Experiences with Sweep3D Implementations in Co-Array Fortran *†

Cristian Coarfa Yuri Dotsenko John Mellor-Crummey

Dept. of Computer Science, Rice University, Houston TX 77005, USA

Abstract

As part of the recent focus on increasing the productivity of parallel application developers, Co-array Fortran (CAF) has emerged as an appealing alternative to the Message Passing Interface (MPI). CAF belongs to the family of global address space parallel programming languages; such languages provide the abstraction of globally addressable memory accessed using one-sided communication. At Rice University we are developing `cafc`, an open source, multiplatform CAF compiler. Our earlier studies show that `cafc`-compiled CAF programs achieve similar performance to that of corresponding MPI codes for the NAS Parallel Benchmarks. In this paper, we present a study of several CAF implementations of Sweep3D on four modern architectures. We analyze the impact of using one-sided communication in Sweep3D, identify potential sources of inefficiencies and suggest ways to address them. Our results show that we achieve comparable performance to that of the MPI version on three cluster-based architectures and outperform it by up to 10% on the SGI Altix 3000.

1 Introduction

Parallel computing is a vital technology for complex scientific simulations, however, achieving high performance with parallel programs is difficult. High performance parallel programs depend on the synergy of good programming models, effective optimizing compilers and capable hardware platforms. For a parallel programming model to be

*This work was supported in part by the Department of Energy under Grant DE-FC03-01ER25504/A000, the Los Alamos Computer Science Institute (LACSI) through LANL contract number 03891-99-23 as part of the prime contract (W-7405-ENG-36) between the DOE and the Regents of the University of California, Texas Advanced Technology Program under Grant 003604-0059-2001, and Compaq Computer Corporation under a cooperative research agreement. This research was performed in part using the Molecular Science Computing Facility (MSCF) in the William R. Wiley Environmental Molecular Sciences Laboratory, a national scientific user facility sponsored by the U.S. Department of Energy's Office of Biological and Environmental Research and located at the Pacific Northwest National Laboratory. Pacific Northwest is operated for the Department of Energy by Battelle. The computations were performed in part on an Itanium cluster purchased with support from the NSF under Grant EIA-0216467, Intel, and Hewlett Packard and on the National Science Foundation Terascale Computing System at the Pittsburgh Supercomputing Center.

†Cristian Coarfa and Yuri Dotsenko contributed equally to this work.

appealing, it has to meet three major goals: programs should be easy to write, it should be expressive and it should enable transparent performance portability. Ideally, a developer would write a parallel program once and then a parallel compiler would tailor the code to achieve high-performance on the parallel platform of choice.

Today, the *de facto* standard of parallel programming is the Message Passing Interface (MPI) [GSNL98]. It provides a standard for two-sided communication and is available on almost every parallel platform. Developers have found that it is difficult and error-prone to write parallel programs using the MPI model. Due to the explicit nature of MPI communication, MPI programs are not well-suited to compiler-based improvement, which leaves the end user solely responsible for choreographing the communication and computation to achieve high performance.

Recently, there has been a significant interest in trying to improve the productivity of parallel programmers by using language-based parallel programming models that abstract away most of the complex details of library-based high-performance communication. Experience with early HPF compilers has shown that in the absence of very capable parallelizing compilers, it is crucial to provide programmers with sufficient control to enable them to employ sophisticated parallelizations by hand. The family of global address space languages, including Co-array Fortran (CAF) [NR98a, NR98b] and Unified Parallel C (UPC) [CDC+99], has attracted interest as promising a near-term alternative to MPI. Both CAF and UPC employ the single-program-multiple-data (SPMD) model for parallel programming and are simple extensions to widely-used languages, Fortran 95 and C respectively. The global address space abstraction of these languages naturally supports a one-sided communication style. With communication and synchronization as part of the language, programs written in these languages are more amenable to compiler-directed communication optimization than MPI programs.

At Rice University we are developing `cafc`—a portable, open-source compiler for Co-array Fortran that performs source-to-source translation of CAF codes into Fortran 95 code augmented with calls to the ARMCI [NC99] communication library. Previous studies [CDEMC03, DCMC04, DCMCC04] show that even without sophisticated automatic communication optimization, our CAF compiler enables us to achieve performance comparable to that of hand optimized MPI applications for CAF versions of the well-known NAS benchmarks [BHS+95].

In this paper we describe our experiences developing Sweep3D [Acc95] implementations in CAF, compare their performance with that of LANL's original MPI version and analyze the performance differences between these codes on several platforms. Our CAF versions of Sweep3D use one-sided communication. While the CAF model eliminates the need for managing messaging in the program, it leaves users responsible for managing memory for communicated data. We first wrote a CAF version that closely follows the structure of the LANL's original MPI code. Because shared arrays are updated in place, this version is very synchronous. Next, we built a CAF version that adds a measure of asynchrony tolerance by using multi-version storage for communicated arrays; this enables communication to overlap

with computation. We built a third CAF version to help us understand the differences in performance between the two prior versions and MPI. To gain deeper insight into the performance differences, we implemented a microbenchmark which mimics the one-sided communication pattern encountered in Sweep3D.

Our results show that for Sweep3D, our best CAF version achieves scalability comparable to the MPI version on cluster architectures and outperforms it by up to 10% on the SGI Altix 3000, a hardware distributed shared-memory platform. For cluster-based architectures we identified three major sources of inefficiency. The first is overhead introduced by our CAF runtime library and `cafc`'s source-to-source translation. The second is that for Sweep3D's communication pattern, ARMCI delivers lower communication performance than MPI for small transfers. The third is that ARMCI's support for non-blocking communication should be refined to enable efficient combination with unidirectional notifications for common communication patterns, such as those found in Sweep3D. On SGI Altix 3000 the CAF versions of Sweep3D outperform the original MPI version by performing direct data movement without any intermediate copies; the MPI version performs more data movement, copying data to and from communication buffers.

In the next section, we give a brief overview of the CAF programming model and communication libraries for one- and two-sided communication. In section 3 we describe key implementation decisions for our CAF compiler. We describe the blocking communication microbenchmarks and the CAF and MPI versions of Sweep3D in section 4. In section 5 we present and analyze our performance results for Sweep3D. Section 6 summarizes our conclusions.

2 Background

Here, we briefly describe the CAF programming model, along with extensions we developed to facilitate portable high-performance. We then describe one-sided and two-sided communication libraries and their implications for parallel application development.

2.1 Co-Array Fortran and Extensions

Co-array Fortran supports SPMD parallel programming through a small set of language extensions to Fortran 95. An executing CAF program consists of a static collection of asynchronous process images. Similar to MPI, CAF programs explicitly manage data locality and computation distribution. However, CAF belongs to the family of Global Address Space programming languages and provides the abstraction of globally accessible memory both for cluster-based and for shared memory architectures.

CAF supports distributed data using a natural extension to Fortran 95 syntax. For example, the declaration

`integer :: a(n,m)[*]` declares a shared co-array `a` with `n` × `m` integers local to each process image. The dimensions inside brackets are called co-dimensions. Co-arrays may also be declared for user-defined types as well as primitive types. A local section of a co-array may be a singleton instance of a type rather than an array of type instances.

Instead of explicitly coding message exchanges to access data belonging to other processes, a CAF program can directly reference non-local values using an extension to Fortran 95 syntax for subscripted references. For instance, process `p` can read the first column of co-array `a` from process `p+1` with the right-hand side reference to `a(:,1)[p+1]`.

CAF has several synchronization primitives. `sync_all` implements a synchronous barrier; `sync_team`, is used for barrier-style synchronization among dynamically-formed *teams* of two or more processes; and `sync_memory` implements a local memory fence and ensures the consistency of the process image memory by completion of all outstanding communication requests issued by this image.

Since both remote data access and synchronization are language primitives in CAF, communication and synchronization are amenable to compiler-based optimization. In contrast, communication in MPI programs is expressed in a more detailed form, which makes it much harder to transform with a compiler. CAF also contains several features that improve the expressiveness and power of the language including dynamic allocation of co-arrays, co-arrays of user-defined types containing pointers, and fledgling support for parallel I/O. A more complete description of the CAF language can be found in [NR98b]

Our previous studies [CDEMC03, DCMC04] identified a few weaknesses of the original CAF language specification that reduce the performance of CAF codes and proposed extensions to CAF to avoid these sources of performance degradation. First, the original CAF specification requires programs to have implicit memory fences before and after each procedure call to ensure that the state of memory is consistent before and after each procedure invocation. This guarantees that each array accessed within a subroutine is in consistent state upon entry and exit from the subroutine. In many cases, an invoked procedure does not access co-array data at all or accesses only co-array data that does not overlap with co-array data accessed by the caller. As a consequence, it is not possible to overlap communication with a procedure's computation with memory fences around the procedure's call sites. Second, CAF's original team-based synchronization required using collective synchronization even in cases when it is not necessary or may complicate the coding, e.g., when using multipartitioning [Van93] data distribution. In [CDEMC03], we propose augmenting CAF with unidirectional, point-to-point synchronization primitives: `sync_notify` and `sync_wait`. `sync_notify(q)` sends a notify to process image `q`; this notification is guaranteed to be seen by image `q` only after all communication events previously issued by the notifier to image `q` have been completed. `sync_wait(p)` blocks its caller until it receives a matching notification message from the process image `p`. Communication events for

CAF remote data accesses are blocking. While it is possible to exploit non-blocking communication in some cases, automatically replacing blocking communication with its non-blocking counterpart and overlapping communication with computation requires sophisticated compiler analysis. To enable savvy application developers to overlap communication and computation in cases where compiler analysis cannot do so automatically, it is useful for CAF to provide a user-level mechanism for exploiting non-blocking communication. To address that, we have proposed a small set of primitives that enable application developers to delay the completion of communication events [DCMC04].

2.2 Communication Libraries

MPI uses a two-sided (send and receive) communication model to communicate data between processes. With two-sided communication, both the sender and receiver explicitly participate in a communication event. As a consequence, both sender and receiver temporarily set aside their computation to communicate data. Having two processes complete a point-to-point communication explicitly synchronizes the sender and receiver.

CAF uses one-sided communication (`PUT` and `GET`) to access remote data. When using one-sided communication, one side specifies both source and destination of communicated data. From the programmer's perspective, the other image is not aware of the communication. Thus, the one-sided model cleanly separates data movement from synchronization; this can be particularly useful for simplifying the coding of irregular applications. On loosely-coupled architectures, a one-sided communication library can take advantage of Remote Direct Memory Access (RDMA) capabilities of modern networks, such as Myrinet [ANS98] and Quadrics [PcFH+02]. During an RDMA data transfer, the Network Interface Chip (NIC) controls the data movement without interrupting the remote host Central Processing Unit (CPU). This enables the CPU to compute while communication is in progress. On all microprocessor-based architectures, a cache coherency protocol is used to maintain consistency between CPU caches and memory that is the source or sink of communication. On shared memory platforms such as Altix 3000, one-sided communication is performed by the CPU using load/store instructions on globally addressable shared memory. The hardware uses directory-based cache coherence to provide fast data movement and to maintain consistency between CPU caches and (local or remote) shared memory. As our recent study [DCMCC04] demonstrated, on shared-memory architectures fine-grain one-sided communication is fastest with compiler generated load/store instructions, while large contiguous transfers are faster when transmitted using a `memcpy` library function optimized for the target platform.

The `cafc` compiler uses the Aggregate Remote Memory Copy Interface (ARMCI) [NC99]—a multi-platform library for high-performance one-sided communication—as its implementation substrate for global address space communication. ARMCI provides both blocking and split-phase non-blocking primitives for one-sided data movement as well as primitives for efficient unidirectional synchronization. On some platforms, using split-phase primitives

enables communication to be overlapped with computation. ARMCI provides an excellent implementation substrate for global address space languages making use of coarse-grain communication because it achieves high performance on a variety of networks (including Myrinet, Quadrics, and IBM's switch fabric for its SP systems) as well as shared memory platforms (Cray X1, SGI Altix3000, SGI Origin2000) while insulating its clients from platform-specific implementation issues such as shared memory, threads, and DMA engines. A notable feature of ARMCI is its support for non-contiguous data transfers [NTSP02].

3 The `cafc` Compiler for Co-Array Fortran

We designed the `cafc` compiler for Co-array Fortran with the major goals of being portable and delivering high-performance on a multitude of platforms. Ideally, a programmer writes a CAF program once in a natural style and `cafc` would adapt it for high performance on the target platform of choice.

To achieve this goal, `cafc` performs source-to-source transformation of CAF code into Fortran 95 code augmented with communication operations. By employing source-to-source translation, `cafc` aims to leverage the best back-end compiler available on the target platform to optimize local computation. For communication, `cafc` typically generates calls to ARMCI's one-sided communication primitives; however, for shared memory systems it also can generate code that uses load and store operations for communication. `cafc` is based on OPEN64/SL [Ope02], a version of the OPEN64 [Ope01] compiler infrastructure that we modified to support source-to-source transformation of Fortran 95 and CAF.

To support efficient access to remote co-array data on the broadest range of platforms, memory for co-arrays must be managed by the communication substrate; typically, this memory is managed separately from memory managed conventionally by a Fortran 95 compiler's runtime system. Currently, co-array memory is allocated and managed by the ARMCI library. On cluster systems with RDMA capabilities, co-arrays are allocated in memory that is registered and pinned, which enables data transfers to be performed directly using the DMA engine of the NIC.

For CAF programs to perform well, access to local co-array data must be efficient. Since co-arrays are not supported in Fortran 95, we need to translate references to the local portion of a co-array into valid Fortran 95 syntax. For performance, our generated code must be amenable to back-end compiler optimization. In an earlier study [DCMCC04], we explore several alternative representations for co-arrays. Our current strategy is to use a Fortran 95 pointer to access local co-array data. Since co-array data is allocated outside the Fortran 95 runtime system, we need the ability to initialize and manipulate compiler-dependent Fortran 95 array descriptors (dope vectors) on a variety of target platforms. We use the CHASM library [RSB03] from Los Alamos National Laboratory for this

purpose.

Communication events expressed with CAF's bracket notation must be converted into Fortran 95; however, this is not straightforward because the remote memory may be in a different address space. Although the CAF language provides shared-memory semantics, the target architecture may not; a CAF compiler must perform transformations to bridge this gap. On a hardware shared memory platform, the transformation is relatively straightforward since references to remote memory in CAF can be expressed as loads and stores to shared locations; the study [DCMCC04] contains a detailed exploration of the alternatives for performing communication on hardware shared memory systems. The situation is more complicated for cluster-based systems with distributed memory.

To perform data movement on clusters, `cafc` must generate calls to a communication library to access data on a remote node. Moreover, `cafc` must manage storage to temporarily hold remote data needed for a computation. For example, in the case of a read reference of a co-array on another image, `arr(:)=coarr(:)[p] + ...`, a temporary, `temp`, is allocated just prior to the statement to hold the value of the `coarr(:)` array section from image p. Then, a call to get data from image p is issued to the runtime library. The statement is rewritten as `arr(:) = temp(:) + ...`. The temporary is deallocated immediately after the statement. For a write to a remote image, such as `coarr(:)[p1,p2]=...`, a temporary `temp` is allocated prior to the remote write statement; the result of the evaluation of the right-hand side is stored in the temporary; a call to a communication library is issued to perform the write; and finally, the temporary is deallocated. When possible, the generated code avoids using unneeded temporary buffers. For example, for an assignment performing a co-array to co-array copy, `cafc` generates code to move the data directly from the source into the destination. In general, `cafc` generates blocking communication operations. However, user directives [DCMC04] enable `cafc` to exploit non-blocking communication.

To support point-to-point synchronization in CAF (`sync_notify` and `sync_wait`), we collaborated with the developers of ARMCI on the design of suitable `armci_notify` and `armci_wait` primitives. ARMCI ensures that if a blocking or non-blocking `PUT` to a remote process image is followed by a notify to the same process image, then the destination image receives the notification after the `PUT` operation has completed. While ARMCI supports non-blocking communication, on some architectures, the implementation of `armci_notify` itself is blocking. This limits the overlap of communication and computation if a CAF programmer writes a non-blocking write to a remote co-array and notifies the destination process image immediately thereafter. To maximize the overlap of communication and computation, `sync_notify` should have a non-blocking implementation as well. We are actively collaborating with the ARMCI developers regarding non-blocking notifications.

`cafc` is available as open-source. It supports COMMON, SAVE, and ALLOCATABLE co-arrays of primitive and user-defined types, passing of co-array arguments, co-arrays with multiple co-dimensions, co-array communica-

tion using array sections, the CAF synchronization primitives and most of the CAF intrinsic functions. The following features of CAF are currently not supported: allocatable co-array components, triplets in co-dimensions, and parallel I/O. Ongoing work is aimed at removing these limitations. `cafc` compiles natively and runs on the following architectures: Pentium clusters with Ethernet interconnect, Itanium2 clusters with Myrinet or Quadrics interconnect, Alpha clusters with Quadrics interconnect, SGI Origin 2000 and SGI Altix 3000.

Previous studies [CDEMC03, DCMC04, DCMCC04] have shown that even without automatic communication optimizations we are able to obtain performance and scalability comparable to that of hand-coded MPI applications for the NAS benchmarks on a range of cluster and hardware shared-memory systems.

4 Comparing One-sided and Two-sided Communication

The goal of our study is to experiment and compare one-sided and two-sided communication schemes for a code with a sophisticated parallelization. For this purpose, we selected the ASCI Sweep3D [Acc95] application.

4.1 Sweep3D

Sweep3D solves a one-group time-independent discrete ordinates (Sn) 3D Cartesian (XYZ) geometry neutron transport problem. The XYZ geometry is represented by an IJK logically rectangular grid of cells. The angular dependence is handled by discrete angles with a spherical harmonics treatment for the scattering source. The solution involves two steps: the streaming operator is solved by sweeps for each angle and the scattering operator is solved iteratively.

Sweep3D exploits wavefront parallelism. It uses a 2D spatial domain decomposition onto a 2D processor array in the I- and J-directions. For efficient parallelization, Sweep3D is coded to pipeline blocks of MK k-planes and MMI angles through this 2D processor array. Thus, the wavefront exploits parallelism in both I- and J-directions simultaneously. A more complete description of Sweep3D can be found elsewhere [Acc95]. Figure 1 shows a piece of pseudocode representing a high-level view of the Sweep3D kernel.

To investigate the impact of different CAF coding styles, we implemented three CAF versions of the Sweep3D: Sweep3D-CAF, Sweep3D-CAF-pa, and Sweep3D-CAF-mb. We compared their performance to that of the MPI version. The difference among the CAF versions is in the communication implementation, while the local computation is similar.

Sweep3D-CAF was developed from the original MPI code by declaring its `Phiib` and `Phijb` arrays as co-arrays and using blocking `PUT` to communicate them "in-place". For the I-direction communication, the code is presented in Figure 2. For the J-direction communication, the code is similar except that the process image communicates the

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

       recv e/w into Phiib !  recv block I-inflows
       recv n/s into Phijb !  recv block J-inflows

       ...
       !  intense computation with use/update Phiib and Phijb
       ...

       send e/w Phiib !  send block I-outflows
       send n/s Phijb !  send block J-outflows

     enddo
   enddo
enddo
```

Figure 1: Sweep3D kernel pseudocode.

`Phijb` array with its J-predecessor and with its J-successor.

```
...
if (receiving from I-predecesor) then
   !  notify the I-pred that the local Phiib buffer is ready to accept new data
   call sync_notify(I-pred)
   !  wait for the new data to arrive from the I-predecessor
   call sync_wait(I-pred)
endif

...
!  intense computation with use/update Phiib and Phijb
...

if (sending to I-successor) then
   !  wait for the I-succ notification that its Phiib is ready to accept new data
   call sync_wait(I-succ)
   !  transfer the data to the I-successor (contiguous blocking PUT)
   Phiib(:,:,:)[I-succ] = Phiib(:,:,:)
   !  notify the I-succ that the new data has been sent
   call sync_notify(I-succ)
endif
...
```

Figure 2: Sweep3D-CAF kernel pseudocode.

The Sweep3D-CAF communication is very similar to that of the MPI version. The data movement statement — assignment to `Phiib` — communicates the same data as the `send/recv` pair of the MPI version. The `sync_notify` and `sync_wait` provide synchronization analogous to that induced by an MPI `send/recv` pair. For Sweep3D-CAF, there is no data copy from `Phiib` or `Phijb` into an auxiliary communication buffer; the data is communicated directly "in-place". In contrast, the MPI version might use extra data copies to/from a communication buffer to move the data.

Sweep3D-CAF-pa is similar to the Sweep3D-CAF version but `Phiib` and `Phijb` arrays are not communicated "in-place". First, they are copied into auxiliary co-arrays on the source; then, the auxiliary co-arrays are communicated using a blocking `PUT` to the destination; finally, at the destination, the communicated data is copied from the auxiliary co-arrays into the `Phiib` and `Phijb`. The synchronization remains the same. In essence, this version simulates

possible data copies that the MPI version might execute.

Sweep3D-CAF-mb aims to overlap PUTs with computation on the successor and to provide asynchrony tolerance. It uses additional storage, namely, three instances of Phiib and Phijb to overlap communication with computation. In the wavefront steady state, one instance of Phiib can be used to receive the data from the I-predecessor; at the same time another instance can be used to perform the local computation, while the third instance can be used to communicate the data to the I-successor (three-buffer scheme). For shared-memory architectures without hardware support for asynchronous data transfers, a two-buffer scheme, in which one buffer is used for local computation and the other is used for a PUT performed by a predecessor, is likely to yield the best performance. Phiib has an extra high-order dimension to manage the instances in the circular-buffer fashion to avoid unnecessary copies. Similarly, three instances of Phijb can be used to provide for communication and computation overlap for the wavefront parallelism in the J-direction. The simplified pseudocode for three-buffer scheme is given in Figure 3. Note that more buffers can be used. Our implementation is general and supports an arbitrary number of buffers holding incoming data from the predecessor and outgoing data to the successor. On the platforms where non-blocking PUTs and notifications are supported, the code uses non-blocking communication directives proposed in [DCMC04] to overlap communication with local computation, otherwise only blocking PUTs are used.

```
...
advance the phiib_wrk_idx index (index rotation to avoid extra memory copies)
if (receiving from I-predecesor) then
   !  wait for the data from the I-predecessor
   call sync_wait(I-pred)
   !  notify the I-predecessor that we have a buffer available to receive new data
   call sync_notify(I-pred)
endif

...
!  heavy computation with use/update Phiib and Phijb
...

if (sending to I-successor) then
   finalize locally the previous non-blocking PUT to the I-succ

   start the region of non-blocking communication with index phiib_wrk_idx
   !  transmit the new data to the I-succ using non-blocking contiguous PUT (due to
decomposition symmetry, phiib_wrk_idx has the same value locally and on the I-succ)
   Phiib(:,:,:,phiib_wrk_idx)[I-succ] = Phiib(:,:,:,phiib_wrk_idx)
   !  notify the I-successor that more data is available
   call sync_notify(I-succ)
   stop the region of non-blocking communication with index phiib_wrk_idx
endif
...
```

Figure 3: Sweep3D-CAF-mb kernel pseudocode.

4.2 Blocking Communication Throughput Microbenchmark

To explain some of the Sweep3D performance results we also created a microbenchmark that measures blocking communication throughput.

Data movement is an important component of parallel applications and often a key factor in application performance. To evaluate the capabilities of CAF programs for blocking communication we have devised a producer-consumer microbenchmark and evaluated several versions: MPI, CAF and ARMCI. Such a benchmark enables us to evaluate the performance of the communication library used for blocking communication on a particular target platform.

The core of the MPI version is presented in figures 4 (a) and (b). In the microbechmark process `img1`, the sender (figure 4(a)), communicates `SIZE` double precision numbers to process `img2`, the receiver (figure 4(b)). The MPI blocking communication calls, `MPI_send` and `MPI_recv`, provide both data movement and synchronization between the sender and the receiver. The communication event is performed `NRUNS` times; to overcome limitations in the system clock precision, the value we chose for `NRUNS` was 500000. We measure throughput by dividing the total size of the data sent to the execution time and is measured in MB/second:

$throughput = \frac{SIZE*NRUNS*8*10e-6}{execution\_time}$

To compose a CAF version with the same semantics, we explicitly specify data movement and synchronization separately. The destination process image, `img2`, must notify the source process image, `img1`, that the co-array data to be written is no longer being used by computation and is, thus, available to be overwritten. After the `PUT`, the source process image uses `sync_notify` to signal the destination that the data has arrived. Issuing `PUT`s without the synchronization is, in general, not realistic and would result in race conditions. The code for the source and the destination processes is presented in figures 4 (c) and (d).

Finally, we have written a version of the blocking `PUT` microbenchmark using ARMCI calls directly, to evaluate the overhead introduced by the CAF runtime layer for communication events. The code for the ARMCI version is similar to the CAF code and is shown in figures 4 (e) and (f).

5 Experiments

We evaluated the performance of our CAF and MPI variants of Sweep3D on four platforms.

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

The second platform used was a cluster of 2000 HP Long's Peak dual-CPU workstations at the Pacific Northwest National Laboratory. The nodes are connected with Quadrics QSNet II (Elan 4). Each node contains two 1.5GHz Itanium2 processors with 32KB/256KB/6MB L1/L2/L3 cache and 4GB of RAM. The operating system is Red Hat

```
double precision a(N1)
.....
do i=1, 500000
  call mpi_send(a1(1), size,
   MPI_DOUBLE_PRECISION, img2,
   99, MPI_COMM_WORLD, ierr)
end do
```
(a) Source, MPI

```
double precision a(N1)
.....
do i=1, 500000
  call mpi_recv(a1(1), size,
   MPI_DOUBLE_PRECISION, img1,
   99, MPI_COMM_WORLD, ierr)
end do
```
(b) Destination, MPI

```
double precision a(N1)[0:*]
.....
do i=1, 500000
  call sync_wait(img2)
  a(1:SIZE)[img2]=a(1:SIZE)
  call sync_notify(img2)
end do
```
(c) Source, CAF

```
double precision a(N1)[0:*]
.....
do i=1, 500000
  call sync_notify(img1)
  call sync_wait(img1)
end do
```
(d) Destination, CAF

```
double precision addressA(NUM_IMAGES)
.....
do i=1, 500000
  call armci_wait(img2)
  call ARMCI_put(addressA[img1],
   addressA[img2],
   SIZE, img2 )
  call armci_notify(img2)
end do
```
(e) Source, ARMCI

```
double precision addressA(NUM_IMAGES)
.....
do i=1, 500000
  call armci_notify(img1)
  call armci_wait(img1)
end do
```
(f) Destination, ARMCI

Figure 4: MPI, CAF and ARMCI versions of the blocking `PUT` microbenchmark.

Linux (kernel version 2.4.20). The back-end compiler is the Intel Fortran compiler version 8.0.

The third platform we used for experiments was a cluster of 92 HP zx6000 workstations interconnected with Myrinet 2000. Each workstation node contains two 900MHz Intel Itanium 2 processors with 32KB/256KB/1.5MB of L1/L2/L3 cache, 4-8GB of RAM, and the HP zx1 chipset. Each node is running the Linux operating system (kernel version 2.4.18-e plus patches). We used the Intel Fortran compiler version 8.0 for Itanium as our Fortran 90 back-end compiler.

The fourth platform is an SGI Altix 3000, with 128 Itanium2 1.5GHz processors with 6MB L3 cache, and 128 GB RAM, running the Linux64 OS with the 2.4.21 kernel and the Intel Fortran compiler version 8.0.

For the Sweep3D benchmark, we compare the parallel efficiency of the MPI and CAF versions. We compute parallel efficiency as follows. For each parallelization ρ, the efficiency metric is computed as $\frac{t_s}{P \times t_p(P,\rho)}$. In this equation, t_s is the execution time of the sequential version; P is the number of processors; $t_p(P, \rho)$ is the time for the parallel execution on P processors using parallelization ρ. Using this metric, perfect speedup would yield efficiency of 1.0 for each processor configuration. We use efficiency rather than speedup or execution time as our comparison

```
double precision a1(N1)[0:*]
double precision a2(N1)[0:*]
.....
do i=1, 500000/2
  call sync_wait(img2)
  a1(1:SIZE)[img2]=a1(1:SIZE)
  call sync_notify(img2)
  call sync_wait(img2)
  a2(1:SIZE)[img2]=a2(1:SIZE)
  call sync_notify(img2)
end do
```

(a) Source, CAF

```
double precision a1(N1)[0:*]
double precision a2(N1)[0:*]
.....
call sync_notify(img1)
call sync_notify(img1)
do i=1, 500000/2
  call sync_wait(img1)
  call sync_notify(img1)
  call sync_wait(img1)
  call sync_notify(img1)
end do
call sync_wait(img1)
call sync_wait(img1)
```

(b) Destination, CAF

Figure 5: CAF version of the two-buffer `PUT` microbenchmarks.

metric because it enables us to accurately gauge the relative performance of multiple benchmark implementations across the *entire* range of processor counts. We present results for sizes 50x50x50, 150x150x150 and 300x300x300, with the total memory requirements of 16MB, 434MB and 3463MB respectively .

For the blocking `PUT` microbenchmarks we present the throughput measured in MB/second (10^6 B/second), for message sizes ranging from 512B to 128KB, which covers the message sizes encountered in the Sweep3D experiments.

The results for the Alpha cluster with Quadrics interconnect are shown in figures 6, 7 and 8. The results for the Itanium2 cluster connected with Quadrics are presented in figures 9, 10 and 11. Figures 12, 13 and 14 displays the results for the Itanium2 cluster with Myrinet 2000 interconnect. Finally, the results on the SGI Altix 3000 machine are shown in figures 15, 16 and 17. The result for the microbenchmark on the various architecture are presented in figures 18, 19, 20 and 21.

Our results show that for Sweep3D we achieve a scalability comparable to that of the MPI version on the cluster architectures and outperform it by up to 10% on the hardware shared-memory machine.

On the Alpha cluster, the Sweep3D-CAF-mb version slightly outperforms the MPI version for the 50x50x50 problem size, while MPI outperforms Sweep3D-CAF-mb for the 150x150x150 problem size, and they perform comparably for the 300x300x300 problem size. The Sweep3D-CAF-mb enables better latency tolerance; by using multiple communication buffers, it reduces the wait time of the source process image for a buffer to become available for a `PUT` to the destination process image. Currently, the `notify` is implemented in ARMCI with a blocking `PUT`. While we can overlap the `PUT` to the successor with the `PUT` from the predecessor (both performed as independent RDMA by the NIC, as described in section 2.2), we cannot overlap the `PUT` with computation on the source process image. The blocking communication microbenchmark shows that the CAF translation doesn't introduce a significant overhead

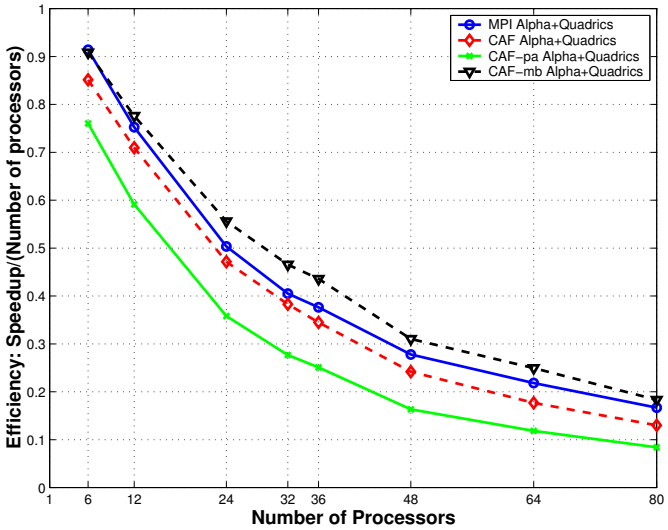

Figure 6: Results for Sweep3D size 50x50x50 on an Alpha cluster with a Quadrics Elan3 interconnect.

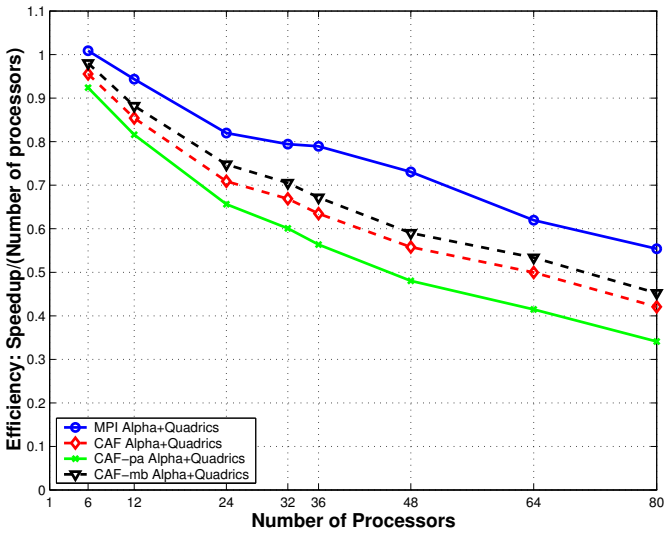

Figure 7: Results for Sweep3D size 150x150x150 on an Alpha cluster with a Quadrics Elan3 interconnect.

over the ARMCI library, but the ARMCI library itself yields a lower throughput than the native MPI implementation. As expected, the one-buffer versions of Sweep3D — Sweep3D-CAF and Sweep3D-CAF-pa — perform worse than the multiple-buffer version Sweep3D-CAF-mb. Also, Sweep3D-CAF-pa performs expectedly worse than Sweep3D-CAF because of extra data copies.

On the Itanium2 cluster with Quadrics Elan4 interconnect, MPI slightly outperforms Sweep3D-CAF-pa and Sweep3D-CAF-mb versions for the problem sizes 50x50x50, and achieves comparable performance for the 150x150x150 and 300x300x300 problem size. A surprising finding was that the Sweep3D-CAF version performs 20-30% worse

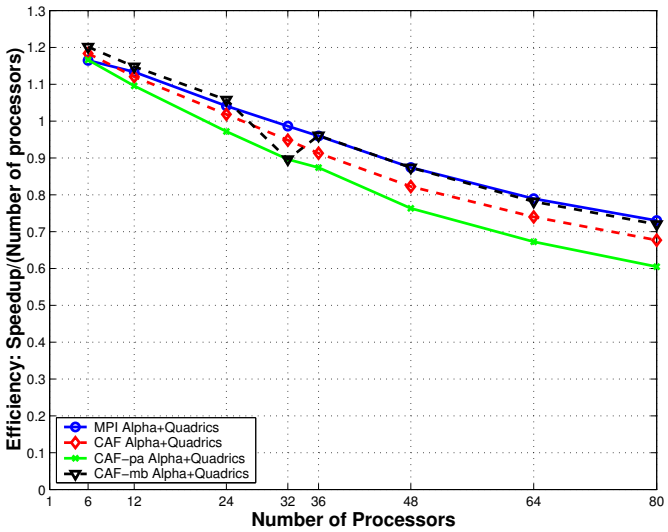

Figure 8: Results for Sweep3D size 300x300x300 on an Alpha cluster with a Quadrics Elan3 interconnect.

compared to the other versions. Our investigation lead to the conclusion that the performance difference is due to inefficient code generated by the Intel Fortran Compiler. We discovered that the Sweep3D-CAF has 33% more L1 and L2 instruction cache misses than its MPI counterpart, while experiencing comparable number of data cache misses. Because Sweep3D employs wave-front parallelism, both local code and communication efficiency are very important to achieving overall good performance. Our initial suspicion was that the performance difference is due to the `Phiib` and `Phijb` buffers being invalidated in the CPU cache because of the RDMA data transfers. These arrays are used

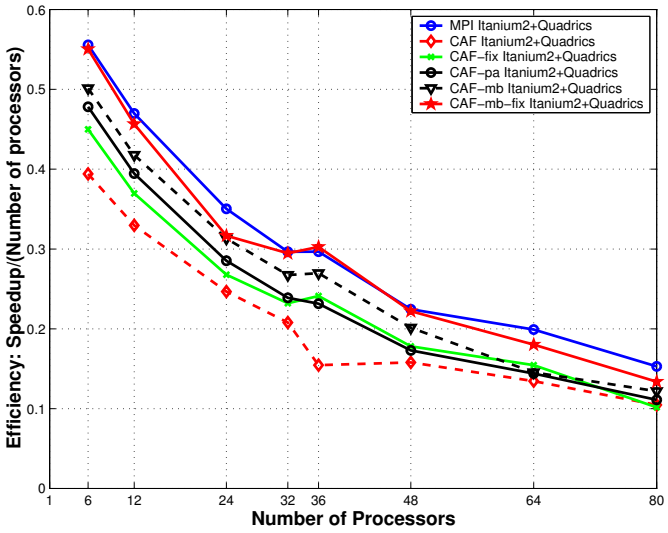

Figure 9: Results for Sweep3D size 50x50x50 on an Itanium2 cluster with a Quadrics Elan4 interconnect.

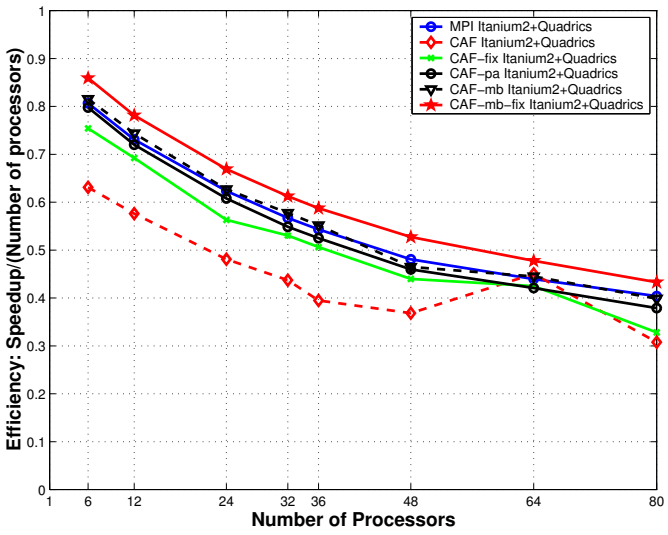

Figure 10: Results for Sweep3D size 150x150x150 on an Itanium2 cluster with a Quadrics Elan4 interconnect.

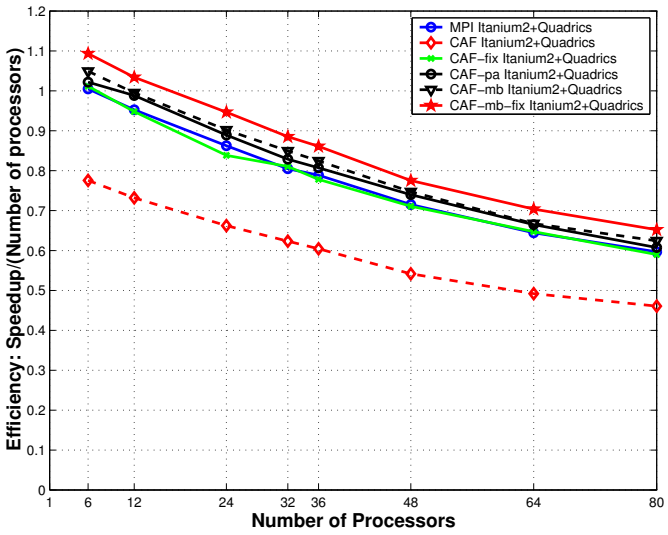

Figure 11: Results for Sweep3D size 300x300x300 on an Itanium2 cluster with a Quadrics Elan4 interconnect.

in the local computation extensively, so we inserted loop nests that access the `Phiib` and `Phijb` in the code right after the data is communicated, thus warming up the cache. That change fixed the performance problem. However, moving the loop nest in a separate procedure and passing `Phiib` and `Phijb` as parameters again lead to performance degradation. Inserting loop nests that access a small temporary array, not used in the computation, again improved the performance. At this point, we measured the instruction and data cache misses and saw that the performance degradation was caused by the extra instruction cache misses. This explains counterintuitive results for the Sweep3D-CAF-pa version that performs extra data copying while achieving better performance than the Sweep3D-CAF version. We plan

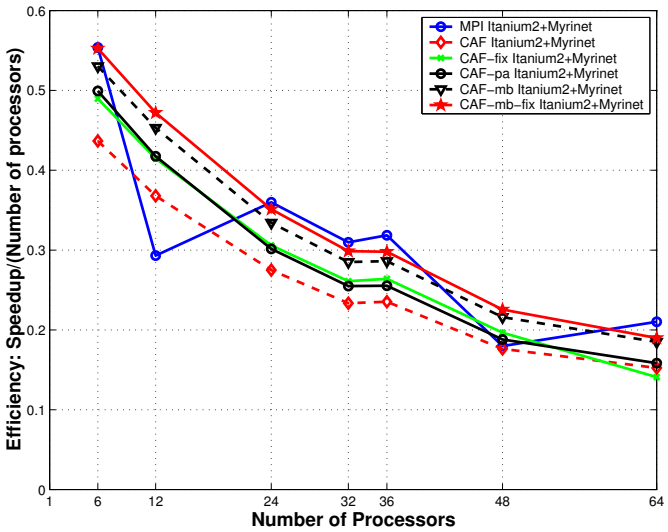

Figure 12: Results for Sweep3D size 50x50x50 on an Itanium2 cluster with a Myrinet 2000 interconnect.

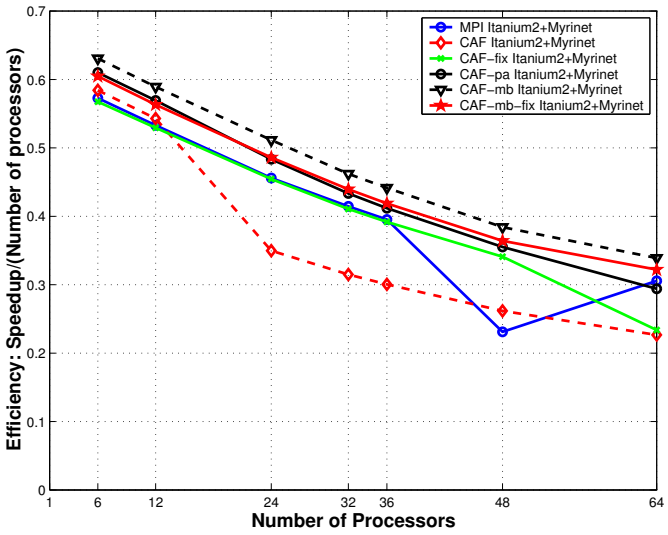

Figure 13: Results for Sweep3D size 150x150x150 on an Itanium2 cluster with a Myrinet 2000 interconnect.

to contact Intel to investigate this unintuitive behavior of code generated by their Fortran compiler. The Sweep3D-CAF-fix version is derived from Sweep3D-CAF by adding the "fix-up" code; similarly, the Sweep3D-CAF-mb-fix is derived from the Sweep3D-CAF-mb version. The Sweep3D-CAF-mb-fix version achieves performance comparable to that of the MPI version for the 50x50x50 problem size, and outperforms it by up 9% to for the 150x150x150 and 300x300x300 problem size. By analyzing the microbenchmark results, we have discovered that even though the throughput of the ARMCI version is close to that of the MPI version, the code generated by the CAF compiler introduces an overhead over the ARMCI version. Also, the multiple-buffer version of the microbenchmark gains over the

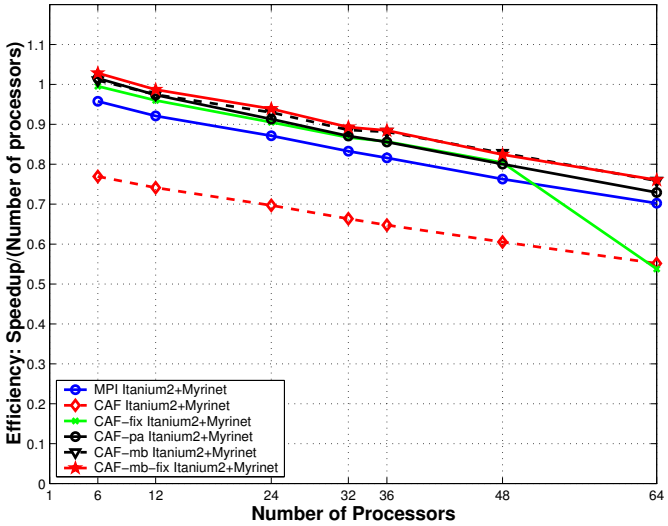

Figure 14: Results for Sweep3D size 300x300x300 on an Itanium2 cluster with a Myrinet 2000 interconnect.

one-buffer version due to better asynchrony tolerance.

On the Itanium2 cluster with Myrinet 2000 interconnect, the MPI version performs comparably to the Sweep3D-CAF-mb versions for the 50x50x50 problem size. The Sweep3D-CAF-mb version exceeds MPI performance by up to 12% for the 150x150x150 and by up to 9% for 300x300x300 problem size. These excellent results are due to a high degree of communication and computation overlap possible because the ARMCI library implements full support for non-blocking `PUT`s and `notify` on the Myrinet 2000 interconnect. The microbenchmark results show that the

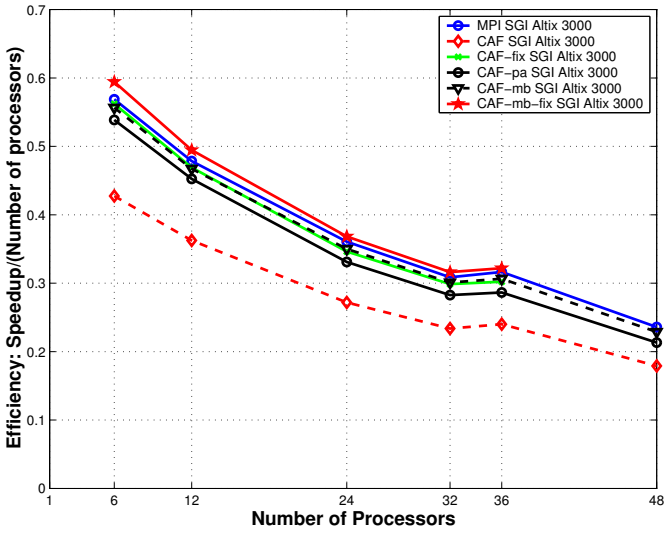

Figure 15: Results for Sweep3D size 50x50x50 on an SGI Altix 3000.

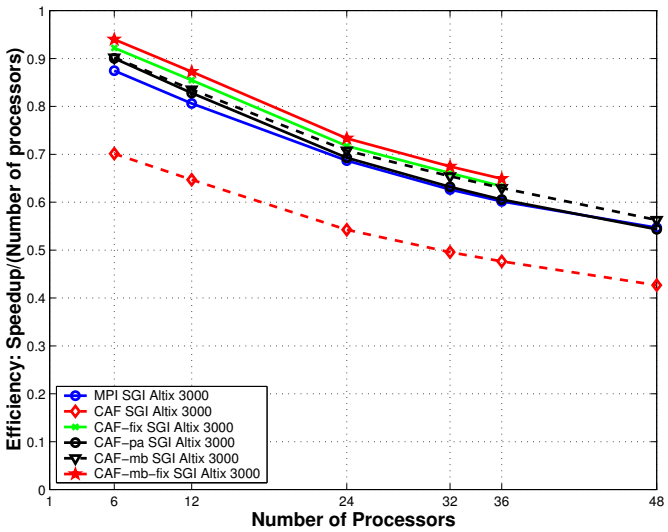

Figure 16: Results for Sweep3D size 150x150x150 on an SGI Altix 3000.

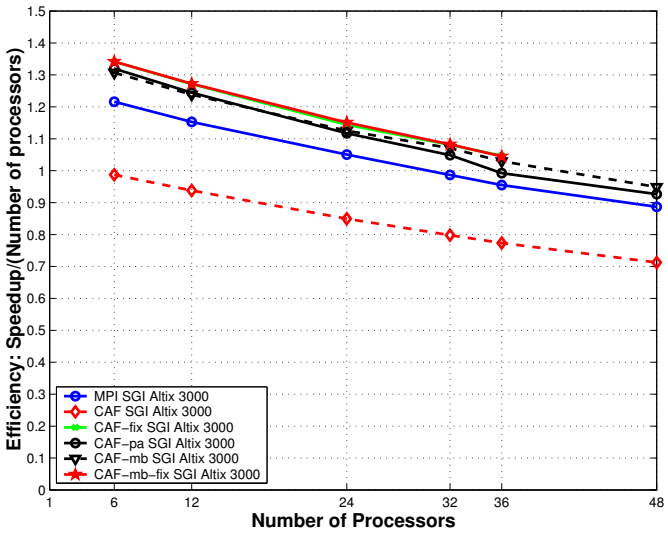

Figure 17: Results for Sweep3D size 300x300x300 on an SGI Altix 3000.

multiple buffer version with non-blocking communication outperforms the other CAF versions because it allows good asynchrony tolerance and pipelining of communication events. We noticed that MPI achieves superior performance for small message sizes (up to 16KB); it is our understanding that the Myrinet-based implementation of MPI buffers small message sizes on the sender, thus, in reality, sending larger messages limited by MPI's internal buffer size.

The Sweep3D-CAF version requires `notifys` to be sent to the predecessors to indicate that the `Phiib` and `Phijb` buffers are available for incoming data. To further reduce the number of `notifys`, we created a CAF version that expanded the `Phiib` and `Phijb` buffers so that a separate buffer is used per each pair of plane and angle. This

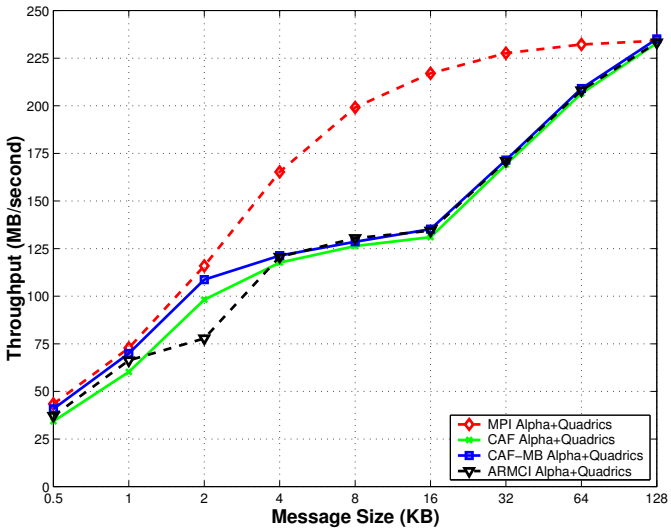

Figure 18: Results for blocking put microbenchmark on an Alpha cluster with a Quadrics Elan3 interconnect.

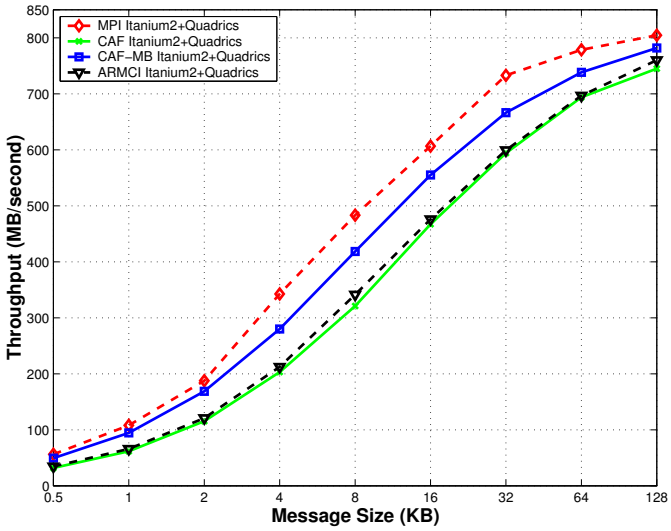

Figure 19: Results for the blocking put microbenchmarks on an Itanium2 cluster with a Quadrics Elan4 interconnect.

enabled us to remove all the `notifys` to the predecessors. However, experiments performed on the Itanium2 cluster with the Myrinet 2000 interconnect showed that the extra buffer memory led to 5-38% more L3 cache misses, resulting in 5-13% runtime increase compared to the Sweep3D-CAF-mb version.

To summarize our findings for cluster-based architectures, there are three major sources of inefficiencies for CAF codes compared to the equivalent MPI codes. First, source-to-source translation and CAF runtime library add more instructions to the program causing some, usually minor, inefficiency. Second, there is a communication performance difference between ARMCI and native MPI implementations: for the blocking communication patterns commonly

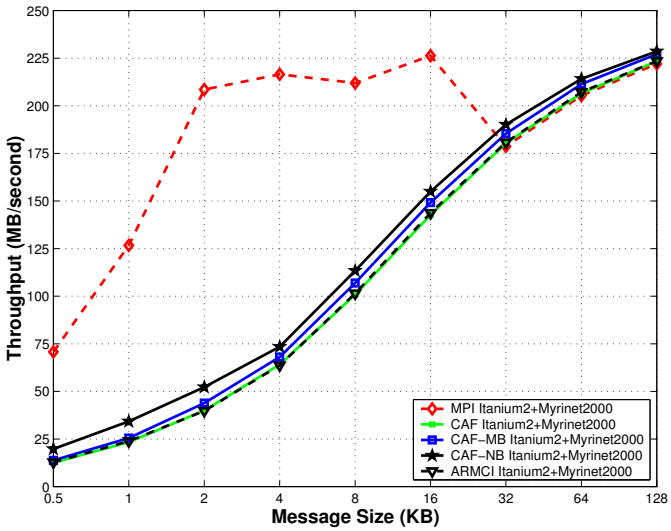

Figure 20: Results for the blocking put microbenchmarks on an Itanium2 cluster with a Myrinet 2000 interconnect.

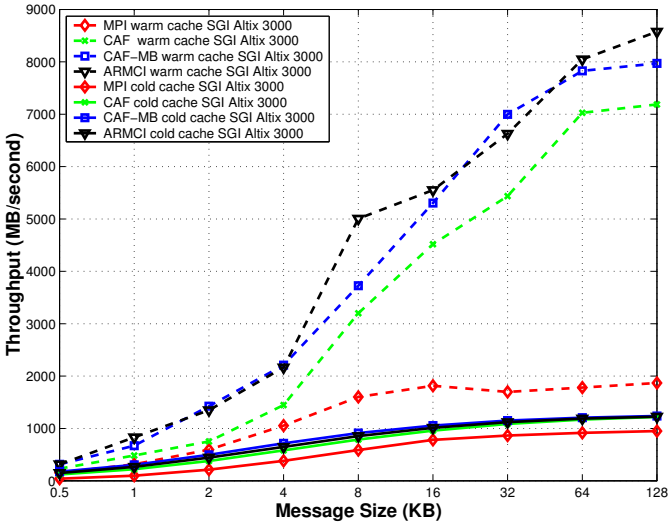

Figure 21: Results for the blocking put microbenchmarks on an SGI Altix 3000.

used in Sweep3D, the ARMCI version of our microbenchmark performs slightly worse than the MPI version. Third, ARMCI does not have full support for non-blocking `PUT`s and `notify`, implemented via a blocking `PUT`, for the Quadrics interconnects, precluding potential overlap of computation and communication. We are working closely with the ARMCI developers to provide a non-blocking implementation of `notify`.

On the SGI Altix 3000 machine, the Sweep3D-CAF-mb and Sweep3D-CAF-pa versions perform slightly worse than the Sweep3D version for 50x50x50 problem size, slightly outperform the Sweep3D version for 150x150x150 problem size, and consistently outperform the Sweep3D version for 300x300x300 problem size by up to 10%. On

this architecture, due to lack of hardware support for efficient non-blocking communication, ARMCI implements non-blocking communication by using memory copy subroutines, which is equivalent to blocking communication. For this reason, the execution times of Sweep3D-CAF-pa and Sweep3D-CAF-mb are almost indistinguishable. Although ARMCI performs direct data movement, MPI may perform one or more extra copies. The Sweep3D-CAF-mb-fix version demonstrates even higher performance due to more efficient local computation. The SGI recommended configuration of the MPI library did not improve the performance of Sweep3D.

The SGI Altix 3000 provides hardware cache consistency with cache line granularity. For this reason, we measured two versions of the microbenchmark: one that repeatedly sends data from the same memory location on the source to the same memory location on the destination (thus, keeping the transmitted data in cache), and the other that at every step transmits data from a different memory address, so that the data needs to be brought in cache for every transmission. The first version is denoted "warm cache" in figure 21, while the second is denoted "cold cache". For the warm cache version, the results show that while the throughput for the CAF and ARMCI versions scales with the message size, achieving values as high as 7000–8000 MB/second, the throughput of the MPI version is limited to 600MB/s. For the cold cache version, the CAF versions outperform the MPI version by a factor of two for messages smaller than 4KB and by 30% for messages larger than 4KB. For the Sweep3D application, we expect most of transmitted data to be in cache prior to the transmission, so the "warm cache" version is a closer approximation of the communication behavior of Sweep3D, which is one of the reasons the CAF versions outperform the MPI version on the SGI Altix 3000.

6 Conclusions

In the quest to increase the productivity of parallel application developers, programming models based on one-sided communication have emerged as appealing alternatives to MPI. To better understand the implications of such models on program performance, we developed and studied several CAF implementations of Sweep3D. For each program variant, we describe its implementation strategy and analyze its performance on three cluster architectures and a hardware shared memory machine. Our results show that the CAF versions achieve performance comparable to that of LANL's original MPI version on the cluster-based architectures and outperform the MPI version by up to 10% on the SGI Altix 3000. For the cluster-based architectures, we identified three major sources inefficiency in our compiler-generated code for CAF; we plan to address each in the nearest future. On the SGI Altix 3000 architecture, sophisticated CAF versions outperform the MPI version of Sweep3D because they are able to effectively exploit hardware support for direct data transfers.

In our experience, CAF's one-sided communication model is easier to use than MPI for writing simple programs. However, at present, developing carefully-tuned parallel codes in CAF seems as difficult as it is in MPI. While MPI manages message buffering transparently, in CAF programs multi-version storage for communicated arrays and associated synchronization must currently be managed at source code level. To increase programmer's productivity, we are planning to explore compiler and runtime support for transparently managing multi-version storage of communicated arrays; this should improve asynchrony tolerance of CAF programs written in a natural one-sided communication style by enabling better overlap of communication and computation.

7 Acknowledgements

We thank J. Nieplocha and V. Tipparaju for collaborating on the refinement and tuning of ARMCI. We thank F. Zhao for her work on the Open64/SL Fortran front-end. We thank D. Chavarría-Miranda for his insightful discussions on Sweep3D.

References

[Acc95] Accelerated Strategic Computing Initiative. The ASCI Sweep3D Benchmark Code. `http://www.llnl.gov/asci_benchmarks/asci/limited/sweep3d/asci_sweep3d.h%tml`, 1995.

[ANS98] ANSI. *Myrinet-on-VME Protocol Specification (ANSI/VITA 26-1998).* American National Standard Institute, 1998.

[BHS+95] D. Bailey, Tim Harris, William Saphir, Rob van der Wijngaart, Alex Woo, and Maurice Yarrow. The NAS parallel benchmarks 2.0. Technical Report NAS-95-020, NASA Ames Research Center, December 1995.

[CDC+99] W. W. Carlson, J. M. Draper, D. E. Culler, K. Yelick, and K. Warren E. Brooks. Introduction to UPC and language specification. Technical Report CCS-TR-99-157, IDA Center for Computing Sciences, May 1999.

[CDEMC03] Cristian Coarfa, Yuri Dotsenko, Jason Eckhardt, and John Mellor-Crummey. Co-Array Fortran Performance and Potential: An NPB Experimental Study. In *Proc. of the 16th Intl. Workshop on Languages and Compilers for Parallel Computing*, number 2958 in LNCS. Springer-Verlag, October 2-4, 2003.

[DCMC04] Yuri Dotsenko, Cristian Coarfa, and John Mellor-Crummey. A Multiplatform Co-Array Fortran Compiler. In *Proceedings of the 13th Intl. Conference of Parallel Architectures and Compilation Techniques*, Antibes Juan-les-Pins, France, September 29 - October 3 2004.

[DCMCC04] Yuri Dotsenko, Cristian Coarfa, John Mellor-Crummey, and Daniel Chavarría-Miranda. Experiences with Co-Array Fortran on Hardware Shared Memory Platforms. In *Proceedings of the 17th International Workshop on Languages and Compilers for Parallel Computing*, September 2004.

[GSNL98] William Gropp, Marc Snir, Bill Nitzberg, and Ewing Lusk. *MPI: The Complete Reference*. MIT Press, second edition, 1998.

[NC99] J. Nieplocha and B. Carpenter. *ARMCI: A Portable Remote Memory Copy Library for Distributed Array Libraries and Compiler Run-Time Systems*, volume 1586 of *Lecture Notes in Computer Science*, pages 533–546. Springer-Verlag, 1999.

[NR98a] R. W. Numrich and J. K. Reid. Co-Array Fortran for parallel programming. Technical Report RAL-TR-1998-060, Rutheford Appleton Laboratory, August 1998.

[NR98b] R. W. Numrich and J. K. Reid. Co-Array Fortran for parallel programming. *ACM Fortran Forum*, 17(2):1–31, August 1998.

[NTSP02] J. Nieplocha, V. Tipparaju, A. Saify, and D.K. Panda. Protocols and strategies for optimizing performance of remote memory operations on clusters. In *Proc. Workshop Communication Architecture for Clusters (CAC02) of IPDPS'02*, Ft. Lauderdale, Florida, April 2002.

[Ope01] Open64 Developers. Open64 compiler and tools. `http://sourceforge.net/projects/open64`, September 2001.

[Ope02] Open64/SL Developers. Open64/SL compiler and tools. `http://hipersoft.cs.rice.edu/open64`, July 2002.

[PcFH+02] Fabrizio Petrini, Wu chun Feng, Adolfy Hoisie, Salvador Coll, and Eitan Frachtenberg. The Quadrics network: high performance clustering technology. *IEEE Micro*, 22(1):46–57, January-February 2002.

[RSB03] Craig Rasmussen, Matt Sottile, and Tom Bulatewicz. CHASM language interoperability tools. `http://sourceforge.net/projects/chasm-interop`, July 2003.

[Van93] R. F. Van der Wijngaart. Efficient implementation of a 3-dimensional adi method on the ipsc/860. In *Proceedings of the 1993 ACM/IEEE conference on Supercomputing*, pages 102–111. ACM Press, 1993.