ECMWF Workshop on the Use of Parallel Processors in Meteorology "Towards Teracomputing"*, pages 301–313, Reading, United Kingdom, Nov. 1998.

[24] D. Chavarría-Miranda. *Advanced Data-Parallel Compilation*. PhD thesis, Rice University, Houston, TX, Dec. 2003.

[25] D. Chavarría-Miranda and J. Mellor-Crummey. An evaluation of data-parallel compiler support for line-sweep applications. In *Proceedings of the 11th International Conference on Parallel Architectures and Compilation Techniques*, Charlottesville, VA, Sept. 2002.

[26] D. Chavarría-Miranda and J. Mellor-Crummey. An evaluation of data-parallel compiler support for line-sweep applications. *The Journal of Instruction-Level Parallelism*, 5, Feb. 2003. (`http://www.jilp.org/vol5`). Special issue with selected papers from: The Eleventh International Conference on Parallel Architectures and Compilation Techniques, September 2002. Guest Editors: Erik Altman and Sally McKee.

[27] D. Chavarría-Miranda, J. Mellor-Crummey, and T. Sarang. Data-parallel compiler support for multipartitioning. In *European Conference on Parallel Computing*, Manchester, United Kingdom, Aug. 2001.

[28] W. Chen, D. Bonachea, J. Duell, P. Husbands, C. Iancu, and K. Yelick. A performance analysis of the Berkeley UPC compiler. In *Proceedings of the 17th ACM International Conference on Supercomputing*, San Francisco, CA, June 2003.

[29] C. Coarfa. *Portable High Performance and Scalability of Partitioned Global Address Space Languages*. PhD thesis, Rice University, Houston, TX, Jan. 2007.

[30] C. Coarfa, Y. Dotsenko, J. Eckhardt, and J. Mellor-Crummey. Co-Array Fortran performance and potential: An NPB experimental study. In *Proceedings of the 16th International Workshop on Languages and Compilers for Parallel Computing*, number 2958 in LNCS, College Station, TX, Oct. 2003. Springer-Verlag.

[31] C. Coarfa, Y. Dotsenko, and J. Mellor-Crummey. Experiences with Sweep3D implementations in Co-Array Fortran. In *Proceedings of the Los Alamos Computer Science Institute Fifth Annual Symposium*, Santa Fe, NM, Oct. 2004. Distributed on CD-ROM.

[32] C. Coarfa, Y. Dotsenko, and J. Mellor-Crummey. Experiences with Sweep3D implementations in Co-Array Fortran. *Journal of Supercomputing*, 36(2):101–121, May 2006.

[33] C. Coarfa, Y. Dotsenko, J. Mellor-Crummey, F. Cantonnet, T. El-Ghazawi, A. Mohanti, Y. Yao, and Chavarría-Miranda. An evaluation of Global Address Space Languages: Co-Array Fortran and Unified Parallel C. In *Proceedings of the 10th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, Chicago, IL, June 2005.

[34] C. Coarfa, Y. Dotsenko, L. Nakhleh, J. Mellor-Crummey, and U. Roshan. PRec-IDCM3: A parallel framework for fast and accurate large scale phylogeny reconstruc-

tion. In *Proceedings of the 2005 International Workshop on High Performance Computing in Medicine and Biology*, Fukuoka, Japan, July 2005. **Best Paper Award**.

[35] K. D. Cooper and L. Torczon. *Engineering a Compiler*. Morgan Kaufmann Publishers, San Fransisco, CA, 2004.

[36] Cray, Inc. Cray X1 server. `http://www.cray.com`.

[37] Cray, Inc. Cray X1E server. `http://www.cray.com`.

[38] Cray, Inc. Cray XT3. `http://www.cray.com/products/xt3`.

[39] Cray, Inc. Chapel specification v0.4. `http://chapel.cs.washington.edu/specification.pdf`, Feb. 2005.

[40] D. E. Culler, A. C. Arpaci-Dusseau, S. C. Goldstein, A. Krishnamurthy, S. Lumetta, T. von Eicken, and K. A. Yelick. Parallel programming in Split-C. In *Proceedings of Supercomputing*, pages 262–273, Nov. 1993.

[41] R. Cytron, J. Ferrante, B. Rosen, M. Wegman, and K. Zadeck. An efficient method of computing static single assignment form. In *Proceedings of the 16th Annual ACM Symposium on the Principles of Programming Languages*, Austin, TX, Jan. 1989.

[42] L. Dagum and R. Menon. OpenMP: An industry-standard API for shared-memory programming. *IEEE Computational Science and Engineering*, 5(1):46–55, 1998.

[43] A. Darte, D. Chavarría-Miranda, R. Fowler, and J. Mellor-Crummey. Generalized multipartitioning for multi-dimensional arrays. In *Proceedings of the International Parallel and Distributed Processing Symposium*, Fort Lauderdale, FL, Apr. 2002. Selected as **Best Paper**.

[44] A. Darte and R. Schreiber. A linear-time algorithm for optimal barrier placement. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, Chicago, IL, June 2005.

[45] K. Datta, D. Bonachea, and K. Yelick. Titanium performance and potential: an NPB experimental study. In *Proceedings of the 18th International Workshop on Languages and Compilers for Parallel Computing*, Hawthorne, NY, Oct. 2005.

[46] E. A. de Kock, W. J. M. Smits, P. van der Wolf, J.-Y. Brunel, W. M. Kruijtzer, P. Lieverse, K. A. Vissers, and G. Essink. YAPI: application modeling for signal processing systems. In *DAC '00: Proceedings of the 37th conference on Design Automation*, pages 402–405, Los Angeles, CA, 2000. ACM Press.

[47] Y. Dotsenko, C. Coarfa, and J. Mellor-Crummey. A multiplatform Co-Array Fortran compiler. In *Proceedings of the 13th International Conference of Parallel Architectures and Compilation Techniques*, Antibes Juan-les-Pins, France, Sept.–Oct. 2004.

[48] Y. Dotsenko, C. Coarfa, J. Mellor-Crummey, and D. Chavarría-Miranda. Experiences with Co-Array Fortran on hardware shared memory platforms. In *Proceedings of the 17th International Workshop on Languages and Compilers for Parallel Computing*, West Lafayette, IN, Sept. 2004.

[49] Y. Dotsenko, C. Coarfa, L. Nakhleh, J. Mellor-Crummey, and U. Roshan. PRec-IDCM3: A parallel framework for fast and accurate large scale phylogeny reconstruction. *International Journal of Bioinformatics Research and Applications*, 2(4):407–419, 2006.

[50] J. B. Drake, P. W. Jones, and G. R. Carr, Jr. Overview of the software design of the community climate system model. *International Journal of High Performance Computing Applications*, 19:177–186, 2005.

[51] J. A. for Marine-Earth Science and Technology. Earth Simulator. `http://www.es.jamstec.go.jp/esc/eng`.

[52] H. P. F. Forum. High Performance Fortran language specification, v2.0. `http://dacnet.rice.edu/Depts/CRPC/HPFF/versions/hpf2/hpf-v20`, Jan. 1997.

[53] I. Foster. Strand and PCN: Two generations of compositional programming languages. Technical Report MCS-P354-0293, Argonne National Laboratories, Argonne, IL, 1993.

[54] I. Foster and K. M. Chandy. Fortran M: A language for modular parallel programming. Technical Report MCS-P237-0992, Argonne National Laboratories, Argonne, IL, June 1992.

[55] V. Freeh and G. Andrews. `fsc`: A SISAL compiler for both distributed and shared-memory machines. Technical Report 95-01, University of Arizona, Tucson, AZ, Feb. 1995.

[56] A. Gara, M. A. Blumrich, D. Chen, G. L.-T. Chiu, P. Coteus, M. E. Giampapa, R. A. Haring, P. Heidelberger, D. Hoenicke, G. V. Kopcsay, T. A. Liebsch, M. Ohmacht, B. D. Steinmacher-Burow, T. Takken, and P. Vranas. Overview of the Blue Gene/L system architecture. *IBM Journal of Research and Development*, 49(2/3):195, 2005.

[57] D. A. Garza-Salazar and W. Bohm. D-OSC: a SISAL compiler for distributed-memory machines. In *Proceedings of the 2nd Parallel Computation and Scheduling Workshop*, Ensenada, Mexico, Aug. 1997.

[58] C. Grelck. Implementing the NAS benchmark MG in SAC. In *Proceedings of the 16th International Parallel and Distributed Processing Symposium*, Fort Lauderdale, FL, Apr. 2002.

[59] C. Grelck and S.-B. Scholz. SAC – from high-level programming with arrays to efficient parallel execution. In *Proceedings of the 2nd International Workshop on High Level Parallel Programming and Applications*, pages 113–125, Paris, France, June 2003.

[60] C. Grelck and S.-B. Scholz. Towards an efficient functional implementation of the NAS benchmark FT. In *Proceedings of the 7th International Conference on Parallel*

Computing Technologies, volume 2763 of *Lecture Notes in Computer Science*, pages 230–235, Nizhni Novgorod, Russia, Sept. 2003. Springer-Verlag.

[61] W. Gropp, E. Lusk, and A. Skjellum. *Using MPI: Portable Parallel Programming with Message-Passing Interface*. The MIT Press, Cambridge, MA, 1994.

[62] W. Gropp, M. Snir, B. Nitzberg, and E. Lusk. *MPI: The Complete Reference*. The MIT Press, Cambridge, MA, second edition, 1998.

[63] S. T. Group. Cilk 5.3.2 reference manual. `http://supertech.lcs.mit.edu/cilk/manual-5.3.2.pdf`, June 2000.

[64] M. Gupta, S. Midkiff, E. Schonberg, V. Seshadri, D. Shields, K. Wang, W. Ching, and T. Ngo. An HPF compiler for the IBM SP2. In *Proceedings of Supercomputing*, San Diego, CA, Dec. 1995.

[65] R. Gupta. The fuzzy barrier: a mechanism for high speed synchronization of processors. In *Proceedings of the 3rd International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 54–63, Boston, MA, 1989. ACM Press.

[66] 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. Technical Report UCB/EECS-2005-15, University of California at Berkeley, Berkeley, CA, Nov. 2005.

[67] J. P. Hoeflinger. Extending OpenMP to clusters. `http://cache-www.intel.com/cd/00/00/28/58/285865_285865.pdf`, 2005.

[68] Y. Hu, H. Lu, A. Cox, and W. Zwaenepoel. OpenMP for networks of SMPs. *Journal of Parallel and Distributed Computing*, 60(12):1512–1530, Dec. 2000.

[69] IBM Corporation. Report on the experimental language X10, draft v0.41. `http://domino.research.ibm.com/comm/research_projects.nsf/pages/x10.index.html/$FILE/ATTH4YZ5.pdf`, Feb. 2006.

[70] Intel Corporation. Cluster OpenMP user's guide v9.1 rev4.1. `http://cache-www.intel.com/cd/00/00/32/91/329148_329148.pdf`, May 2006.

[71] Intrepid Technology Inc. GCC Unified Parallel C. `http://www.intrepid.com/upc`.

[72] T. Jeremiassen and S. Eggers. Static analysis of barrier synchronization in explicitly parallel systems. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*, Montreal, Canada, Aug. 1994.

[73] H. Jin, M. Frumkin, and J. Yan. The OpenMP implementation of NAS parallel benchmarks and its performance. Technical Report NAS-99-011, NASA Ames Research Center, Moffett Field, CA, Oct. 1999.

[74] A. Kamil, J. Su, and K. Yelick. Making sequential consistency practical in Titanium. In *Proceedings of Supercomputing*, Seattle, Washington, Nov. 2005.

[75] A. Kamil and K. Yelick. Concurrency analysis for parallel programs with textually aligned barriers. In *Proceedings of the 18th International Workshop on Languages and Compilers for Parallel Computing*, Hawthorne, NY, Oct. 2005.

[76] P. Keleher, S. Dwarkadas, A. L. Cox, and W. Zwaenepoel. TreadMarks: Distributed shared memory on standard workstations and operating systems. In *Proceedings of the Winter 1994 USENIX Conference*, pages 115–131, San Francisco, CA, Jan. 1994.

[77] C. Koelbel, D. Loveman, R. Schreiber, G. Steele, Jr., and M. Zosel. *The High Performance Fortran Handbook*. The MIT Press, Cambridge, MA, 1994.

[78] T. Lengauer and R. E. Tarjan. A fast algorithm for finding dominators in a flowgraph. *ACM Transactions on Programming Languages and Systems*, 1(1):121–141, 1979.

[79] J. Mellor-Crummey, R. Fowler, G. Marin, and N. Tallent. HPCView: A tool for top-down analysis of node performance. *The Journal of Supercomputing*, 23:81–101, 2002. *Special Issue with selected papers from the Los Alamos Computer Science Institute Symposium.*

[80] J. M. Mellor-Crummey and M. L. Scott. Algorithms for scalable synchronization on shared-memory multiprocessors. *ACM Transactions on Computer Systems*, 9(1):21–65, 1991.

[81] V. Naik. A scalable implementation of the NAS parallel benchmark BT on distributed memory systems. *IBM Systems Journal*, 34(2):273–291, 1995.

[82] B. Nichols, D. Buttlar, and J. P. Farrell. *Pthreads Programming*. O'Reilly & Associates, Inc., Sebastopol, CA, 1996.

[83] J. Nieplocha and B. Carpenter. ARMCI: A portable remote memory copy library for distributed array libraries and compiler run-time systems. *Lecture Notes in Computer Science*, 1586:533–546, 1999.

[84] J. Nieplocha, V. Tipparaju, and D. Panda. Protocols and strategies for optimizing performance of remote memory operations on clusters. In *Proceedings of the Workshop on Communication Architecture for Clusters (CAC02) of IPDPS'02*, Fort Lauderdale, Florida, Apr. 2002.

[85] D. S. Nikolopoulos, T. S. Papatheodorou, C. D. Polychronopoulos, J. Labarta, and E. Ayguadé. Leveraging transparent data distribution in OpenMP via user-level dynamic page migration. *Lecture Notes in Computer Science*, 1940:415–427, 2000.

[86] R. W. Numrich and J. Reid. Co-arrays in the next Fortran standard. *SIGPLAN Fortran Forum*, 24(2):4–17, 2005.

[87] R. W. Numrich and J. K. Reid. Co-Array Fortran for parallel programming. Technical Report RAL-TR-1998-060, Rutherford Appleton Laboratory, Didcot, United Kingdom, Aug. 1998.

[88] R. W. Numrich and J. K. Reid. Co-Array Fortran for parallel programming. *ACM Fortran Forum*, 17(2):1–31, Aug. 1998.

[89] H. Ogawa and S. Matsuoka. OMPI: optimizing MPI programs using partial evaluation. In *Proceedings of the 1996 ACM/IEEE conference on Supercomputing (CDROM)*, page 37, Pittsburgh, PA, 1996. IEEE Computer Society.

[90] D. R. O'Hallaron. Spark98: Sparse matrix kernels for shared memory and message passing systems. Technical Report CMU-CS-97-178, Carnegie Mellon University, Pittsburgh, PA, Oct. 1997.

[91] L. Oliker, A. Canning, J. Carter, J. Shalf, and S. Ethier. Scientific computations on modern parallel vector systems. In *Proceedings of Supercomputing*, Pittsburgh, PA, Nov. 2004.

[92] Pete Beckman, *et al*. ZeptoOS: the small Linux for big computers. `http://www-unix.mcs.anl.gov/zeptoos`.

[93] A. Petitet, R. C. Whaley, J. Dongarra, and A. Cleary. HPL – a portable implementation of the high-performance linpack benchmark for distributed-memory computers. `http://www.netlib.org/benchmark/hpl`, 2004.

[94] F. Petrini, W. Feng, A. Hoisie, S. Coll, and E. Frachtenberg. The Quadrics network: High performance clustering technology. *IEEE Micro*, 22(1):46–57, Jan.–Feb. 2002.

[95] S. Prakash, M. Dhagat, and R. Bagrodia. Synchronization issues in data-parallel languages. In *Proceedings of the 6th Workshop on Languages and Compilers for Parallel Computing*, pages 76–95, Aug. 1993.

[96] U. Ramachandran, R. S. Nikhil, N. Harel, J. M. Rehg, and K. Knobe. Space-Time Memory: A parallel programming abstraction for interactive multimedia applications. In *Proceedings of the 7th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 183–192, Atlanta, GA, May 1999.

[97] K. H. Randall. *Cilk: Efficient Multithreaded Computing*. PhD thesis, MIT Department of Electrical Engineering and Computer Science, Cambridge, MA, June 1998.

[98] C. Rasmussen, M. Sottile, and T. Bulatewicz. CHASM language interoperability tools. `http://sourceforge.net/projects/chasm-interop`, July 2003.

[99] D. P. Reed and R. K. Kanodia. Synchronization with eventcounts and sequencers. *Communications of the ACM*, 22(2):115–123, 1979.

[100] Rice University. HPCToolkit performance analysis tools. `http://www.hipersoft.rice.edu/hpctoolkit`.

[101] Rice University. Open64/SL compiler and tools. `http://hipersoft.cs.rice.edu/open64`.

[102] Rice University. Rice Terascale Cluster. `http://rcsg.rice.edu/rtc`.

[103] J. Robert H. Halstead. MULTILISP: a language for concurrent symbolic computation. *ACM Transactions on Programming Languages and Systems*, 7(4):501–538, 1985.

[104] S. H. Roosta. *Parallel Processing and Parallel Algorithms*. Springer, New York, NY, 2000.

[105] H. Sakagami, H. Murai, Y. Seo, and M. Yokokawa. 14.9 TFLOPS three-dimensional fluid simulation for fusion science with HPF on the Earth Simulator. In *Proceedings of the 2002 ACM/IEEE conference on Supercomputing*, pages 1–14. IEEE Computer Society Press, Nov. 2002.

[106] V. Saraswat, R. Jagadeesan, A. Solar-lezama, and C. von Praun. Determinate imperative programming: A clocked interpretation of imperative syntax. `http://www.saraswat.org/cf.pdf`, July 2005. Submitted for publication.

[107] V. Sarkar and D. Cann. POSC – a partitioning and optimizing SISAL compiler. In *Proceedings of the 4th International Conference on Supercomputing*, pages 148–164, Amsterdam, The Netherlands, June 1990. ACM Press.

[108] M. L. Scott. The Lynx distributed programming language: motivation, design and experience. *Computer Languages*, 16(3-4):209–233, Sept. 1991.

[109] S. L. Scott. Synchronization and communication in the T3E multiprocessor. In *Architectural Support for Programming Languages and Operating Systems*, pages 26–36, Cambridge, MA, Oct. 1996.

[110] N. Shavit and A. Zemach. Diffracting trees. *ACM Transactions on Computer Systems*, 14(4):385–428, 1996.

[111] Silicon Graphics, Inc. The SGI Altix 3000 global shared-memory architecture. `http://www.sgi.com/servers/altix/whitepapers/tech_papers.html`, 2004.

[112] M. Snir, S. W. Otto, S. Huss-Lederman, D. W. Walker, and J. Dongarra. *MPI: The Complete Reference*. The MIT Press, Cambridge, MA, 1995.

[113] Sun Microsystems, Inc. Java 2 platform standard edition 5.0 API specification. `http://java.sun.com/j2se/1.5.0/docs/api`.

[114] The Climate, Ocean and Sea Ice Modeling (COSIM) Team. The Parallel Ocean Program (POP). `http://climate.lanl.gov/Models/POP`.

[115] The Portland Group Compiler Technology, STMicroelectronics, Inc. PGHPF compiler user's guide. `http://www.pgroup.com/doc/pghpf_ug/hpfug.htm`.

[116] W. Thies, M. Karczmarek, and S. P. Amarasinghe. StreamIt: A language for streaming applications. In *Proceedings of the International Conference on Compiler Construction*, pages 179–196, Grenoble, France, Apr. 2002.

[117] W. Thies, M. Karczmarek, J. Sermulins, R. Rabbah, and S. Amarasinghe. Teleport Messaging for distributed stream programs. In *Proceedings of the 10th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, Chicago, IL, June 2005.

[118] C.-W. Tseng. Communication analysis for shared and distributed memory machines. In *Proceedings of the Workshop on Compiler Optimizations on Distributed Memory Systems*, San Antonio, TX, Oct. 1995.

[119] C.-W. Tseng. Compiler optimizations for eliminating barrier synchronization. In *Proceedings of the 5th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 144–155, Santa Barbara, CA, 1995.

[120] A. Uno. Software of the Earth Simulator. *Journal of the Earth Simulator*, 3:52–59, Sept. 2005.

[121] UPC Consortium. UPC language specifications, v1.2. Technical Report LBNL-59208, Lawrence Berkeley National Laboratory, Berkeley, CA, 2005.

[122] T. von Eicken, D. E. Culler, S. C. Goldstein, and K. E. Schauser. Active Messages: A mechanism for integrated communication and computation. In *Proceedings of the 19th International Annual Symposium on Computer Architecture*, pages 256–266, Gold Coast, Australia, 1992.

[123] Wikipedia. Kahn process networks. `http://en.wikipedia.org/wiki/Kahn_process_networks`.

[124] Wikipedia. Run-length encoding. `http://en.wikipedia.org/wiki/Run-length_encoding`.

[125] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. The SPLASH-2 programs: Characterization and methodological considerations. In *Proceedings of the 22th International Symposium on Computer Architecture*, pages 24–36, Santa Margherita Ligure, Italy, June 1995.

[126] K. Zhang, J. Mellor-Crummey, and R. Fowler. Compilation and runtime-optimizations for software distributed shared memory. In *LCR'00: Selected Papers from the 5th International Workshop on Languages, Compilers, and Run-Time Systems for Scalable Computers*, pages 182–191, London, United Kingdom, 2000. Springer-Verlag.