RICE UNIVERSITY

Function Shipping in a Scalable Parallel Programming Model

by

Chaoran Yang

A Thesis Submitted
in Partial Fulfillment of the
Requirements for the Degree

Master of Science

Houston, Texas

April, 2012

ABSTRACT

Function Shipping in a Scalable Parallel Programming Model

by

Chaoran Yang

Increasingly, a large number of scientific and technical applications exhibit dynamically generated parallelism or irregular data access patterns. These applications pose significant challenges to achieving scalable performance on large scale parallel systems. This thesis explores the advantages of using *function shipping* as a language level primitive to help simplify writing scalable irregular and dynamic parallel applications. Function shipping provides a mechanism to avoid exposing latency, by enabling users ship data and computation together to a remote worker for execution. In the context of the Coarray Fortran 2.0 Partitioned Global Address Space language, we implement function shipping and the *finish* synchronization construct, which ensures global completion of a set of shipped function instances. We demonstrate the usability and performance benefits of using function shipping with several benchmarks. Experiments on emerging supercomputers show that function shipping is useful and effective in achieving scalable performance with dynamic and irregular algorithms.

Acknowledgments

I want to express my deepest gratitude to my adviser, Dr. John Mellor-Crummey, for his invaluable guidance and assistance during my study at Rice. This thesis would not have been possible without him.

I want to thank my other committee members, Keith Cooper and Vivek Sarkar, for their insightful comments and discussions. I want to thank my colleague Karthik Murthy, the principle author of UTS implementation in CAF 2.0, for a fruitful and rewarding collaboration. I would like to thank Bill Scherer, Laksono Adhianto, Guohua Jin, for their work on the runtime system of CAF 2.0, and Scott Warren, Fengmei Zhao, Dung Nguyen, and Mark Krentel for their work on the CAF 2.0 compiler.

I would like to thank Brian and Lily Lam, for helping me walk through my bad days, and Powei Feng, for being a faithful and truthful brother to me all the time. I would like to thank members of my spiritual family—Rice Chinese Christian Fellowship, for their encouragement and support through prayers.

Last but not the least, I am grateful to God for his everlasting love and abundant grace, for as it is said in 1 Samuel 7:12, "*Thus far the LORD has helped us*".

This research was supported by the Department of Energy's Office of Science under cooperative agreements DE-FC02-07ER25800 and DE-FC02-06ER25754. This research used resources of the National Energy Research Scientific Computing Center (NERSC) at Lawrence Berkeley National Laboratory and the National Center for Computational Sciences (NCCS) at Oak Ridge National Laboratory. Both facilities are supported by the Office of Science of the U.S. Department of Energy. NCCS is supported under Contract No. DE-AC05-00OR22725 and NERSC is supported under Contract No. DE-AC02-05CH11231.

Contents

Illustrations

Tables

Chapter 1

Introduction

Writing shared-memory parallel programs is generally considered easier than writing parallel programs using a distributed memory model, in which accessing remote data is achieved through explicit communication. However, the inability to specify, manage, and exploit locality when using a shared-memory programming models hinders their application in scientific and technical computing, where high performance and scalability is required. For parallel scientific and technical applications, the message passing model is still the pervasive parallel programming model being used on supercomputers and clusters today. The *Message Passing Interface* (MPI) [1] is a standard API that supports programming in the message passing model.

The Partitioned Global Address Space (PGAS) model has been developed to bridge the gap between the ease of programming of shared-memory models and the performance and scalability of message passing models. In the PGAS model, threads or processes share a global address space. For convenience, we refer to all active entities, i.e., threads and processes, as simply threads. The shared address space is *partitioned* into local and remote portions. Figure 1.1 depicts an example of a partitioned global address space with four threads. Within it, each thread's local address space is partitioned into private and shared portions. Only a local thread may reference its private area, whereas the shared area may be accessed by any thread. A thread accessing data located in other threads' shared space pays a much higher cost than accessing local data. The list of PGAS languages includes but is not lim-

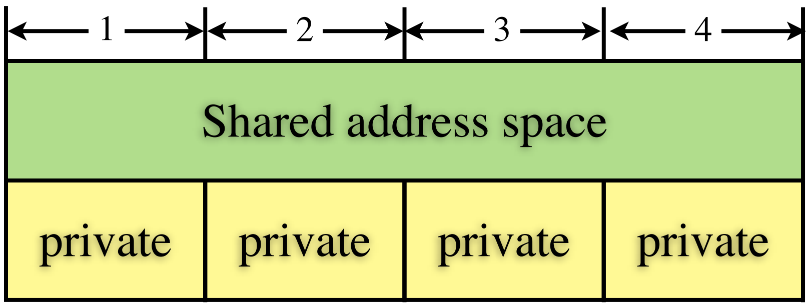

Figure 1.1 : An example of an address space with 4 threads in PGAS model

ited to: Unified Parallel C (UPC) [2], Titanium [3], Coarray Fortran [4], X10 [5], and Chapel [6]. The PGAS model provides both performance and usability benefits for implementing parallel applications on large distributed systems. By having each thread primarily compute on data in its local portion of the address space, programs written in PGAS model are able to achieve good performance. The PGAS model also simplifies programming by adopting a one-sided communication model, where a thread can read or modify remote data without any explicit involvement of a thread on the remote node.

However, many algorithms still require great programming effort to achieve reasonable performance on large-scale parallel systems. Algorithms that feature dynamically generated computation or data access patterns where it is difficult to exploit locality are particularly difficult to map onto scalable parallel systems. Dynamically generated computation prevents work from being pre-partitioned equally among threads. Algorithms with dynamic generated parallelism require constant, dynamic load balancing to maintain high efficiency. Algorithms that have little data locality suffer from the fact that the cost of accessing remote data on distributed memory systems is often several magnitudes higher than that of local computation. Programs with low locality often spend more time waiting for data than actually performing computation on it. Examples of dynamic and irregular algorithms include the Self-Consistent Field method [7]— a technique commonly used in *ab initio* computational chemistry that involves irregular data access and requires dynamic load balancing while results are accumulated, and MADNESS (Multiresolution ADaptive NumErical Scientific Simulation) [8]— a framework that uses adaptive multiresolution analysis methods in multiwavelet bases for scientific simulation, where the size and shape of a resulting task tree depend on a user-specified analytic function.

To help achieve scalable performance for irregular and dynamic parallel applications, this thesis explores the effectiveness of using *function shipping*, a mechanism that allows moving data and computation together between threads, in the PGAS model. Function shipping provides new opportunity for users to better manage dynamic generated computation and avoid exposing communication latency. Taking a simple load balancing problem depicted in Figure 1.2 as an example, when thread p runs out of work locally and tries to steal work from a randomly-picked thread q, it sends out *request* to q to check whether q has work left ($req\ A$). If q has no work, q must *reply* back to p ($rep\ A$) that it has no work, so that p can try to steal from another thread r ($req\ B$). p incurs the cost of a round trip across the network ($req\ A$ & $rep\ A$) for each steal attempt it makes until work is found on a thread; another round trip ($req\ B$ & $rep\ B$) is needed to obtain the work. With function shipping, however, thread p may ship the action needed after detecting q's work pool status along with its initial request ($reqf\ C$).* Thus, the attempt of stealing from thread r may be made by q directly ($reqf\ D$), thereby avoiding the intervening reply

*$reqf$ denotes shipping a function as a request

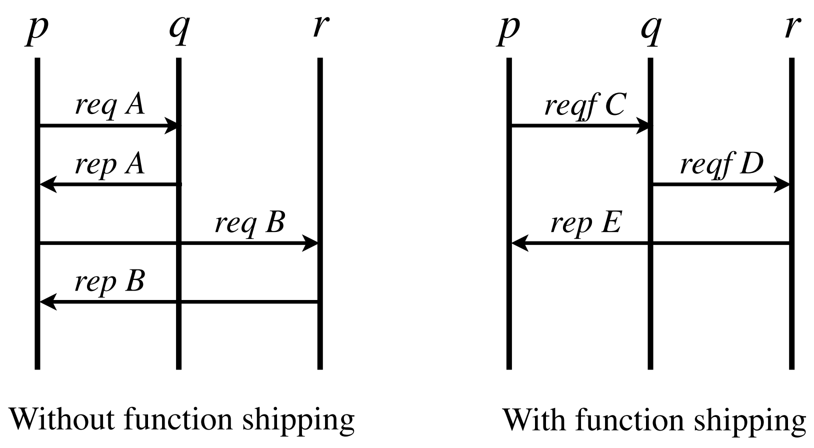

Figure 1.2 : An example of function shipping avoiding exposing latency in a simple load balancing problem

back to p. If a thread attempts to steal w times before it finds work, the approach without using function shipping costs $2w$ messages in the worst case, whereas the function shipping approach uses only $w + 1$ messages, which avoids exposing the latency of $w - 1$ messages.

This thesis explores the integration of function shipping into the Coarray Fortran 2.0 (CAF 2.0) [9] language under development at Rice University. CAF 2.0 is a rich set of PGAS extensions to Fortran 2003. A detailed description of the CAF 2.0 language is presented in Section 3.2. Our implementation of function shipping is build upon Berkeley's GASNet communication system [10]. We show that the performance of our implementation of function shipping in CAF 2.0 is comparable to MPI's send & receive routines and Active Messages [11] in GASNet. We demonstrate its utility by using it for load balancing in the implementation of an Unbalanced Tree Search (UTS) benchmark [12].

Our design of function shipping enables a shipped function to perform the full range of operations as a normal function in CAF 2.0. In particular, a shipped function may spawn more functions and ship them to others in CAF 2.0. Given the SPMD nature of CAF 2.0 programs, where computation originates from multiple threads causing no single thread knowing the global status of the system, the flexibility of spawning functions from a shipped function complicates the problem of detecting completion of these dynamically spawned functions. Solutions to detecting termination of dynamic computation fall in a class of algorithms known as *termination detection* [13]. To address this, we developed a `finish` construct, a block construct that guarantees completion of asynchronous operations, i.e., shipped functions within it. Our finish construct is inspired by X10 programming language, but differs from X10's construct with the same name because we adapted it to fit into CAF 2.0's SPMD programming model. We also introduce a scalable distributed global termination algorithm that is used by our `finish` construct. Our termination detection problem exhibits two desirable properties: 1) it requires only a constant amount of space per thread; 2) it detects global termination using only a bounded number of rounds of speculative waves.

1.1 Statement of thesis

Adding function shipping and the `finish` synchronization construct, which ensures global completion of a set of shipped function instances, to a PGAS programming language improves the expressiveness of the language and helps the language deliver high performance for dynamic and irregular algorithms on large-scale parallel systems. We support this thesis by integrating function shipping and the `finish` construct into the Coarray Fortran 2.0 language. We demonstrate the utility of function shipping and the `finish` construct by using them to implement several benchmarks and algorithms. Evaluating these benchmarks and algorithms on different supercomputers and clusters shows that function shipping and the `finish` construct simplifies programming and increases performance of applications with dynamically generated parallelism and irregular data access patterns.

1.2 Thesis outline

The rest of the thesis is organized as follows. Chapter 2 discusses previous work related to function shipping and global termination detection. Chapter 3 introduces CAF 2.0 and its features; this background provides context for our extensions to CAF 2.0. Chapter 4 describes in detail our design and implementation of function shipping and the `finish` construct in CAF 2.0. Chapter 5 evaluates their performance and usability with benchmarks and algorithms. Chapter 6 concludes and discusses topics for future work.

Chapter 2

Related Work

This chapter summarizes previous work related to the idea of transfer computation and data to a different active entity, i.e., thread or process, for execution. Then, it discusses several termination detection algorithms used by different parallel programming systems and languages. Where appropriate, our approach is compared and contrasted with existing approaches.

2.1 Function shipping

The idea of transferring computation and data to a different active entity to simplify programming of dynamic irregular parallel applications has been explored by many programming languages and systems. We group them into three categorizes simply for the convenience of discussion.

First, in the realm of distributed computing, Remote Procedure Call (RPC) [14] is used for invoking a procedure in a different address space. Examples of popular RPC systems include Java Remote Method Invocation [15] and Open Network Computing RPC [16]. RPC systems are designed for usability, and more importantly to support dynamic dispatching, interface discovery, and security features, often at the expense of performance. They are not suitable to be applied to scientific computing where high performance is required. ARMI [17] and Charm++ [18] are two programming systems that introduced the RPC-style communication into high performance computing. These RPC and RPC-related systems expose either functions or objects and their methods for remote access. Compared with these systems, our design of function shipping exposes all functions and subroutines in a program as candidates for remote invocation.

In the realm of high performance computing, Active Messages (AM), originally developed by von Eicken et al. in 1992 [11] is the widely adopted communication pattern in the design of several runtime systems and libraries, including GASNet [10], IBM's Deep Computing Messaging Framework (DCMF) [19], IBM's Low-level Application Programming Interface (LAPI) [20], and the Asynchronous PGAS runtime system [21]. These active message systems have a low-level interface and have restrictions on their message handlers's capability to perform long-lived computations or communicate. In particular, they forbid an active message handler to communicate with other processes except to send a reply message to its source process. Also, they forbid an active message handler to make blocking calls. These restrictions are used to avoid deadlocks [11]. Because of these restrictions and the fact that they are designed as low-level transportation layers for higher level systems or languages to build upon, they are unsuitable for use as a user-level communication layer. AM++ [22], developed by Willcock and Hoefler et al., relaxed the restriction to send only a reply message for more flexibility. It supports runtime optimizations such as message combining and filtering for better performance. Optimistic Active Messages [23] introduced a technique that allows an active message to run inside an interrupt handler but revert to creating a separate thread to handle the active message when the active message makes a forbidden operation such as a blocking call. Our design of function shipping adopts the method of Optimistic Active Message to avoid the overhead of switching threads whenever possible. We also provides the flexibility of arbitrary actions within a shipped function. Unlike AM++ and Optimistic Active Messages, which provide library-based support for function shipping, we explore the integration of function shipping into a PGAS programming language.

Several parallel programming languages have adopted concepts related to function shipping to help better manage dynamic task parallelism by supporting language primitives to create asynchronous tasks. The `future` construct originally introduced by Multilisp [24] serves the purpose of both creating new tasks and synchronizing among them. A statement $future\ X$ immediately returns a future object and may evaluate the expression X later. A read operation to the future object will be suspended until the evaluation of X is completed. Java and X10 also include `future` as a construct to support task parallelism. In Cilk [25], a `spawn` statement *spawns* a function and creates a continuation* of the program that it provides to the runtime scheduler. Another thread may then *steal* the continuation frame and execute concurrently with the spawned function. Although an implementation of Cilk for distributed memory systems was attempted [26], it is best suited for systems with hardware support for shared memory. X10 and Habanero Java [27] extend Cilk's `spawn` statement by enables spawning of arbitrary statements or blocks of statements as a new parallel task. They also enable spawning new tasks across nodes by using a *place* as the optional `at` argument to an `async`; the *place* specifies where a computation should execute. Chapel provides a `begin` primitive similar to the `async` in X10 and Habanero Java. The functionality of creating new tasks across *locales*, however, is not implemented as of January 2012. Our work on function shipping is different from these language primitives mentioned above in that it is tightly integrated with

*A continuation is the calling stack and program counter of a program—the information needed for resuming execution from the program's current state.

the other features of the CAF 2.0 language such as asynchronous operations and synchronization constructs. This thesis discusses subtle interactions between function shipping and point-to-point synchronization and `finish` constructs in Section 4.2.1.

2.2 Global termination detection

Shipping a function for asynchronous execution without knowing when it is completed is useless. Synchronization constructs that ensure completion of these asynchronous tasks are needed. Cilk uses a `sync` statement to ensure completion of all previously spawned tasks. Each function that contains *spawn* calls in Cilk also has an implicit `sync` statement at the end. The implicit `sync` restricts Cilk's computation model to be a *fully-strict* model, where a parent task must wait for its children to complete before it ends. Our `finish` construct is inspired by X10's synchronization construct with the same name. The `finish` construct in X10 relaxes this model by allowing parent tasks to exit without waiting for the completion of their children, which defines the *terminally-strict* computation model [28]. Our work on `finish` in CAF 2.0 follows the terminally-strict computation model since it can express a broader range of algorithms.

Detecting termination of nested asynchronous tasks on a shared memory system is a trivial problem. Cilk uses a simple algorithm in which each parent task records the number of active children tasks and exits when the counter is decremented to zero. This suffices to ensure completion of spawned functions in its fully-strict model.

The fact that a shipped function in CAF 2.0 can ship more functions complicates the problem of detecting completion of all these *nested* shipped functions. *Distributed termination detection* algorithms are required to determine termination reliably and efficiently in a distributed system. The distributed termination problem has been extensively studied [29, 30, 31, 32]. The algorithms in this area can be broadly classified as *symmetric* and *asymmetric* algorithms. In the symmetric algorithms all processes execute identical code and detect termination together. The asymmetric algorithms rely on a pre-designated process for termination detection.

The implementation of X10 for scalable parallel systems uses a *vector counting* algorithm to detect global termination [33]. Each worker maintains a vector that contains a count per place, tracking the number of activities it spawned remotely and completed locally. Once a worker has quiesced (no active tasks in this place), it sends its vector to the place that owns the `finish`. Global termination is detected once the place that owns the `finish` receives vectors from everyone and the sum-reduced vector is zero. This algorithm suffers from the fact that a single place is responsible to receive p vectors of size p, where p denotes the number of places. This will become a bottleneck when scaling to a large number of places.

Scioto [34], a framework that provides dynamic load balancing on distributed memory machines, implemented a wave-based algorithm similar to that proposed by Francez and Rodeh [35] to detect global termination. Their termination detection algorithm, a token wave broadcasts up and down a binary spanning tree mapped onto the process space. Each process owns a token, which is initially *white*. In the up-wave of termination detection, a *black* token is generated when one has performed a load balancing operation since the last down-wave. A black token is passed to a node's parent in the tree to signal a new round of down-and-up waves.

Because both X10 and Scioto are global-view programming models, the algorithms they used for termination detection are asymmetric. All dynamic tasks in X10 and Scioto share the same ancestor, which is a natural place to oversee termination detection. Since CAF 2.0 follows the SPMD model, where computation starts simultaneously from multiple places, these algorithms are not suitable for use by the `finish` construct in CAF 2.0.

AM++ employs an algorithm that uses four counters [36, §4] and non-blocking global sum reductions to accumulate the counts to determine global termination [22]. The four-counter algorithm *twice* counts the messages sent and received by each process. Equality of these four counters guarantees correct detection of termination by the system. Because this algorithm counts twice, it always incurs an extra global reduction to detect termination; our algorithm does not pay this extra cost. Moreover, in the implementation of termination detection in AM++, if a user specifies the longest length of the chain of messages will be used, they report that a known chain length allows a simpler algorithm with lower message complexity to be used for termination detection. Our algorithm does not require the knowledge of the length of the chain of shipped functions, but still keeps a tight bound on message complexity. Details of our algorithm are described in Section 4.2.3.

Chapter 3

Background

3.1 Coarray Fortran

In 1998, Numrich and Reid proposed a small set of extensions to Fortran 95 to support parallel programming that they dubbed Coarray Fortran (CAF) [4]. Their major extension to Fortran was *coarrays* which users use to declare and access shared data. For example, the declaration

```
integer :: A(N,M)[*]
```

declares a shared coarray A with $N \times M$ integers local to each image. Dimensions in the bracketed tuple are called *codimensions*. Using coarrays, one can directly access data associated with another image by adding a bracketed tuple to a coarray variable reference. For example, the statement

```
A(:,N)[q] = A(:,1)[p]
```

reads the first column of data in coarray A on image p then uses it to update the last column of A on image q.

3.2 Coarray Fortran 2.0

In 2005, the Fortran Standards committee began exploring the addition of coarray constructs to a new version of Fortran that would later become Fortran 2008. The design for coarrays in Fortran 2008 closely follows Numrich and Reid's original vision of

Coarray Fortran, however, is flawed in several respects. Among its flaws, it lacks support for image subsets and is unable to express that the latency for accessing remote data should be overlapped with computation. A detailed critique of the proposed coarray extensions for Fortran 2008 can be found in a paper by Mellor-Crummey et al. [37].

Our group has been working on a set of extensions to Fortran that we call Coarray Fortran 2.0 (CAF 2.0). CAF 2.0 adds a richer set of extensions to Fortran to enable users to express a wider spectrum of parallel algorithms in a more efficient and scalable way. CAF 2.0 features include but are not limited to *teams, events, asynchronous copy, asynchronous collectives, copointers, and topologies.* In the following subsections, we briefly summarize some key features of Coarray Fortran 2.0, for their knowledge is essential in discussing the new constructs studied in this thesis. A more detailed description of Coarray Fortran 2.0 language is presented in earlier work [9, 38, 39].

3.2.1 Teams

A *team* is a first-class entity that represents a process subset in CAF 2.0. The existence of teams in CAF 2.0 has three purposes. First, this set of images serves as a domain onto which coarrays may be allocated. Second, it provides a namespace within which process images can be indexed by their relative rank within that team, instead of an absolute image ID. Third, a team provides an isolated domain on which a subset of process images to communicate and synchronize collectively.

Initially, CAF 2.0 programs begin with a global team named `team_world` to which every image belongs. New teams are created by calling `team_split` on an existing team. All images that supply the same color to a `team_split` become members of

```
1  team :: rowteam, colteam
2  integer :: myrow, mycol, me, nprow
3  double precision, allocatable :: A(:)[*]
4
5  mycol = team_rank() / nprow
6  myrow = team_rank() - mycol * nprow
7
8  call team_split(team_world, mycol, myrow, colteam, mycol)
9
10 with team colteam
11     allocate(A(10)[])
12 end with team
```

Figure 3.1 : `team_split` and coarray allocation within a team

the same new team after the operation completes. Figure 3.1 shows an example of creating a new team with `team_split` and using the team to control allocation of coarray `A`. CAF 2.0 also introduces the concept of a default team, which is specified using a `with team` block (line 10–12). The default team is implicitly used any time a team is required but not specified. Note that `with team` blocks are dynamically scoped and may be nested. For example, the `team_size` function (line 5–6) inspects the number of process images in the default team, which is `team_world` since no `with team` block is created yet; however, the allocation of coarray `A` (line 11) is performed by images of the column sub-team (`colteam`).

3.2.2 Events

Events in CAF 2.0 serve as a mechanism to support point-to-point synchronization. Figure 3.2 lists the APIs for manipulating events in CAF 2.0. Event objects can be viewed as *counting semaphores*. An event must be initialized before use by invoking `event_init`. A `event_notify` operation increases an event's count by one, or if the

```
event_init(event e)
event_notify(event e[,integer n])
event_wait(event e[,integer n])
event_trywait(event e,[integer n,]logical success)
```

Figure 3.2 : Examples of event operations in CAF 2.0

optional argument `n` is specified, by `n`. An `event_wait` operation blocks the execution of the current thread until the event has been notified once or the specified number of times. CAF 2.0 also includes a non-blocking `event_trywait` operation that attempts to consume an event's counter by one or `n` if specified, and report whether it succeeded or not with the logical variable `success`. The optional argument `n` enhances the usability of events in cases such as stencil calculation, where a process might use the same event to wait on all of its north, south, east and west neighbors to update their values before local computation can proceed. In this case, specifying `4` for `n` simplifies programming.

In CAF 2.0, there are two ways that an event can be posted: 1) it can be notified explicitly through `event_notify`; 2) also, it can be attached to an asynchronous operation so that it is notified when the operation completes. The following sections describe the use of events with asynchronous operations.

3.2.3 Predicated asynchronous copy

There are three categories of asynchronous operations in CAF 2.0: asynchronous copy, asynchronous collectives, and shipped functions [38]. A *predicated asynchronous copy* provides a flexible mechanism to copy data from one process image to another asynchronously. The `copy_async`, as shown below,

```
copy_async(destA[p1], srcA[p2], pre_event, src_event, dest_event)
```

broadcast*	gather	scatter	reduce
scan	shift	sort	permute
alltoall	allreduce	allgather	barrier*

* Asynchronous versions of these collectives are implemented at the time of this writing

Table 3.1 : Collective operations in CAF 2.0

copies `srcA` on process image `p2` to `destA` on image `p1`. The copy will be initiated after `pre_event` is posted. It could be used by the sender to indicate the source data is ready to be copy, or used by the receiver to trigger the copy operation when the receiver is ready to receive the data. Notification of `src_event` indicates that the `srcA` is free to be modified, and notification of `dest_event` indicates that the copy operation is complete on the destination process `p1`. Note that CAF 2.0 uses two separate events to distinguish two stages of completion of an asynchronous copy operation. This yields opportunity for user to overwrite the source data at the possible earliest time so that a subsequent `copy_async` operation can be initiated without waiting for destination completion.

3.2.4 Asynchronous collectives

The fact that most high performance computing applications are written using MPI for collective operation proves that collectives are effective for expressing scalable algorithms. As does MPI, CAF 2.0 includes many collective operations; they are shown in Figure 3.1. A process image that participates a synchronous collective operation waits for two reasons: communication latency and asynchrony among images. CAF 2.0 provides asynchronous versions of these collectives, which enable users to overlap waiting that occurs in collective operations with local computation.

We describe the semantics of an *asynchronous barrier* operation, also known as a split-phase barrier, as an example to show the usefulness of asynchronous collectives. Upon reaching a synchronization point, each image initiates an asynchronous barrier and proceeds with local computation that can be done before everyone else arrives. Later, it can require the completion of the barrier by using either an `event_wait` or a `finish` block. Asynchronous barriers enables the cost of synchronization to be overlapped with local computation.

Chapter 4

Approach

On today's large-scale supercomputers, the latency of accessing data on a remote node is often thousands times more than that from local memory. To achieve high performance on these systems, it is critical to hide communication latency by overlapping it with local computation. One obvious way of hiding latency is to continue local computation while data is transferring and synchronization is in flight. Under many circumstances, determining whether a `PUT` or `GET` operation can be transformed into its non-blocking form is hard for a compiler. In particular, this optimization opportunity is difficult to exploit when code is compiled separately. Since a programmer knows what computation can be safely overlapped with computation, CAF 2.0 provides asynchronous copy and asynchronous collective constructs for programmers to express such asynchronous communication.

Function shipping is another primitive that we integrate into CAF 2.0 to deal with communication latency on large-scale machines. It provides users the ability to co-locate computation with data. This ability is essential in avoiding unnecessary communication in certain circumstances. As we discussed in Section 5.2.2, in studies with the HPC Challenge RandomAccess benchmark, shipping element updates of a remote location saves one round trip compared with the method of reading an element remotely, updating it, and writing it back. Moreover, function shipping enables users to reduce synchronization points in a program. In the RandomAccess benchmark example, without function shipping the thread updating a table entry on a remote

```
1 event :: ev
2 ...
3 spawn(ev) foo(table(i,j)[p], n)[p]
4 call event_wait(ev)
5
6 finish (a_team)
7         spawn foo(table(i,j)[p], n)[p]
8         ...
9 end finish
```

Figure 4.1 : Explicit and implicit model examples of CAF 2.0 function shipping

thread needs to wait until the entry arrives. But using a shipped function to update a remote table entry does not require synchronization on both remote and local threads. We present our design and implementation of function shipping in CAF 2.0 in detail in Section 4.1.

To support the integration of function shipping into CAF 2.0, we also designed a synchronization construct `finish` that efficiently detects global completion of a set of shipped function instances spawned within the `finish` block. Section 4.2 discusses our algorithm implementing the `finish` construct, and proves that its communication complexity is bounded by the depth of the longest chain of shipped functions.

4.1 Function shipping

4.1.1 Syntax

As shown in Figure 4.1, replacing the Fortran keyword `call` with `spawn` executes a procedure call (line 3) in CAF 2.0 on a remote image *asynchronously.* The destination image is specified within the ending square bracket pair. A spawn immediately returns after the shipped function leaves the source buffer and the source data can be modified.

Dummy argument declaration for an coarray argument	Meaning of different references of the coarray in a shipped function	
`integer :: A`	`A`:	a copy of A on sender
`integer :: A[*]`	`A`:	coarray A on receiver
	`A[x]`:	coarray A on image x

Table 4.1 : Meaning of coarray references within a shipped function.

Like other asynchronous operations in CAF 2.0, a programmer can synchronize a shipped function by binding the shipped function with an event object. The event appearing between a pair of parentheses after the keyword `spawn` (line 2) will be notified when the shipped function completes. Instead of tracking shipped functions one by one with events, users can also use the `finish` construct to manage completion of spawned functions efficiently when they do not need to know when each individual function completes (line 6). A detailed description of using the *explicit* and *implicit* models of completion in CAF 2.0 is in Section 4.2.1.

4.1.2 Semantics

Arguments to a shipped function

Arguments passed into a shipped function are treated differently from arguments to normal functions. Coarray arguments are handled according to the dummy argument declaration within that function. Within a shipped function, the meaning of a reference depends upon the combination of the actual argument passed to the shipped function and its dummy declaration with the function. Table 4.1 shows that coarray arguments declared as non-coarray variables in dummy arguments are dereferenced at the call site. Non-coarray dummy arguments are copied to the remote image along with the shipped function; they act as if they were implicitly marked with `VALUE` attributes: modifications on them will not be reflected back to the calling image. Support for intent attributes such as `IN`, `OUT`, `INOUT` is not implemented at the time this thesis is written. However, support for intent attributes in CAF 2.0 are desirable to make the semantic of shipped functions compliant with Fortran calling conventions; it will be included as part of our future work. Table 4.1 also shows that a coarray dummy argument gives the function the ability to access portions of the coarray on any image. The meaning of a reference to a coarray dummy argument within a shipped function is the same as that of a coarray reference outside a shipped function in CAF 2.0.

Execution context

Shipped functions are usually executed on a different image from the one that spawned them. Thus, the context in which shipped functions execute is the one on the image that is the target of the spawn. Therefore, they can access global data local to the target image, even if the global data is not shared across images. We believe the change of execution context is necessary in making function shipping fully expressive. Manipulating many distributed data structures such as distributed hash tables, lists and graphs requires shipped functions to have the ability to operate on global data on the target image.

4.1.3 Implementation

This section presents the implementation details of function shipping in CAF 2.0. Since the runtime system of CAF 2.0 uses GASNet [10] as a communication layer, function shipping in CAF 2.0 is implemented on top of Active Messages of GASNet. We start this section with a brief introduction of the asynchronous engine we build into

```
typedef struct async_record_s {
        async_status_t status;
        caf_team_t team;
        size_t async_id;
        async_progress_fxn *progress_fxn;
        size_t finish_handle;
        struct async_record_s *next;
        long async_state_data[0];
} async_record_t;
```

Figure 4.2 : `async_record_t` structure used in the asynchronous engine of CAF 2.0 runtime system.

CAF 2.0's runtime system, which is used for execution of asynchronous operations.

Asynchronous engine

The asynchronous progress engine is a key piece of machinery in CAF 2.0. It implements cooperative multithreading and message-based parallelism in support of asynchrony. The implementation of the progress engine is fairly straightforward. We maintain a linked list of the asynchronous operations that are currently pending. Associated with each operation is a `async_record_t` structure which has three major fields: `status`, `progress_fxn`, `finish_handle` and `async_state_data`, as shown in Figure 4.2. In it, `status` is an indicator of the current state of the operation, and may be any of the values of `ASYNC_UNINITIALIZED`, `ASYNC_INPROGRESS`, or `ASYNC_COMPLETE`. The `progress_fxn` field is a progress function, invoked on behalf of the operation whenever the progress engine is active. The `async_state_data` is a placeholder for operation-specific data. Finally, `finish_handle` records the finish block that spawned the operation so that upon completion of an asynchronous operation, we are able to report it back to its corresponding finish scope. For asynchronous

`atomic_add`	`atomic_sub`	`atomic_or`	`atomic_and`	`atomic_xor`
`atomic_fadd`*	`atomic_fsub`*	`atomic_for`*	`atomic_fand`*	`atomic_fxor`*

* These atomic operations will first perform the update then *fetch* the old value back.

Table 4.2 : Remote atomic operations in CAF 2.0.

operations that carry explicit event objects, the `finish_handle` field is assigned an invalid value.

Function shipping in CAF 2.0 makes use of the asynchronous progress engine to execute and save a shipped function on remote image. When a shipped function cannot be executed inside an active message handler, it will be converted into an `async_record_t` struct, then be added to the asynchronous operations list on the target image for later execution when the asynchronous engine is invoked.

Argument marshaling

The compiler generates a structure for each `spawn` call to hold the arguments passed to that call. The compiler also generates a pair of functions to marshal and de-marshal arguments to a shipped function. For scalar variables or normal Fortran arrays, data is serialized and packed into a buffer which sent along with an active message. For coarray arguments, we create a reference type for it which encapsulates coarray's handle, upper bound, lower bound and stride of each dimension together to support reconstruction of the reference on the remote process. To improve performance of function shipping, where possible, we have integrated into CAF 2.0 support for several remote atomic operations. Table 4.1.3 lists remote atomic operations currently supported in CAF 2.0. Atomic operations in CAF 2.0 are optimized to avoid the overhead of marshaling and de-marshaling. They also avoid the overhead of invoking asynchronous progress engine because they will be executed directly within active message handlers.

Optimistic execution

As a optimization, we adopt the idea of Optimistic Active Messages [23]. When a shipped function arrives, the image first *optimistically* assumes that the function is lightweight and non-blocking, and executes it directly within an active message handler. If the assumption is wrong — the function does not complete within a short period of time, or attempts to invoke a blocking subroutine, its execution state will be saved into an `async_record_t` for later execution. Optimistic execution of shipped functions avoids the overhead of asynchronous engine for lightweight shipped functions. The technique we used to generate continuations for shipped functions mimics Cilk's compilation strategy of `spawn` functions [25]. In particular, the CAF 2.0 compiler generates a continuable copy of the procedure which accepts a `frame` argument. The `frame` structure contains a `entry_point` field and the procedure's local variables. A function's frame is created and initialized when the optimistic assumption fails. For functions that cannot be executed within the active message handler, its continuable copy is used as the progress function in the asynchronous engine instead.

4.1.4 Deadlock-free execution

Compared with Active Messages [11], function shipping in CAF 2.0 gains its advantage in its ability to communicate with remote images and make blocking calls. In shared memory programming languages such as Cilk [25], a spawned thread gains it power in expressiveness by having the same capability with regard to accessing shared memory, spawn new threads, and communicate with other threads. In CAF 2.0,

```
program
    ...
    call team_barrier()
    if (my_rank == p) spawn foo(...)[q]
    if (my_rank == q) then
        ...
        call event_notify(e)
    end if
    ...
contains
    subroutine foo(...)
        call event_wait(e)
        ...
    end subroutine
end program
```

Figure 4.3 : Example of deadlock caused by blocking shipped function.

shipped functions can perform a full range of operations such as accessing coarray located remotely and spawning more functions to other processes.

However, allowing spawned functions to block, can cause deadlock in certain circumstances even when the program is semantically correct. Figure 4.3 shows an example of this case. In CAF 2.0, shipped functions are executed when the local process makes blocking calls to operations such as *barrier*, *event_wait*, etc.. In this example, if the spawned function from image `p` is received by image `q` before *q* exits from the *barrier*, the spawned function `foo` will be executed and *q* will block, waiting for event `e`. Since event `e` can only be notified by *q* and that has not yet happened. This causes a deadlock.

To avoid deadlock caused by executing blocking calls within a shipped function, for each shipped function that might block, the CAF 2.0 translator generates a *continuable* copy of the procedure for it. A continuable copy converts every blocking operation in the procedure into its corresponding non-blocking version. The continuable copy stores local variables on the heap and records the program counter before entering a non-blocking call, then makes the non-blocking call and saves itself in the pending operations list of the asynchronous engine. Later, when the asynchronous engine is invoked, it resumes the procedure if the non-blocking call has completed. The continuable copy of a procedure removes itself from the execution stack and schedules itself for resumption later when a blocking call is undergoing; thus avoids blocking the current thread. By providing continuation, we are able to overlap the communication cost incurred by shipped functions with local computation, yet also ensure that the execution of shipped functions is dead-lock free.

4.2 Finish

Adding function shipping into an SPMD language such as CAF 2.0 enriches the set of algorithms that can be expressed efficiently in SPMD fashion. However it also raised new challenges in synchronizing these asynchronous operations efficiently. Similar synchronization problems exist in other programming languages that support spawning asynchronous activities. Cilk provides the `sync` statement to await termination of all previously spawned threads. Since it executes on a shared memory machine, a simple algorithm to keep track of active child threads suffices. In X10, global completion of activities spawned by `async` statements are managed by a `finish` block. Although X10 supports a distributed memory model through the concept of *place* [40], a simple algorithm to determine global termination is still possible. Because in X10 all asynchronous activities from a `finish` block originate from the same place that spawns the `finish`, their global completion can be monitored by the image owning the `finish`.

In CAF 2.0, functions are spawned from arbitrary processes within a team, and thus no single process has the complete knowledge of the global state. This leads to two problems: 1) after each process has finished its local work, it cannot determine by itself whether there will be incoming work from other processes in the same team; 2) similarly, an image can't determine whether a remote image has quiesced because of the reason stated in 1). To solve these two problems, we propose a global termination algorithm which detects global completion in collectively in $O((L+1)\log(p))$ time; where p denotes the number of images in the team, and L denotes the length of the longest chain of spawns that occurs in the dynamic instance of `finish` scope.

In this section, we first introduce the semantics of CAF 2.0's `finish` construct. Then we discuss the computation model of CAF 2.0, and compared it with Cilk's *fully-strict* model and X10's *terminally-strict* model. After that, we present the algorithm for detecting global completion used by `finish`, and describe implementation details of the algorithm. Finally, we prove this algorithm correctly detects termination and establish this algorithm's theoretical upper bound with respect to communication.

4.2.1 Semantics

Finish construct is present in CAF 2.0 as a `finish` block, marked by `finish` and `end finish` statements, as shown in Figure 4.4. `finish` blocks are associated with *teams*; they work as collective operations. Every image within the associated team needs to create a finish block that matches those of its teammates. `finish` blocks can be nested and the team associated with the nested block can be a different team of its parent `finish` block. This can be useful for computation on a large multi-dimensional matrix distributed across a 2D processor grid. A structure of a finish block on row-wise team (line 5) nested within another finish scope on column-wise team (line 3)

```
1  copy_async(A(:), B(:)[p])
2  ...
3  finish (col_team)
4      spawn foo(...)[p]
5      ...
6      finish (row_team)
7          ...
8          copy_async(C(:), D(:)[q], ev)
9          ...
10     end finish
11 end finish
12 ...
13 call event_wait(ev)
```

Figure 4.4 : Example of nested `finish` scopes and interaction with `event` objects.

allows the operations on line 4–5 to be overlapped with the computation on line 7–9. when using `finish` to control the completion of asynchronous collective operations, the team associated with an asynchronous collective operation has to be a subset of the team of enclosing `finish` block to help correctly determine the completion of these asynchronous collectives.

Models of completion

CAF 2.0 provides two models of asynchrony to control an asynchronous operation: the *explicit* and *implicit* models. In the *explicit* model, programers use `event` objects associated with asynchronous operations to monitor when an certain type of completion occurs or to trigger an pending operation. For example, a `copy_async` statement in CAF 2.0 has three optional event arguments: an event indicating the operation may proceed, an event indicating that the source may be overwritten, and an event indicating global completion. This enables an expert user to write an event-driven processing loop that cooperates with other images by triggering asynchronous operations and reacting to specific events. The *explicit* model gives user full control over asynchronous operations in CAF 2.0.

An alternative way of controlling asynchronous operations is by using an *implicit* model for completion. A user can simplify programming using `finish` blocks to synchronize all asynchronous operations spawned within. All asynchronous operations within a `finish` block are guaranteed to be complete before the program exits from the `finish` block. However, we note that users are allowed to mingle the two models of asynchrony together by ensuring the completion of all activities using a `finish` block and, at the same time, precisely controlling the completion of some asynchronous operations using `event`.

Asynchronous operations that occur outside all explicit finish blocks and have no events bound with them (line 1) will all be captured before program exits, because the runtime system encloses the entire program execution inside an implicit global finish block on team `team_world`.

4.2.2 CAF 2.0 computation model

Blumofe et al. [41] defined the computation model generated by Cilk named *fully-strict* computation. A *fully-strict* computation is a multithreaded computation in which a parent task must wait for the completion of its children tasks before it exits. Agarwal et al. [42] extended the *fully-strict* model in X10. A `finish` block in X10 creates a dynamic scope in which users may spawn activities using the `async` primitive. Asynchronous activities spawned from a `finish` block must complete before the program exits from the `finish` block, but a descendant activity is allowed to continue executing even if its parent activity has terminated. This computation model in X10 is known as the *terminally-strict* model. Both *fully-strict* and *terminally-strict* com-

```
subroutine foo1()
    ...
    spawn foo2()[mod(team_rank() + 2, team_size())]
    call work() ! a call that can block
end subroutine foo1
subroutine foo2()
    ...
    call work() ! a call that can block
end subroutine
...
finish
    ...
    spawn foo1()[mod(team_rank() + 1, team_size())]
    call work() ! a call that can block
end finish
```

Figure 4.5 : An example of CAF 2.0 code fragment.

putations are multithreaded computation, where their task spawn trees have single roots, thus their computations can be represented as a directed acyclic graph.

The `finish` construct in CAF 2.0 differs from the construct with the same name in X10. Each `finish` block in CAF 2.0 is associated with a `team`; a `finish` works as a collective operation. Shipped functions from the same `finish` block may originate from multiple process images. We depict this computation model as a *space-time diagram*. In a space-time diagram, each horizontal line corresponds to a process image of a CAF 2.0 program. Activities in CAF 2.0 are represented as *dots* in the space-time diagram. The order that dots appear on a horizontal line from left to right reflects the order that these activities happen in real time in the process image. Figure 4.6 shows an space-time diagram for a `finish` computation in CAF 2.0; Figure 4.5 lists its corresponding code. Activities belong to the same shipped function or are local to a `finish` block are tagged with different labels to distinguish them from each

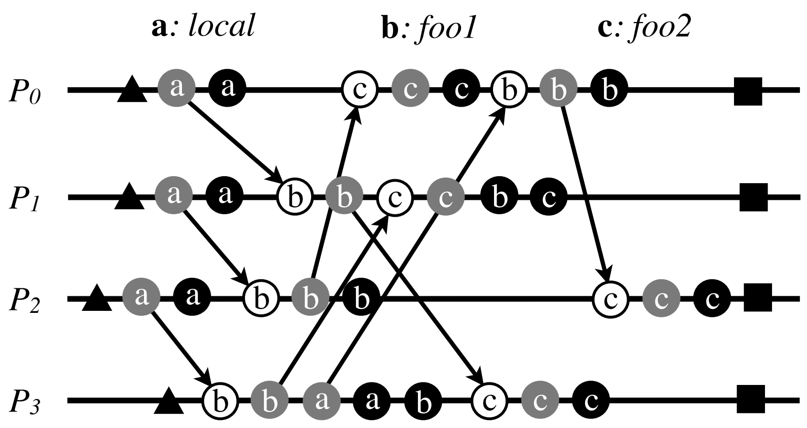

Figure 4.6 : CAF 2.0 computation diagram example.

other. In the `finish` computation model of CAF 2.0, three kinds of activities are of particular interest to us: the first operation in a shipped function that we call *start* operation, `spawn` operations that spawn new functions, and completion of a local computation or a shipped function. A *start* operation, represented as a *hollow dot* in a space-time diagram, marks the time a shipped function is received; the completion of a local operation or a spawned function is represented as a *solid dot*. A `spawn` operation is represented as a *grey dot*; the edge from a grey dot on one horizontal line to a hollow dot on another horizontal line is called a *spawn edge*. For convenience, in Figure 4.6 we omitted dots that represent activities other than the `start`, `spawn`, and completion in computation local to a `finish` block or shipped functions. The triangle and rectangle on each line of a space-time diagram respectively mark the start and end of a `finish` block in CAF 2.0. Because a program must wait for all shipped functions within a `finish` block to complete before it exits from the block, a rectangle must appear on each horizontal line after all shipped functions spawned after the proceeding triangle. Activities from different shipped functions or local computation are marked with different labels. Note that because shipped functions in CAF 2.0 obey the *run-until-block* semantics — a shipped function's execution will not be interrupted until it performs a blocking operation; another shipped function may start execution when a shipped function is blocked. Operations of a shipped functions may be interleaved or nested within operations of another shipped function in a space-time diagram. For example, in the space-time diagram in Figure 4.6, calls to subroutine `work`, which is an subroutine that might block, cause the execution of function `foo1` interleaved with function `foo2` on process $P1$, and the local computation on $P3$ be nested within the shipped function `foo1`. To simplify writing, we use the term *messages* to refer to shipped functions in a `finish` computation in later sections of this chapter.

4.2.3 Algorithm for termination detection

A vast number of distributed termination [29] algorithms have been proposed. However, many any of them suffer from two shortcomings which make them undesirable for large-scale parallel machines. Some algorithms make assumptions about their underlying system or communication layer such as a consistent global time across the system [36, §6.1], a synchronous communication model [30], or message channels that obey the First-In-First-Out (FIFO) rule [32], etc. These requirements are often costly to achieve on scalable parallel architectures. Also, many algorithms require storage that grows at least linearly with the number of processes [31][36, §6.2, §7]. Considering the fact that today's large-scale machines have tens of thousands of cores these algorithms will have a large memory footprint.

Hence, we develop a new termination detection algorithm that fits the architecture of today's large-scale parallel systems. Our algorithm is based on a simple but *incorrect* algorithm described in Mattern's paper [36]. In Mattern's algorithm, each process has a two-component vector to count the number of sent and received messages. It detects termination repeatedly by performing sum-reductions on the difference of the two components of the vectors on each process in a component-wise manner. Global termination is reached when the result of a sum-reduction is zero. This simple algorithm exhibits a good property: it uses only a small, constant space—an integer vector of size two—on each process.

Although Mattern's algorithm is space-efficient, it is flawed in two respects: correctness and time-efficiency. The algorithm is *incorrect* as mentioned by Mattern himself in his paper [36], because the vectors collected from each process could be an *inconsistent time cut* of the system. An inconsistent time cut could cause the vectors collected from each process to sum-reduce to zero even when there are still ongoing computations. In this case the algorithm falsely detects termination. This inconsistent time cut problem is discussed in detail in a later section.

With respect to time efficiency, this algorithm may perform poorly in practice because it does not specify when to retry the detection after receiving a non-zero value from a sum-reduction. If a new round of detection is initiated immediately every time after the last round of detection fails, it may use a potentially unbounded number of rounds before reaching termination. If a process waits too long before starting a new round of termination detection, it may cause unnecessary waiting when a termination state is reached.

Figure 4.7 lists the termination algorithm used in the `finish` construct in CAF 2.0. We fixed the inconsistent time cut problem by using *epochs* to manage updates to the counters (line 12 and 14) in our algorithm. And we enforced a condition that has to be satisfied before initiating a new round of termination detection: an image needs to wait until all shipped functions it spawns are received and all shipped functions

```
1     struct epoch {
2       int sent, delivered, received, completed;
3     }
4
5     epoch e, e1; //global var; even and odd epoch counters
6
7     function detect_termination (epoch e, team t)
8     {
9       while (work_left) {
10        blockuntil(e.sent == e.delivered
11            && e.completed == e.received);
12        if (not_odd(PRESENT_EPOCH)) e1 = next_epoch(e);
13        allreduce(sum, e.sent-e.completed, work_left, t);
14        e = next_epoch(e1);
15      }
16    }
```

Figure 4.7 : The termination detection algorithm in `finish`.

it received must be completed before the image performs a new sum-reduction (line 13). This prerequisite condition, as proven in Section 4.2.4, reduces the number of rounds of detection performed, and detects termination at the earliest possible time. In our algorithm, an `epoch` structure contains four integers that counts the number of functions an image has sent, completed, and received locally as well as the number of functions it has delivered remotely. The `allreduce` operation is a collective all-to-all sum-reduction. Each image waits at an `allreduce` until all images in team `t` have arrived at the same `allreduce`.

Dealing with inconsistent time cuts

In a space-time diagram described in Section 4.2.2, a *time cut* is defined as a curve that crosses each horizontal line exactly once. A time cut is considered to be *inconsistent*, if after the cut an image spawns a message that “travels back in logical time” and

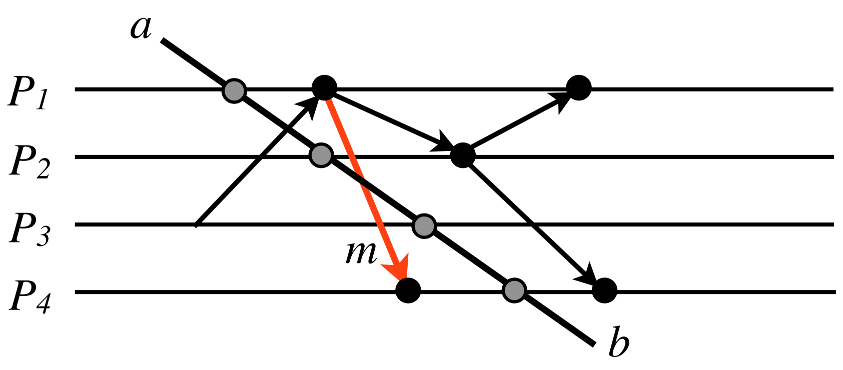

Figure 4.8 : An example of inconsistent time cut.

lands before the cut. Figure 4.8 shows an example of inconsistent time cut. The message m travels from a time after cut ab and lands before the cut, thus cut ab is an inconsistent time cut. When collecting counters from each image in a team using the `allreduce` operation in our algorithm, the time at which each image provides its local counters to the `allreduce` can be connected with a curve that crosses each horizontal line to form a time cut across the team. Consider the example depicted in Figure 4.8, a shipped function is received by image P_1 after P_1 has contributed his counters to his communication partners in an `allreduce` then spawns two more functions. One of the children "travels back in time" and lands on P_4 before P_4 participates the `allreduce`. The counter values collected by this `allreduce` on P_1, P_2, P_3, P_4 are $(0, 0), (0, 0), (1, 0), (0, 1)$, respectively. Although the result of sum-reduction over these counters is zero, the system has not terminated yet.

We tackle the inconsistent time cut problem by introducing the notion of *epochs*. We divide interval in time between when a `finish` scope starts and the time when the `finish` computation ends into a series of epochs. These epochs are numbered one after another starting from zero; we call them even and odd epochs based on whether its sequence number is even or odd. Thus, each round of reduction consists of two epochs: an even epoch and an old epoch. An image proceeds from an even epoch into an odd epoch when 1) it enters an `allreduce`, or 2) when it receives a message from an image in an odd epoch. An image proceeds from an odd epoch into the next epoch when it exits an `allreduce`.

Each epoch contains a set of counters, as listed in Figure 4.7. A `finish` scope counts the number of functions it shipped, delivered, received, and completed locally using the set of counters in the epoch the image presents. For example, if a shipped function is received in epoch n and is completed in epoch $n+1$, the counter `received` in epoch n notes the shipped function's reception and the counter `completed` in epoch $n+1$ notes the completion of the shipped function. In the beginning, all the counters in epoch 0 are initialized to zero. Then the counters in a new epoch are initialized to the value of corresponding counters from the last epoch.

When an image participates an `allreduce` it always provides to the `allreduce` the counters from the even epoch; counters in odd epochs are used to track functions sent, received, delivered, and completed during an `allreduce`. Because a message sent from an odd epoch will cause the receiver image to proceed into the odd epoch, the sending, reception, delivery and completion of a shipped function are all counted by counters in the odd epoch. Therefore, functions shipped during a reduction are not counted in that `allreduce`; they will be included in the next round of reduction. This property ensures the time cut constructed using an `allreduce` is consistent and guarantees our algorithm is correct, as proven in Section 4.2.4.

Because an image can only present in one epoch at any time and only counters from the previous epoch are needed to initialize counters in the present epoch, we need only two sets of counters for each `finish` scope on each image. Using epochs to

```
struct finish_scope_t {
    finish_handle_t handle;
    epoch_t epoch[2];  //even and odd epoch
    team_t team;
}
```

Figure 4.9 : `finish_scope_t` structure used in the `finish` construct.

manage counters in our algorithm enables us to maintain a constant space requirement on each image for each `finish` scope.

Managing finish scopes

Figure 4.9 shows the `finish_scope_t` structure used by the `finish` construct. Each `finish` block in CAF 2.0 is represented as an instance of `finish_scope_t` at runtime. Finish scopes are dynamically created at the time a CAF 2.0 program enters a `finish` block, and destroyed when program exits from the block. Finish scopes are managed using a *splay tree* data structure [43]. A splay tree is a self-adjusting binary search tree which exploits data access locality that enables recently accessed elements to be accessed again cheaply. Each time a node n in a splay tree is accessed, a *splay* operation rearranges the tree so that node n is placed at the root of the tree. The *splay* operation is performed every time a node is accessed, causing nodes that are frequently accessed move nearer to the root. Splay trees performs insertion, look-up and removal operations in $O(\log n)$ amortized time. Because most frequently an image accesses and operates on the innermost finish scope, using a splay tree to manage finish scopes minimizes the cost of accessing the innermost finish scope, yet it also enables accessing random finish scopes efficiently as well.

4.2.4 Proof of termination and communication bound

In this section, we prove our algorithm correctly detects termination of a distributed computation defined by a `finish` block. Then, we establish a theoretical upper bound for our algorithm with respect to the communication time for termination detection.

Proof of termination

We define that the computation of a `finish` scope terminates at time t when all messages spawned before t have completed by t and computation local to the `finish` block has completed. Let m denote a message in a CAF 2.0 program and M be the set of all messages. We define sets

$$M_s(i) = \{m : m \textit{ is spawned by image } i\} \textit{ and}$$
$$M_c(i) = \{m : m \textit{ is completed by image } i\}.$$

Each message in a CAF 2.0 program that uses the implicit model for completion is tagged with a handle of its enclosing `finish` block; each `finish` scope uses a different handle to tag its messages. If F is the computation belongs to a `finish` block, $M|F$ denotes the projection of set M on `finish` block F. Because our algorithm detects termination of computation in a `finish` block, though multiple nested instances of `finish` can be alive simultaneously, our discussion focuses only on detecting termination of messages that originate from one `finish` block. For this reason, we omit the "$|F$" notation for all definitions and references describing a set of messages in later section for simplicity.

Let $T_s(m)$ denotes the time when message m is spawned, and $T_c(m)$ the time m is completed. In our algorithm each round of termination detection uses two epochs:

an even epoch and an odd epoch. In a system with n images, we denote as $t(i)$ the time when image i, $i \in \{1, 2, ...n\}$, transits from the even epoch to the odd epoch. We define

$$\begin{aligned} S_i &= \{m \in M_s(i) : T_s(m) < t(i), i = 1...n\}\,, \\ C_i &= \{m \in M_c(i) : T_c(m) < t(i), i = 1...n\}\,,\ \textit{and} \\ O_i &= \{m \in S_i : T_s(m) > t(i), i = 1...n\} \end{aligned}$$

Then, we define

$$\bar{S} = \bigcup_{i=1}^{n} S_i \ \textit{and}\ \bar{C} = \bigcup_{i=1}^{n} C_i.$$

Finally, define

$$\begin{aligned} S(t) &= \{m : T_s(m) < t\}\,,\ \textit{and} \\ C(t) &= \{m : T_c(m) < t\}\,. \end{aligned}$$

In the following proof, we prove that when the *epoch* technique described in last section is applied, if a global sum-reduction over the difference of the count of messages spawned and the count of messages completed returns zero, that is $\sum_{i=1}^{n}(|S_i| - |C_i|) = 0$, the computation in a `finish` block has completed before the global reduction completes.

Lemma 4.1. *If $m \in O_i$ and $m \in M_c(j)$, $t(j) < T_c(m)$.*

Proof. Follows from the property of *epochs*: a message spawned from an odd epoch will cause its target image to enter an odd epoch when it is received. □

Lemma 4.2. *If $|\bar{S}| = |\bar{C}|$, then $\bar{S} = \bar{C}$.*

Proof. We prove by contradiction.

If $|\bar{S}| = |\bar{C}| \wedge \bar{S} \neq \bar{C}$, $\exists m$, s.t. $m \in \bar{C}$ and $m \notin \bar{S}$.

Let p be the image that completes m, so $m \in C_p$, and

$$T_c(m) < t(p). \tag{4.1}$$

Let q be the image that spawns m, so $m \in M_s(q)$.

Because $m \notin \bar{S}$, then $T_s(m) > t(q)$, so $m \in O_q$.

From Lemma 4.1, we have $T_c(m) > t(p)$, which contradicts (4.1). □

Lemma 4.3. *If* $\bar{S} = \bar{C}$*,* $S(t_{end}) = C(t_{end})$*, where* $t_{end} = max(t_i)$*,* $i = 1...n$*.*

Proof. We prove $S(t_{end}) = C(t_{end})$ by showing that $\bar{S} = S(t_{end})$ and $\bar{C} = C(t_{end})$. We prove $\bar{S} = S(t_{end})$ by contradiction.

Assume $\bar{S} = \bar{C} \wedge \bar{S} \neq S(t_{end})$, $\exists m$, s.t. $m \in S(t_{end}) \wedge m \notin \bar{S}$. Let i be the image that spawns m, we have

$$t(i) < T_s(m) < t_{end}. \tag{4.2}$$

Let $\acute{m}$ be the message that has a spawn edge connects to m, we know it is true that

$$t_c(\acute{m}) > t_s(m). \tag{4.3}$$

From (4.2) and (4.3), we have $t_c(\acute{m}) > t(i)$, which implies $\acute{m} \notin \bar{C}$. Then we consider two scenarios:

1) If $\acute{m} \in \bar{S}$, $\bar{S} \neq \bar{C}$. This contradicts our assumption.

2) If $\acute{m} \notin \bar{S}$, we choose $\acute{m}$ as m and repeat the above process until a $\acute{m}$ s.t. $\acute{m} \in \bar{S}$ is found. This also implies $\bar{S} \neq \bar{C}$. Contradiction.

Thus, $\bar{S} = S(t)$.

$\bar{C} = C(t)$ can be proven with the same method. □

Lemma 4.4. *If* $\sum_{i=1}^{n}(|S_i| - |C_i|) = 0$, $\bar{S} = \bar{C}$.

Proof. $\sum_{i=1}^{n}(|S_i| - |C_i|) = \sum_{i=1}^{n}|S_i| - \sum_{i=1}^{n}|C_i|$.

Because $\forall m \in S_i$, $m \notin S_x$, where $x \neq i$, And $\forall n \in C_i$, $n \notin C_x$, where $x \neq i$, we have $\sum_{i=1}^{n}|S_i| = |\bigcup_{i=1}^{n} S_i| = \left|\bar{S}\right|$, and $\sum_{i=1}^{n}|C_i| = |\bigcup_{i=1}^{n} C_i| = \left|\bar{C}\right|$.

Thus, $\sum_{i=1}^{n}(|S_i| - |C_i|) = 0 \Leftrightarrow \left|\bar{S}\right| - \left|\bar{C}\right| \Leftrightarrow \left|\bar{S}\right| = \left|\bar{C}\right|$

From Lemma 4.2, we have $\bar{S} = \bar{C}$. □

Theorem 4.1. *If* $\sum_{i=1}^{n}(|S_i| - |C_i|) = 0$, *computation in a* ***finish*** *block in CAF 2.0 terminates by the time* t_{end}.

Proof. From Lemma 4.4 and Lemma 4.3, $\sum_{i=1}^{n}(|S_i| - |C_i|) = 0 \Rightarrow \bar{S} = \bar{C} \Rightarrow S(t_{end}) = C(t_{end})$, which means all messages spawned before t_{end} have completed by t_{end}. Each image enters termination detection only after it completes computation local to the `finish` block, thus at t_{end} local computation completes. Hence, computation in the `finish` block in CAF 2.0 terminates by t_{end}. □

This completes the proof of correctness of our termination detection algorithm.

Communication bound

This section proves that the termination detection algorithm described in Section 4.2.3 takes $O((L+1)\log(p))$ communication time in the worst case, where p is the number of processes and L is the length of longest spawning chain of shipped functions. One can perform an all-to-all reduction in $O(\log p)$ time using a reduction tree and a broadcast tree. Here, we prove that our algorithm uses at most $L+1$ rounds of sum-reduction.

Theorem 4.2. *The algorithm listed in Figure 4.7 uses at most $L+1$ rounds of sum-reduction to detect termination, where L is the length of longest spawning chain of shipped functions.*

Proof. We prove by induction.

Base case $L = 0$ means no functions are shipped within a `finish` scope. In this case the number of messages sent, received, delivered and completed are $(0, 0, 0, 0)$ on each image. These counters will sum-reduce to zero after they are collected by an `allreduce` operation. Thus, one `allreduce` detects termination correctly.

Inductive hypothesis Assume that when the longest chain of spawned functions in a `finish` scope is L, at most $L + 1$ rounds of reductions are needed to detect termination.

Inductive step We must show that at most $L+2$ rounds of reductions are needed to detect termination when the longest chain of spawned functions is $L + 1$.

$t_i(k)$ denotes the kth time when image i switches from an even epoch to its corresponding odd epoch. $t_s(m)$, $t_r(m)$, and $t_c(m)$ denote the time when a shipped function m is sent, received, and completed respectively.

In a `finish` scope where the longest chain of spawned functions has length $L+1$, at the time that the $L+1$ round of reduction completes, the longest chain of shipped functions that has been spawned is equal to or larger than L (L or $L+1$); otherwise the system would have terminated in an earlier round by the inductive hypothesis. If it is $L + 1$, termination is detected successfully with $L + 1$ rounds of reduction. If it is L, we know that all shipped functions have completed, except the shipped function that is the last function in the chain of length $L+1$. We denote the shipped function that has yet to complete as m. Let i be the image sends m, and j be the image completes m. We know that $t_c(m) > t_j(L+1)$. In our algorithm, because an image waits all messages it sends to land before it enters `allreduce`, we have $t_r(m) < t_i(L+1)$. Then, because an image completes all messages it receives before it starts another round of reduction, we have $t_c(m) < t_j(L+2)$. Because m is the only message left after $L+1$ reductions, the $L+2$ reduction detects termination.

Since both the basis and the induction step have been proven, Theorem 4.2 is proven. □

Theorem 4.2 combined with the fact that an all-to-all reduction uses $O(\log p)$ time shows that our termination detection algorithm takes at most $O((L+1)\log p)$ time when the longest chain of spawned functions is L.

Nested `finish` blocks

In this section we prove that our previously proven results for our termination detection algorithm with respect to both correctness and performance apply to a `finish` block nested inside another `finish` block, and a `finish` block that contains a nested `finish` block.

Let F_i, $i = 1, 2, ...n$ denote a series of nested `finish` blocks; F_1 is the innermost block and F_n is the outermost one.

Theorem 4.3. *The algorithm listed in Figure 4.7 correctly detects termination of `finish` blocks* F_i, $i = 1, 2, ...n$.

Proof. We prove that our algorithm correctly detects termination of F_i by induction.

Base case Since we tag each message sent from a `finish` scope with its handle. We apply the algorithm in Figure 4.7 to only messages tagged with the handle of

F_1 and detects termination of F_1. By Theorem 4.1, we detect the termination of F_1 correctly.

Inductive hypothesis If the algorithm in Figure 4.7 detects termination of F_n, it can also detects termination of F_{n+1}.

Inductive step By the time that the program reached the `end finish` statement of F_{n+1}, it has already exited from F_n. Thus, F_{n+1} is the innermost `finish` block when the termination detection algorithm starts execution. By Theorem 4.1, our algorithm detects the termination of F_{n+1}. $\square$

Finally, Theorem 4.2 which establishes an upper bound on the number of rounds of sum-reduction used by a `finish` also holds when nested `finish` blocks exits. This can be easily proven by applying the same logic as in Theorem 4.3.

Chapter 5

Evaluation

Here, we evaluate the performance of our implementation of function shipping and the `finish` construct in CAF 2.0 with three benchmarks: pingpong, RandomAccess, and Unbalanced Tree Search (UTS). We describe these benchmarks in detail in Section 5.2. These three benchmarks evaluate different aspects of the performance of function shipping and the `finish` construct in CAF 2.0. We start by introducing the experiment platforms that we used and then present our evaluations with each of these three benchmarks.

5.1 Platforms

We experiment on three platforms with different characteristics. The first one is *Jaguar* in the Oak Ridge National Lab Leadership Computing Facility, which is the newest Cray XK6 supercomputer. The second one is *Franklin* in National Energy Research Scientific Computing Center, which is a Cray XT4 system. The third one is *STIC*, a Intel Core i7 based computing cluster at Rice University. Table 5.1 lists the configuration of these systems in detail.

Table 5.1 : Configuration of experiment systems.

	Jaguar	Franklin	STIC*
Architecture:	Cray XK6	Cray XT4	Appro Greenblade E5530
Processor:	16-core AMD	4-core 2.3GHz AMD	4-core 2.66GHz Intel
Nodes:	18,688	9,572	170
Cores/node:	16	4	8
Total cores:	299,008	38,288	1,360
Memory/node:	32GB	8GB	12GB
Interconnect::	Gemini	SeaStar	4X DDR Infiniband

* STIC also includes 44 Appro Greenblade E5650 compute nodes each with two six-core 2.6GHz Intel Xeon Processors. But these nodes are not used in our experiments.

5.2 Benchmarks and evaluation

This section presents each of our three benchmarks, Pingpong, RandomAccess and UTS, describes their implementation, evaluates their performance.

Each benchmark serves a purpose. We use the pingpong benchmark to measure the overhead of our implementation of function shipping. We use RandomAccess to explore the utility of function shipping for applications with irregular data access patterns. We use the UTS benchmark to exploit the utility of function shipping for managing load balancing in applications with dynamically generated parallelism. We describe and evaluate each of these benchmarks in the following sections.

```
1  program pingpong
2      integer :: my_rank, my_partner, counter, i
3
4      my_partner = 1 - team_rank()
5      counter = 1000000
6
7      if (my_partner .ne. 0) spawn ping()[my_partner]
8
9      do while (counter .ne. 0)
10         call caf_async_advance()
11     enddo
12 contains
13     subroutine ping()
14         counter = counter - 1
15         spawn pong()[my_partner]
16     end subroutine
17
18     subroutine pong()
19         counter = counter - 1
20         if (counter .ne. 0) spawn ping()[my_partner]
21     end subroutine
22 end program
```

Figure 5.1 : Source code for Pingpong benchmark implemented with function shipping in CAF 2.0.

5.2.1 Pingpong benchmark

To measure the performance of our implementation of function shipping in CAF 2.0, we developed a Pingpong benchmark that tests latency of shipping functions in CAF 2.0. Figure 5.1 lists the source code for implementation of Pingpong benchmark using function shipping. With function shipping, the receiver of the `ping` function does not need to synchronize with the sender to initiate the function `pong`. In our implementation, the function `ping` spawns the function `pong` after it decrements the counter (line 14) and function `pong` spawns `ping` back. This implementation is different from the usual way of implementing the Pingpong benchmark, where one process pings the other then waits until it receives a pong back to ping again. Our implementation removes the synchronization cost of waiting for pong.

It is worth noting that an Note that an implicit `finish` block that encloses the entire program will complete these shipped functions before program exits. This implicit `finish` block can correctly detect termination of the program without having the wait loop at line 10. Because the number of reductions used for termination detection is bounded by the maximum length of the spawning chains, which in this case is one million, one million reductions may interleave with the pingpong benchmark when it executes. Since the focus of this benchmark is to test the roundtrip latency of function shipping, we use the wait loop to avoid incurring the overhead of `finish`.

Figure 5.2 shows a comparison of function shipping with Active Messages in GASNet and the `MPI_send` & `MPI_recv` routines in MPI. Because of our implementation of function shipping is implemented using Active Messages in GASNet, the implementation of pingpong benchmark using Active Messages in GASNet provides an upper bound on the performance that we can possibly achieve. We also show the performance of an implementation of pingpong using MPI's `MPI_send` & `MPI_recv` alongside the AM and function shipping implementations to demonstrate how our implementation of function shipping performs compared to these other kinds of communication. From Figure 5.2, we see that the function shipping implementation above Active Messages in GASNet takes about 90% more time than the implementation using Active Messages and MPI. Analyzing the results on STIC and Jaguar with HPCToolkit [44] shows that the local overhead of function shipping implementation of pingpong takes about 17% of the total running time on STIC and 28% on Jaguar. This local overhead accounts for time marshaling and de-marshaling the shipped function and managing

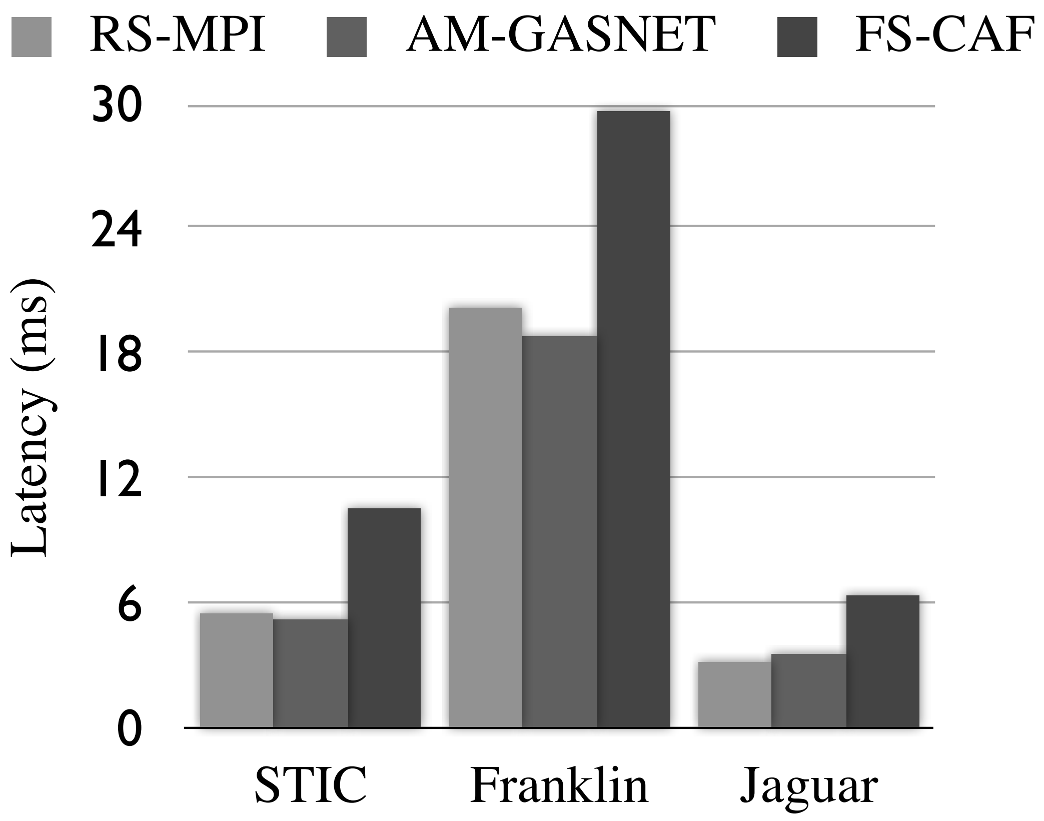

Figure 5.2 : A comparison of the roundtrip latency of a ping-pong pair implemented using function shipping, Active Messages in GASNet, and `MPI_send` & `MPI_recv`.

shipped functions using the CAF 2.0 asynchronous engine. Other than local overhead, the increase of latency in the function shipping implementation is due to the cost of transferring shipped functions and the data used to manage these shipped functions.

5.2.2 RandomAccess benchmark

The HPC Challenge RandomAccess benchmark evaluates the rate at which a parallel system can apply updates to randomly indexed entries in a distributed table. Performance of the RandomAccess benchmark is measured in Giga Updates Per second (GUP/s). GUP/s is calculated by identifying the number of table entries that can be

```
do j=1,1024
    finish
        do i=1,1024
            k = generate_random_number()
            p = image_of(k)
            i = offset_of(k)
            spawn remote_update(table(i)[p], i)[p]
        end do
    end finish
end do
```

Figure 5.3 : Pseudo-code for a implementation of RandomAccess benchmark in CAF 2.0.

randomly updated in one second, divided by 1 billion (10^9). The term "randomly" means that there is little relationship between one table index to be updated and the next. An update is a read-modify-write operation on a 64-bit word in the table. A table index is generated, the value at that index is read from memory, modified by an integer operation (`xor`) that combines the current value of the table entry with a literal value, and the resulting value is written back to the table entry. A detailed specification for RandomAccess benchmark can be found in HPC Challenge Benchmarks specification [45].

We compared two versions of implementation of RandomAccess in CAF 2.0: one implementation that uses function shipping to spawn a function to perform update remotely; the other implementation first reads (`GET`) an entry from a remote image, updates it locally, then writes (`PUT`) the updated value back. Figure 5.3 lists the pseudo-code for our implementation of RandomAccess with function shipping. To measure the performance of our `finish` algorithm, we grouped 2048/1024/512 updates in a `finish` block, so that 2048/4096/8192 instances of our `finish` algorithm will be invoked when we update a table of 2^{22} entries.

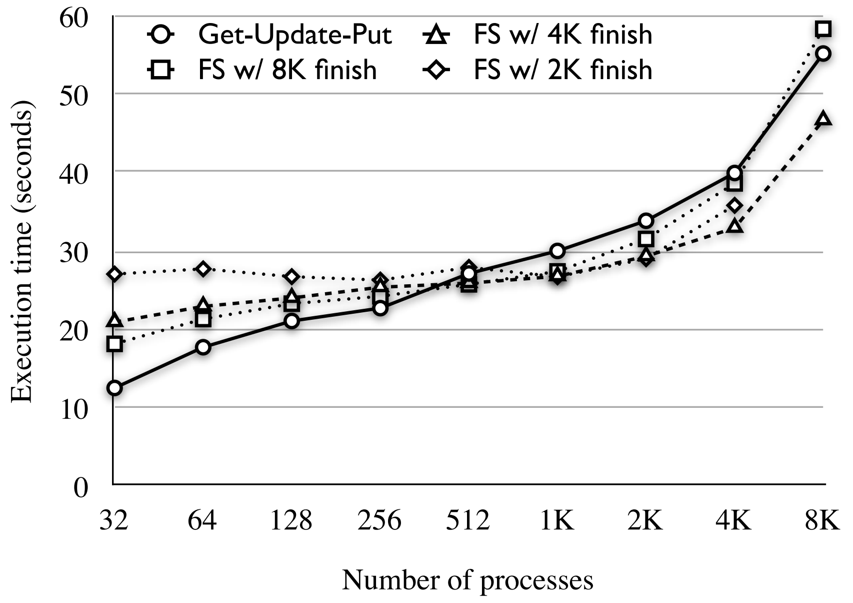

Figure 5.4 : A comparison of two implementations of the RandomAccess benchmark in CAF 2.0 (GP = GET & PUT; FS = function shipping) on Jaguar.

Figure 5.4 shows the performance of two implementations of RandomAccess benchmark. We measure the running time of applying remote update on a distributed table of size 8MB on each process image. From 32 to 4096 processes, we see that the implementation using function shipping performs better on more than 512 processes than than the implementation using `GET` and `PUT`. And we see that the cost of invoking our `finish` algorithm is not excessive because the running time is significantly different for different number of rounds of termination detection.

5.2.3 Unbalanced Tree Search benchmark

The UTS benchmark [12] performs an exhaustive parallel search on a deterministic, unbalanced search space. The UTS tree traversal starts with a single root node and proceeds in nested parallel style to generate billions of nodes. The number of children

```
!while there is work to do
do while(queue_count .gt. 0)
  delete_queue_end(descriptor , depth)
  call process_work_item(descriptor , depth)
  ...
  !check if someone needs work
  if ((incoming_lifeline .ne. 0) .and. &
    (queue_count .ge. lifeline_threshold)) call push_work ()
  endif
  ! attempting to steal work from another image
  steal_from_img = get_random_image_other_than_me(my_rank)
  spawn steal_work(my_rank, 0)[steal_from_img]
  ! set up lifelines
  neighbor_index = 0
  do while (neighbor_index .lt. max_neighbor_index)
    next_neighbor = mod(my_rank+(2**neighbor_index), world_size)
    spawn set_lifelines(my_rank, neighbor_index)[next_neighbor]
    neighbor_index = neighbor_index + 1
  enddo
end do
```

Figure 5.5 : Pseudo-code for the UTS benchmark in CAF 2.0.

of a node is a random variable with a given distribution. Each node in the tree contains a 20 byte descriptor. Descriptor of children node is created by applying the SHA-1 cryptographic hash on the pair $(parentdescriptor, childindex)$. Due to imbalance in the search space and the volume of nodes created, the performance of UTS depends heavily on efficient dynamic load balancing. Our implementation of the UTS benchmark in CAF 2.0 closely follows the implementation in X10 presented by Saraswat [33]. Function shipping in CAF 2.0 replaces the `async` construct in X10, and, instead of implementing a global termination detection algorithm in UTS, we use our `finish` construct. Figure 5.5 lists the pseudo code of the core of our UTS implementation in CAF 2.0. We use function shipping and the `finish` construct in

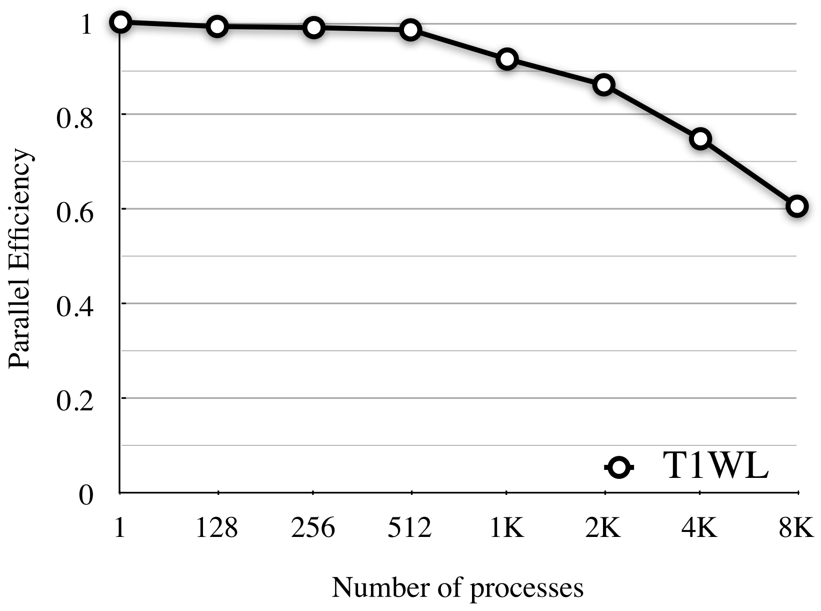

Figure 5.6 : Unbalanced Tree Search benchmark results.

CAF 2.0 to implement a UTS based on the T1WL UTS benchmark. The nodes in the tree of this benchmark uses a geometric distribution with expected child count per node of 4 and a maximum tree depth 18.

Figure 5.6 shows the parallel efficiency of our CAF 2.0 UTS implementation on various number of processors on Jaguar. For a tree with size of 270 billion nodes, our parallel efficiency is above 87% for up to 2K processes. Our scaling efficiency up to 1K processors is essentially equivalent of that of the original X10 implementation [33]. We show the scaling behavior between 1K and 8K processors for CAF 2.0; no information about the scaling efficiency of the X10 implementation is available for this range.

With our UTS implementation, we show that our `finish` construct is capable of managing termination detection in UTS. This demonstrates its utility in CAF 2.0. Moreover, our experimental results with UTS benchmark demonstrate that our `finish`

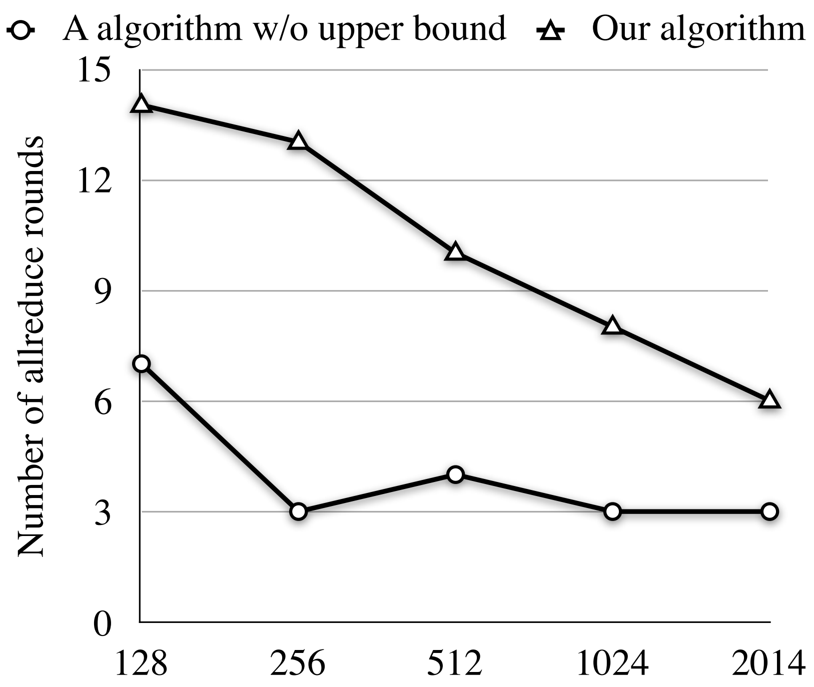

Figure 5.7 : Number of `allreduce` operations used for detecting termination in UTS

construct's implementation is scalable and efficient. In Figure 5.7, we show that enforcing an upper bound on the number of rounds of detection before termination is effective and beneficial. As Figure 5.7 shows, the number of `allreduce` operations used for detecting termination in our algorithm is about 50% the number of `allreduce` operations used by an algorithm that does not wait for delivery and completion of shipped messages before starting a round of termination detection from 128 to 2048 processors. However we see a trend that the difference in the number of rounds of termination detection reduces as the number of processes increases. This is because when the number of processes increases in the system it takes longer time for an `allreduce` to complete and the total running time is reduced.

Chapter 6

Conclusions and Future Work

Partitioned Global Address Space languages now are viewed as the most promising alternative to MPI to address the programming difficulty across compute nodes [1]. A critical way of achieving scalable performance on large parallel systems is hiding communication latency. Our group has explored extending enriches the Coarray Fortran language with a set of primitives including asynchronous copies and collective operations that enable users to hide communication latency with computation on large scale parallel systems [38]. This thesis explores function shipping as an additional mechanism that programmers can use to hide or avoid communication latency. Function shipping gives users the ability to co-locate computation with data. This helps avoid exposing latency in certain circumstances, as shown in the RamdomAccess benchmark. We now summarize the contributions of this thesis and also propose extensions to be pursued as part of future work.

CAF 2.0's support for function shipping is a full-featured construct that enables a shipped function to perform a full range of operations, including blocking remote operations. In particular, a shipped function can spawn more functions, communicate, and synchronize with other processes. These abilities enrich CAF 2.0 expressiveness by making possible to implement algorithms that do work-stealing and work-sharing for load balancing, and help avoid exposing latency by enabling users to co-locate computation with data.

To achieve an implementation of function shipping with such degree of flexibility, we used a compilation strategy similar to Cilk 5 [25], generating a continuable copy of the same function so that a function's stack and program counter can be saved onto the heap and resumed later. This allows a shipped function to yield its processor to another thread while it is waiting for communication or synchronization. More importantly, it eliminates the possibility of deadlock caused by trying to synchronize while blocking the same thread, as demonstrated in Section 4.1.4.

We adopt the idea of Optimistic Active Messages [23] to execute shipped functions in a optimistic way. When a shipped function is received by a process, it executes it directly in the active message handler until the function makes a blocking call. As demonstrated by Wallach et al., despite the gain in expressiveness over Active Messages, Optimistic Active Messages performs as well as Active Messages [23].

To synchronize shipped functions properly, we introduce a `finish` construct into CAF 2.0, inspired by the `finish` construct in X10 but adapted to fit into CAF 2.0's SPMD programming model. We developed a termination detection algorithm extended from a simple algorithm by Mattern [36]. This algorithm is design for scalability and has a nice upper bound on communication complexity. In this thesis we proved that our algorithm detects termination correctly using a bounded number of rounds of communication. Our experiments show that this upper bound can greatly reduce the number of rounds of `allreduce` operation used to detect termination on parallel architectures.

We demonstrate the usability and performance of function shipping and finish construct in CAF 2.0 with HPCC RandomAccess benchmark and Unbalanced Tree Search benchmark (UTS). Our implementation of RandomAccess yields scalable performance compared with a simple algorithm use only one-sided GET and PUT. Our implementation of UTS achieves parallel efficiency above 90% up to 4K cores, which is comparable to the best performance of the implementation of UTS in X10.

6.1 Future work

In pursuing of the goal of making writing applications with dynamic generated parallelism and irregular data access easier and efficient on large scale parallel systems, there are several directions worth further exploring and experiment. We outline a few directions here.

Combining computation with termination detection In the process of developing the UTS benchmark in CAF 2.0, we noticed that the problem of termination detection and load balancing are often bonded together. Since termination detection algorithms that based on message or channel counting communicates across all nodes through reduction or broadcast, the information about load on each process can also be communicated combined with the messages used for termination detection. In this way, the task of load balancing is embedded into termination detection so that total number of messages in the system during load balancing could be reduced to achieve better performance. This idea extends beyond combining only load balancing with termination detection. AM++ [22] supports a feature that enables a user to provide a user-defined integer to be summed and broadcast across the system along with termination detection. They argue that this feature is useful for many algorithms consist of a phase of active messages followed by a reduction or broadcast operation. We envision that more expressiveness could be achieved by allowing users to provide callback functions to be carried by the messages used for termination detection instead of just a single integer value. The callback functions provided to a termination detection construct can be used to accomplish work sharing or work stealing.

Improving thread support in CAF 2.0 Co-locating computation with data and utilizing intra-node parallelism is essential to fully utilize hardware capabilities of modern parallel architectures. Instead of making different processing cores within the same compute node different processes, using a single multi-threaded process on a node simplifies intra-node communication because of the shared address space. Efficiently scheduling computations generated within a node together with computations shipped across nodes is an interesting research direction to go after. Different applications may need different scheduling policies to best accommodate their concurrency needs. Because the difficulty of analyzing the parallel characteristic of a program by a compiler alone, allowing a user to specify the scheduling policy to use for a program may be necessary to achieve scalable performance on modern parallel systems.

Better support for dynamic load balancing in CAF 2.0 Many real applications and algorithms requires load balancing to achieve scalable performance on parallel machines. Many load balancing algorithms have been proposed to address this issue such as randomized work-stealing. Work-stealing works very well on machines that have hardware support for shared memory address. On distributed systems that does not have hardware support for shared memory, using work-stealing to efficiently address load balancing problems is still under study [46, 47, 33]. One promising way of efficient work-stealing on distributed systems is to use the information gathered from a termination detection protocol to direct which victim to select for stealing. This way may greatly increase the rate of successful stealing attempts.

Bibliography

[1] Message Passing Interface Forum, "MPI Report 2.0." `http://www.mpi-forum.org/docs/mpi-20-html/mpi2-report.html`.

[2] T. El-Ghazawi, W. Carlson, T. Sterling, and K. Yelick, *UPC: Distributed Shared-Memory Programming.* Wiley-Interscience, 2003.

[3] K. Yelick, L. Semenzato, G. Pike, C. Miyamoto, B. Liblit, A. Krishnamurthy, P. Hilfinger, S. Graham, D. Gay, P. Colella, and A. Aiken, "Titanium: A high-performance Java dialect," in *ACM 1998 Workshop on Java for High-Performance Network Computing*, (New York, NY 10036, USA), ACM Press, 1998.

[4] R. W. Numrich and J. Reid, "Co-array Fortran for parallel programming," *SIGPLAN Fortran Forum*, vol. 17, no. 2, pp. 1–31, 1998.

[5] P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. von Praun, and V. Sarkar, "X10: an object-oriented approach to non-uniform cluster computing," in *Proceedings of the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, OOPSLA '05, (New York, NY, USA), pp. 519–538, ACM, 2005.

[6] B. L. Chamberlain, D. Callahan, and H. P. Zima, "Parallel programmability and the Chapel language," *Int. J. High Perform. Comput. Appl.*, vol. 21, no. 3, pp. 291–312, 2007.

[7] R. J. Harrison, M. F. Guest, R. A. Kendall, D. E. Bernholdt, A. T. Wong, M. Stave, J. L. Anchell, A. C. Hess, R. J. Littlefield, G. L. Fann, and et al., "Towards High Performance Computational Chemistry: (II) A Scalable Self-Consistent Field Program," *Journal of Computational Chemistry*, vol. 17, no. 1, pp. 124–132, 1996.

[8] T. Yanai, G. I. Fann, Z. Gan, R. J. Harrison, and G. Beylkin, "Multiresolution quantum chemistry in multiwavelet bases: Analytic derivatives for Hartree–Fock and density functional theory," *The Journal of Chemical Physics*, vol. 121, no. 7, pp. 2866–2876, 2004.

[9] J. Mellor-Crummey, L. Adhianto, W. N. Scherer, III, and G. Jin, "A new vision for Coarray Fortran," in *PGAS '09: Proceedings of the Third Conference on Partitioned Global Address Space Programing Models*, (New York, NY, USA), pp. 1–9, ACM, 2009.

[10] D. Bonachea, "GASNet specification, v1.1," Tech. Rep. UCB/CSD-02-1207, University of California at Berkeley, Berkeley, CA, USA, 2002.

[11] T. von Eicken, D. E. Culler, S. C. Goldstein, and K. E. Schauser, "Active messages: a mechanism for integrated communication and computation," *SIGARCH Comput. Archit. News*, vol. 20, pp. 256–266, April 1992.

[12] S. Olivier, J. Huan, J. Liu, J. Prins, J. Dinan, P. Sadayappan, and C.-W. Tseng, "UTS: an unbalanced tree search benchmark," in *Proceedings of the 19th international conference on Languages and compilers for parallel computing*, LCPC'06, (Berlin, Heidelberg), pp. 235–250, Springer-Verlag, 2007.

[13] N. Francez, “Distributed Termination,” *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 2, pp. 42 –55, jan. 1980.

[14] A. D. Birrell and B. J. Nelson, “Implementing remote procedure calls,” *SIGOPS Oper. Syst. Rev.*, vol. 17, pp. 3–, October 1983.

[15] J. Waldo, “Remote Procedure Calls and Java Remote Method Invocation,” *IEEE Concurrency*, vol. 6, pp. 5–7, July 1998.

[16] R. Srinivasan, “RPC: Remote Procedure Call Protocol Specification Version 2,” 1995.

[17] S. Saunders and L. Rauchwerger, “ARMI: an adaptive, platform independent communication library,” *SIGPLAN Not.*, vol. 38, pp. 230–241, June 2003.

[18] L. V. Kale and S. Krishnan, “CHARM++: a portable concurrent object oriented system based on C++,” in *Proceedings of the eighth annual conference on Object-oriented programming systems, languages, and applications*, OOPSLA ’93, (New York, NY, USA), pp. 91–108, ACM, 1993.

[19] S. Kumar, G. Dozsa, G. Almasi, P. Heidelberger, D. Chen, M. E. Giampapa, B. Michael, A. Faraj, J. Parker, J. Ratterman, B. Smith, and C. J. Archer, “The deep computing messaging framework: generalized scalable message passing on the blue gene/p supercomputer,” in *Proceedings of the 22nd annual international conference on Supercomputing*, ICS ’08, (New York, NY, USA), pp. 94–103, ACM, 2008.

[20] G. Shah and C. Bender, “Performance and Experience with LAPI – A New High-Performance Communication Library for the IBM RS/6000 SP,” in *Proceedings*

of the 12th. International Parallel Processing Symposium on International Parallel Processing Symposium, IPPS '98, (Washington, DC, USA), pp. 260–, IEEE Computer Society, 1998.

[21] V. Saraswat, G. Almasi, G. Bikshandi, C. Cascaval, D. Cunningham, D. Grove, S. Kodali, I. Peshansky, and O. Tardieu, "The Asynchronous Partitioned Global Address Space Model." `http://www.cs.rochester.edu/u/cding/amp/papers/full/The%20Asynchronous%20Partitioned%20Global%20Address%20Space%20Model.pdf`.

[22] J. J. Willcock, T. Hoefler, N. G. Edmonds, and A. Lumsdaine, "AM++: a generalized active message framework," in *Proceedings of the 19th international conference on Parallel architectures and compilation techniques*, PACT '10, (New York, NY, USA), pp. 401–410, ACM, 2010.

[23] D. A. Wallach, W. C. Hsieh, K. L. Johnson, M. F. Kaashoek, and W. E. Weihl, "Optimistic active messages: a mechanism for scheduling communication with computation," in *Proceedings of the fifth ACM SIGPLAN symposium on Principles and practice of parallel programming*, PPOPP '95, (New York, NY, USA), pp. 217–226, ACM, 1995.

[24] R. H. Halstead, Jr., "MULTILISP: a language for concurrent symbolic computation," *ACM Trans. Program. Lang. Syst.*, vol. 7, pp. 501–538, October 1985.

[25] M. Frigo, C. E. Leiserson, and K. H. Randall, "The implementation of the Cilk-5 multithreaded language," *SIGPLAN Not.*, vol. 33, pp. 212–223, May 1998.

[26] Randall, Keith H., *Cilk: efficient multithreaded computing.* PhD thesis, Massachusetts Institute of Technology, 1998.

[27] V. Cavé, J. Zhao, J. Shirako, and V. Sarkar, "Habanero-Java: the New Adventures of Old X10," in *PPPJ '11: Proceedings of 9th International Conference on the Principles and Practice of Programming in Java*, ACM, August 2011.

[28] Y. Guo, R. Barik, R. Raman, and V. Sarkar, "Work-first and help-first scheduling policies for async-finish task parallelism," in *Proceedings of the 2009 IEEE International Symposium on Parallel & Distributed Processing*, (Washington, DC, USA), pp. 1–12, IEEE Computer Society, 2009.

[29] N. Francez, "Distributed termination," *ACM Trans. Program. Lang. Syst.*, vol. 2, pp. 42–55, January 1980.

[30] E. W. Dijkstra, W. H. J. Feijen, and A. J. M. van Gasteren, "Derivation of a termination detection algorithm for distributed computations," *Inf. Process. Lett.*, vol. 16, no. 5, pp. 217–219, 1983.

[31] D. Kumar, "A class of termination detection algorithms for distributed computations," Tech. Rep. CS-TR-85-07, University of Texas at Austin, Austin, TX, USA, 1985.

[32] K. M. Chandy and J. Misra, *A paradigm for detecting quiescent properties in distributed computations*, pp. 325–341. New York, NY, USA: Springer-Verlag New York, Inc., 1985.

[33] V. A. Saraswat, P. Kambadur, S. Kodali, D. Grove, and S. Krishnamoorthy, "Lifeline-based global load balancing," in *Proceedings of the 16th ACM symposium on Principles and practice of parallel programming*, PPoPP '11, (New York, NY, USA), pp. 201–212, ACM, 2011.

[34] J. Dinan, S. Krishnamoorthy, L. Brian, J. Nieplocha, and P. Sadayappan, "Scioto: A Framework for Global-View Task Parallelism," in *Parallel Processing, 2008. ICPP '08. 37th International Conference on*, pp. 586 –593, sept. 2008.

[35] N. Francez and M. Rodeh, "Achieving distributed termination without freezing," *IEEE Transactions on Software Engineering*, vol. SE-8, pp. 287 – 292, May 1982.

[36] F. Mattern, "Algorithms for distributed termination detection," *Distributed Computing*, vol. 2, no. 3, pp. 161–175, 1987.

[37] J. Mellor-Crummey, L. Adhianto, and W. N. Scherer III, "A critique of co-array features in Fortran 2008." Fortran Standards Technical Committee Document J3/08-126, February 2008. http://www.j3-fortran.org/doc/meeting/183/08-126.pdf.

[38] W. N. Scherer, III, L. Adhianto, G. Jin, J. Mellor-Crummey, and C. Yang, "Hiding latency in Coarray Fortran 2.0," in *PGAS '10: Proceedings of the Third Conference on Partitioned Global Address Space Programing Models*, (New York, NY, USA), 2010.

[39] G. Jin, J. Mellor-Crummey, L. Adhianto, W. N. Scherer III, and C. Yang, "Implementation and Performance Evaluation of the HPC Challenge Benchmarks in Coarray Fortran 2.0," in *Proceedings of the 2011 IEEE International Parallel & Distributed Processing Symposium*, IPDPS '11, (Washington, DC, USA), pp. 1089–1100, IEEE Computer Society, 2011.

[40] G. Bikshandi, J. G. Castanos, S. B. Kodali, V. K. Nandivada, I. Peshansky, V. A. Saraswat, S. Sur, P. Varma, and T. Wen, "Efficient, portable implementation of

asynchronous multi-place programs," *SIGPLAN Not.*, vol. 44, no. 4, pp. 271–282, 2009.

[41] R. D. Blumofe and C. E. Leiserson, "Scheduling multithreaded computations by work stealing," *J. ACM*, vol. 46, pp. 720–748, September 1999.

[42] S. Agarwal, R. Barik, D. Bonachea, V. Sarkar, R. K. Shyamasundar, and K. Yelick, "Deadlock-free scheduling of X10 computations with bounded resources," in *Proceedings of the nineteenth annual ACM symposium on Parallel algorithms and architectures*, SPAA '07, (New York, NY, USA), pp. 229–240, ACM, 2007.

[43] D. D. Sleator and R. E. Tarjan, "Self-adjusting binary search trees," *J. ACM*, vol. 32, pp. 652–686, July 1985.

[44] L. Adhianto, S. Banerjee, M. Fagan, M. Krentel, G. Marin, J. Mellor-Crummey, and N. R. Tallent, "HPCToolkit: tools for performance analysis of optimized parallel programs," *Concurr. Comput. : Pract. Exper.*, vol. 22, no. 6, pp. 685–701, 2010.

[45] "HPC challenge benchmark." `http://icl.cs.utk.edu/hpcc`. Last accessed July 13, 2010.

[46] J. Dinan, D. B. Larkins, P. Sadayappan, S. Krishnamoorthy, and J. Nieplocha, "Scalable work stealing," in *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, SC '09, (New York, NY, USA), pp. 53:1–53:11, ACM, 2009.

[47] S.-J. Min, C. Iancu, and K. Yelick, "Hierarchical Work Stealing on Manycore

Clusters," in *PGAS '11: Proceedings of the Fourth Conference on Partitioned Global Address Space Programing Models*, (Galveston, TX, USA), 2011.