Co-Array Fortran Performance and Potential: An NPB Experimental Study[⋆]

Cristian Coarfa, Yuri Dotsenko, Jason Eckhardt, and John Mellor-Crummey

Rice University, Houston TX 77005, USA

Abstract. Co-array Fortran (CAF) is an emerging model for scalable, global address space parallel programming that consists of a small set of extensions to the Fortran 90 programming language. Compared to MPI, the widely-used message-passing programming model, CAF's global address space programming model simplifies the development of single-program-multiple-data parallel programs by shifting the burden for choreographing and optimizing communication from developers to compilers. This paper describes an open-source, portable, and retargetable CAF compiler under development at Rice University that is well-suited for today's high-performance clusters. Our compiler translates CAF into Fortran 90 plus calls to one-sided communication primitives. Preliminary experiments comparing CAF and MPI versions of several of the NAS parallel benchmarks on an Itanium 2 cluster with a Myrinet 2000 interconnect show that our CAF compiler delivers performance that is roughly equal to or, in many cases, better than that of programs parallelized using MPI, even though support for global optimization of communication has not yet been implemented in our compiler.

1 Introduction

Parallel languages and parallelizing compilers have been a long term focus of compiler research. To date, this research has not had the widespread impact on the development of parallel scientific applications that had been hoped. The two standard parallel programming models suited to scientific computation that have received industrial backing are OpenMP [1] and High Performance Fortran (HPF) [2]. However, both of these models have significant shortcomings that reduce their utility for writing portable, scalable, high-performance parallel programs. OpenMP programmers have little control over data layout; as a result, OpenMP programs are difficult to map efficiently to distributed memory platforms. In contrast, HPF enables programmers to explicitly control the mapping of data to processors; however, to date, commercial HPF compilers have failed to deliver high-performance for a broad range of programs. As a result, the Message Passing Interface (MPI) [3] has become the *de facto* standard for parallel programming

[⋆] 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. The Itanium cluster used in this work was purchased with support from the NSF under Grant EIA-0216467, Intel, and Hewlett Packard.

because it enables application developers to write portable, scalable, high-performance parallel programs using very sophisticated parallelizations under programmer's control.

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 high-performance communication (e.g. asynchronous calls), yet provide programmers with sufficient control to enable them to employ sophisticated parallelizations. Two languages in particular have been the focus of recent attention as promising near-term alternatives to MPI: Co-array Fortran (CAF) [4, 5] and Unified Parallel C (UPC) [6]. Both CAF and UPC support a global address space model for single-program-multiple-data (SPMD) parallel programming. Communication in these languages is simpler than MPI: one simply reads and writes shared variables. With communication and synchronization as part of the language, these languages are more amenable to compiler-directed communication optimization than MPI programs.

To date, CAF has not appealed to application scientists as a model for developing scalable, portable codes, because the language is still somewhat immature and a fledgling compiler is only available on Cray platforms [7]. At Rice University, we are working to create an open-source, portable, retargetable, high-quality CAF compiler suitable for use with production codes. Our compiler translates CAF into Fortran 90 plus calls to ARMCI [8], a multi-platform library for one-sided communication. Recently, we completed implementation of the core CAF language features, enabling us to begin experimentation to assess the potential of CAF as a high-performance programming model. Preliminary experiments comparing CAF and MPI versions of the BT, MG, SP and CG NAS parallel benchmarks [9] on a large Itanium 2 cluster with a Myrinet 2000 interconnect, show that our CAF compiler prototype already yields code with performance that is roughly equal to hand-tuned MPI.

In the next section, we briefly describe the CAF language and the ARMCI library that serves as the communication substrate for our generated code. Section 3 proposes extensions to CAF to enable it to deliver portable high performance. In Section 4, we outline the implementation strategy of our source-to-source CAF compiler. Section 5 presents our recommendations for writing high-performance CAF programs. In Section 6, we describe experiments using versions of the NAS parallel benchmarks to compare the performance of CAF and MPI. Section 7 presents our conclusions and outlines our plans for future work.

2 Background

Co-Array Fortran (CAF) supports SPMD parallel programming through a small set of language extensions to Fortran 90. Like MPI programs, an executing CAF program consists of a static collection of asynchronous process images. CAF programs explicitly manage data locality and computation distribution; however, CAF is a global address space programming model. CAF supports distributed data using a natural extension to Fortran 90 syntax. For example, the declaration `integer :: x(n,m)[*]` declares a shared co-array with $n \times m$ 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 obtain data belonging to other processes, a CAF program can directly reference non-local values using an extension to Fortran 90 syntax for subscripted references. For instance, process `p` can read the first column of data in co-array `x` from process `p+1` with the right-hand side reference to `x(:,1)[p+1]`. The CAF language includes several synchronization primitives; the most important of them are `sync_all`, which implements a synchronous barrier, `sync_team`, which is used for barrier-style synchronization among dynamically-formed *teams* of two or more processes, and `start_critical/end_critical` primitives for controlling entry to a single global critical section. Since both remote data access and synchronization are language primitives in CAF, communication and synchronization are amenable to compiler-based optimizing transformations. In contrast, communication in MPI programs is expressed in a more detailed form, which makes it more difficult to improve 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 elsewhere [5].

2.1 ARMCI

The CAF compiler we describe in this paper uses the Aggregate Remote Memory Copy Interface (ARMCI) [8]—a multi-platform library for high-performance one-sided (get and put) communication—as its implementation substrate for global address space communication. One-sided communication separates data movement from synchronization; this can be particularly useful for simplifying the coding of irregular applications. ARMCI provides both blocking and split-phase non-blocking primitives for one-sided communication. 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) while insulating its clients from platform-specific implementation details such as shared memory, threads, and DMA engines. A notable feature of ARMCI is its support for non-contiguous data transfers [10].

3 Towards Portable High-Performance CAF

The CAF programming model is still emerging. Prior to our compiler, the only existing CAF compiler implementation was for the Cray T3E and X1 platforms—tightly-coupled shared memory architectures with high-performance interconnects that support efficient fine-grain communication and global synchronization. The original CAF language specification [4, 5] was influenced by these architectural features, leading to CAF codes that would not perform well on less tightly-coupled architectures. Evaluating the performance of CAF codes written according to the original language specification on a

Myrinet cluster helped us to identify several features of the specification that reduce the potential performance of CAF codes. Below we discuss some of these features along with approaches we propose to address the problems they cause.

Memory fence semantics associated with CAF procedure calls. The `sync_memory` intrinsic is a memory fence that ensures the consistency of a process image's local memory by waiting for the completion of all of that process's outstanding communication events. To ensure a consistent state for co-array data accessed during or after a procedure call, the original CAF model requires implicit memory fences before and after every procedure invocation. We found this requirement to be overly restrictive since it prevents overlapping communication with a procedure call, which is often an important strategy for hiding communication latency. It should be possible for a sophisticated programmer to relax this requirement where it is unnecessary for correctness. We are in the process of exploring design alternatives that will make this possible.

Overly restrictive synchronization primitives. An issue that arose during our application evaluation was that using synchronization primitives in the original CAF language specification reduced the performance of applications we studied. For example, the original CAF specification only supports collective synchronization (`sync_all` and `sync_team`); however, many applications require only unidirectional, point-to-point synchronization. Using collective synchronization where only point-to-point synchronization is needed degrades performance and in some cases makes programming harder. We propose `sync_notify(q)` and `sync_wait(p)` as two new intrinsics for point-to-point synchronization. When a process executes a `sync_notify`, it initiates notification of the specified process image and then can continue immediately. When a process executes a `sync_wait`, it must block until it is notified by the specified process image. When a notification from process p is delivered to process q, all pending communication events (both puts and gets) that p issued to q before p initiated the `sync_notify` have completed.

Collective operations. The CAF language specification does not provide collective communication intrinsics. CAF is expressive enough so that users can write collective communication routines in CAF; however this is likely to result in programs tailored to a particular architecture (and in many cases to a range of processor counts too) that are unlikely to deliver high performance on architectures with different communication latency and bandwidth characteristics. CAF should be extended to include collective communication intrinsics to give a CAF compiler flexibility to choose an appropriate algorithm and implementation suited to the target architecture at hand. We are in the process of designing a set of CAF intrinsics for collective communication.

4 Compiler Implementation Strategy

We have implemented the core features of CAF, enabling us to express non-trivial CAF programs. Section 6 gives a description of some programs we have compiled and evaluated. Our compiler performs source-to-source transformation of CAF codes into F90 plus calls to a communication library (currently ARMCI). This strategy was designed to leverage the best back-end compiler available on the target platform to optimize local computation. Our CAF compiler is implemented on top of OPEN64/SL [11], a version of the OPEN64 compiler infrastructure [12] that we have modified to support source-to-source transformation of Fortran 90. Below we outline some of the principal compiler design issues that arose when implementing CAF.

Memory management issues. Current operating systems do not usually allow for sharing of arbitrary memory allocated independently by different processes. For this reason, memory for co-arrays must be managed by the communication substrate separately from memory managed conventionally by an F90 compiler's language runtime system. Having the communication library allocate co-array memory enables our generated code to use the most efficient communication strategy for a particular platform. For example, on an SMP machine the memory can be allocated in shared memory which would enable communication to be performed using processor load and store instructions. On a Myrinet-based cluster, allocating data for a communication event in pinned physical memory enables the library to perform data transfers on the memory directly using the Myrinet adapter's DMA engine.

For CAF programs to perform well, access to the local portions of co-arrays must be efficient. Since co-arrays are not supported in F90, we need to translate references to the local portion of a co-array into a valid F90 construct and this construct must be amenable to back-end compiler optimization. We believe that the best strategy is to use an F90 pointer to access local co-array data. However, the difficulty with this strategy is that we want to allocate co-array data outside F90-managed memory. To use an F90 pointer to access co-array data, we must initialize the pointer's dope vector outside an F90 compiler's language runtime system. This requires compiler-dependent code for initializing F90 pointers, which poses a minor difficulty when retargeting.

Co-array sequence association and reshaping. CAF explicitly provides sequence association between local parts of co-arrays in common blocks, but equivalence of co-array and non-co-array memory is prohibited. To support sequence association, our compiler allocates storage once for each common block at program launch and then sets up a procedure-level view for each common block containing co-arrays. Our CAF compiler implements this using a two-part strategy. First, at compile time, it generates a set of static initializers, which set up each procedure's view of a common block containing co-arrays. Next, at link time, a global initialization routine is generated by collecting the static initializers. This routine allocates memory once for each common block and invokes the static initializer to create each procedure's view in turn.

CAF allows programmers to pass co-arrays as arguments to procedures. For each formal co-array parameter passed by reference, our implementation augments the subroutine prototype with a "hidden" parameter; each hidden parameter is a pointer to a runtime data structure describing the co-array to the callee. At each call site, every co-array actual parameter is replaced by an F90 pointer to its local co-array data and a pointer to the run-time data structure describing the co-array.

Co-array communication generation. Communication events expressed with CAF's bracket notation must be converted into F90; however, this is not straightforward because the remote memory is in a different address space. Although the language provides shared-memory semantics, the target architecture may not. A CAF compiler must provide transformations to bridge this semantic 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 situation is more complicated for cluster-based systems with distributed memory. To perform the data movement, the compiler must generate calls to a communication library since the data resides on a remote node. Moreover, storage must be managed to temporarily hold off-processor data to perform a computation.

Naive translation may lead to situations where excessive storage is used and superfluous copying is performed. Eventually, our compiler will automatically detect such situations and eliminate the extraneous storage and copying, when possible. Compare two statements where remote memory is updated:

```
a(:)[p] = b(:)
a(:)[p] = b(:) + 1
```

In the first case, a separate communication buffer may not be necessary since the data to be sent to processor `p` is already available in `b`. On the other hand, the second statement calls for local computation; the result should be computed into a temporary communication buffer and then transferred to processor `p`. Now consider the case when remote data is used:

```
a(:) = b(:)[p] + 1
a(:) = b(:)[p] + c(:)[q]
```

In the first statement, no extra communication buffer is necessary because we can use `a` for temporarily storing `b(:)[p]` (`a` becomes dead) to evaluate the expression. But for the second case, one extra communication buffer is required because we need to transfer two vectors of off-processor data to evaluate the expression.

Key missing features. There are a number of language features that are not yet implemented in our preliminary compiler. The most important of these are allocatable co-arrays, co-arrays of user-defined types (including those with pointer components), triplet notation in co-dimensions, and multiple co-dimensions.

5 Writing High Performance Co-array Fortran Code

Once we completed support for core CAF language features in our prototype CAF compiler, we undertook a study of several of the NAS parallel benchmarks to understand the interplay of CAF language, compiler, and runtime issues and their impact on the programmability, scalability, performance and performance portability of applications. From our colleagues Bob Numrich at University of Minnesota and Allan Wallcraft at Naval Research Lab, we received draft CAF versions of the MG, CG, SP, and BT NAS parallel benchmarks that they created from the MPI codes in the NPB version 2.3 release. Analyzing variants of these codes gave us a better understanding of how to develop high performance programs in CAF.

All of the CAF code transformations we describe in this section represent manual source-level tuning we applied to CAF sources for the NAS benchmarks to best exploit CAF language features for performance. It is our goal to enhance the capabilities of our prototype CAF compiler to apply such transformations automatically. Our aim is to generate high-performance code that meets or exceeds the performance of hand-coded MPI parallelizations from easy to write CAF source programs. We are in the process of adding program analysis to our compiler to support automating such transformations.

In our study, we found that there are several key coding strategies for writing high performance CAF code. We list them in the decreasing order of importance:

Communication aggregation and vectorization. This is a critical optimization for architectures in which the communication fabric does not support low-latency, fine-grain memory transactions. Analysis of the NAS benchmark loops revealed that all major communication could be vectorized manually using triplet notation for subscripts of co-array references. Once support for data flow and dependence analysis are in place in our CAF compiler, in most cases it should be straightforward to automate this transformation. Consider Figure 1(a) which is a simple code fragment from the *conj_grad* routine of a first-draft CAF parallelization of NAS CG that we received from our colleagues. For this code, our prototype CAF compiler generates a `get` for every iteration, which is expensive.

```
do k=n1,n2
  q(k) = w(j)[reduce_exch_proc(i)]
  j = j + 1
enddo
```

(a) unvectorized

```
q(n1:n2) =
  w(j:j+n2-n1)[reduce_exch_proc(i)]
```

(b) vectorized

Fig. 1. NAS CG before and after communication vectorization.

Using the triplet notation as in Figure 1(b) enables our present CAF compiler prototype to generate a single ARMCI communication event for such a statement, which is substantially faster than the original. We observed performance improvements up to two orders of magnitude by applying this transformation, even for relatively small problem sizes of the NAS benchmarks we studied. In CAF, when the shapes of source and destination array sections are conformant, vectorized communication can be expressed using triplet notation. Otherwise, a buffer copy is necessary at the source or destination to yield conformant shapes or to pack the data on the sender and unpack the data on the receiver. The latter approach mimics the message packing and unpacking in MPI.

Synchronization strength reduction. Analogous to the well-known operator strength reduction transformation, synchronization strength reduction involves transforming a strong synchronization primitive, e.g., a barrier, into a weaker one(s), e.g., point-to-point notify/wait, while preserving the meaning of the program, with the aim of improving performance. Others have previously employed similar optimizations with significant benefits [13]. This optimization was a key performance boost for each of the NAS benchmarks we studied. Figure 2 uses a fragment from NAS MG, a 3D multigrid solver, to illustrate this transformation. Figure 2(a) shows the original CAF version of the code in which processors perform a barrier synchronization before and after exchanging boundary layers of its 3D block with its pair of neighbors along a coordinate dimension. This code was originally written for the Cray T3E, which has fast hardware support for barriers. However, a barrier provides much stronger synchronization than necessary; only synchronization with the adjacent neighbors is needed. On cluster interconnects that do not have fast hardware support for barriers it is more efficient to use point-to-point synchronization. Figure 2(b) shows the code recast to use our new CAF one-way point-to-point synchronization primitives. On a Myrinet 2000 cluster, this transformation improved performance by about 30% for 64 processors.

```
if( .not. dead(kk) )then
 do  axis = 1, 3
  if( nprocs .ne. 1) then
   call sync_all()

   call give3(axis,+1,u,n1,n2,n3,kk)
   call give3(axis,-1,u,n1,n2,n3,kk)

   call sync_all()

   call take3(axis,-1,u,n1,n2,n3)
   call take3(axis,+1,u,n1,n2,n3)
  else
   call comm1p(axis,u,n1,n2,n3,kk)
  endif
 enddo
else ...
```

(a) using barrier synchronization

```
if( .not. dead(kk) )then
 do  axis = 1, 3
  if( nprocs .ne. 1) then
   call sync_notify(nbr(axis,1,kk)+1)
   call sync_notify(nbr(axis,-1,kk)+1)
   call sync_wait(nbr(axis,1,kk)+1)
   call sync_wait(nbr(axis,-1,kk)+1)
   call give3(axis,+1,u,n1,n2,n3,kk)
   call sync_notify(nbr(axis,1,kk)+1)
   call give3(axis,-1,u,n1,n2,n3,kk)
   call sync_notify(nbr(axis,-1,kk)+1)
   call sync_wait(nbr(axis,1,kk)+1)
   call sync_wait(nbr(axis,-1,kk)+1)
   call take3( axis,-1,u,n1,n2,n3)
   call take3( axis,+1,u,n1,n2,n3)
  else
   call comm1p(axis,u,n1,n2,n3,kk)
  endif
 enddo
else ...
```

(b) using point-to-point synchronization

Fig. 2. Communication in NAS MG before and after synchronization strength reduction.

Conversion of Gets into Puts. On communication fabrics such as Myrinet, `put` operations are supported directly, whereas `get` operations require asking a server-side thread to supply the requested data with a `put`. For such an interconnect, when using regular algorithms, it is feasible and potentially profitable to transform each `get` operation into a `put`.

6 Experiments and Discussion

In this section we compare the performance of the code our compiler generates from CAF with hand-coded MPI implementations of the MG, CG, BT and SP NAS parallel benchmark codes. For our study, we used MPI versions from the NPB 2.3 release. Sequential performance measurements used as a baseline were performed using the NPB 2.3-serial release. The NPB codes are widely regarded as useful for evaluating the performance of compilers on parallel systems.

For each benchmark, we compare the parallel efficiency of MPI and CAF implementations of each benchmark. 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 original sequential version implemented by the NAS group at the NASA Ames Research Laboratory; 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 1.0 for each processor configuration. We use efficiency rather than speedup or execution time as our comparison metric because it enables us to accurately gauge the relative performance of multiple benchmark implementations across the *entire* range of processor counts.

All experiments were performed on 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. Our operating environment is the GNU/Linux operating system (kernel version 2.4.20 plus patches). Although this Linux kernel is SMP-capable, we used only one of the processors on each SMP node for our experiments (1) to avoid contention for the Myrinet and local memory, and (2) to avoid process ping-ponging since our kernel was not configured to support affinity scheduling. We used the Intel Fortran v7.0 for Itanium (efc) as the back-end compiler for all F90 code generated by the CAF translator as well as for the MPI versions of the benchmarks. Optimization level 3 was used along with the *override-limits* option to prevent the compiler from automatically disabling certain expensive optimizations. CAF executables were linked against ARMCI 1.1-beta for Myrinet GM. All executables were linked against Myricom's MPI implementation MPICH-GM 1.2.5..10 (compiled with Intel's efc) running on Myricom's GM 1.6.4 driver substrate.

In the following sections, we briefly describe the NAS benchmarks used in our evaluation, the key features of their MPI and CAF parallelizations and compare the performance of the CAF and MPI implementations.

6.1 NAS CG

In the NAS CG parallel benchmark, a conjugate gradient method is used to compute an approximation to the smallest eigenvalue of a large, sparse, symmetric positive definite matrix [9]. This kernel is typical of unstructured grid computations in that it tests irregular long distance communication and employs sparse matrix vector multiplication. The irregular communication requirement of this benchmark is evidently a challenge for all systems.

On each iteration of loops involving communication the MPI version initiates a non-blocking receive from `reduce_exch_proc(i)` processor followed by an MPI send to the same processor. After the send, the process waits until its MPI receive completes. Thus, no overlap of communication and computation is possible.

```
! notify our partner that we are here and wait for
! him to notify us that the data we need is ready
  call sync_notify(reduce_exch_proc(i)+1)
  call sync_wait(reduce_exch_proc(i)+1)
! get data from our partner
  q(n1:n2) = w(m1:m1+n2-n1)[reduce_exch_proc(i)]
! synchronize again with our partner to
! indicate that we have completed our exchange
! so that we can safely modify our part of w
  call sync_notify(reduce_exch_proc(i)+1)
  call sync_wait(reduce_exch_proc(i)+1)
! local computation
  ... use q, modify w ...
```

Fig. 3. A typical fragment of optimized CAF for NAS CG.

Our tuned CAF version of NAS CG does not differ much from the MPI hand-coded version. In fact, we directly converted two-sided MPI communication into equivalent calls to notify/wait and a vectorized one-sided get communication event. Figure 3 shows a typical fragment of our CAF parallelization using notify/wait synchronization. Our experiments showed that for this code, replacing the co-array read (`get`) operation with a co-array write (`put`) had a negligible effect on performance because of the amount of synchronization necessary to preserve data dependences.

In initial experimentation with our CAF version of CG on various numbers of processors, we found that on less than eight processors, performance was significantly lower than its MPI counterpart. In our first CAF implementation of CG, the receive array q was a common block variable, allocated in the static data by the compiler and linker. To perform the communication shown in Figure 3 our CAF compiler prototype allocated a temporary buffer in memory registered with ARMCI so that the Myrinet hardware could initiate a DMA transfer. After the `get` was performed, data was copied from the temporary buffer into the q array. For runs on a small number of processors, the buffers are large. Moreover, the registered memory pool has the starting address independent of the addresses of the common blocks. Using this layout of memory and a temporary communication buffer caused the number of L3 cache misses in our CAF code to be up to a factor of three larger than for the corresponding MPI code, resulting in performance that was slower by a factor of five. By converting q (and other arrays used in co-array expressions) to co-arrays, it moved their storage allocation into the segment with co-array data (reducing the potential for conflict misses) and avoided the need for the temporary buffer. Overall, this change greatly reduced L3 cache misses and brought the performance of the CAF version back to level of the MPI code. Our lesson from this experience is that memory layout of communication buffers, co-arrays, and common block/save arrays might require thorough analysis and optimization.

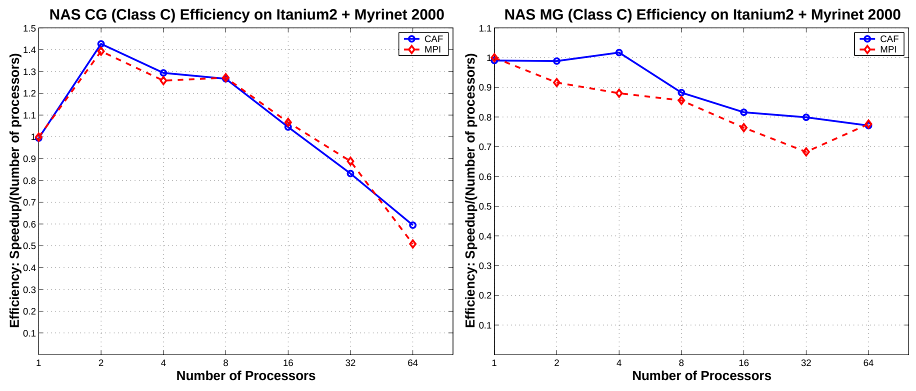
As Figure 4 (a) shows, our CAF version of NAS CG achieves performance comparable to that of the MPI version. The parallel efficiency of the CAF and MPI codes are almost indistinguishable across a range of processor numbers.

6.2 NAS MG

The MG multigrid kernel calculates an approximate solution to the discrete Poisson problem using four iterations of the V-cycle multigrid algorithm on a $n \times n \times n$ grid with periodic boundary conditions [9]. The communication is highly structured and goes through a fixed sequence of regular patterns.

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

The CAF version of MG mimics the MPI version. The communication buffers used in the MPI version are replaced by co-arrays; the communication is expressed using CAF syntax, as opposed to using MPI primitives. This approach requires explicit synchronization. The example code is shown on Figure 2 (b). The `give3` procedure per-



(a) NAS CG (b) NAS MG

Fig. 4. Comparison of MPI and CAF parallel efficiency for NAS CG and MG.

forms a one-sided put to the appropriate neighbor. Because of communication buffer reuse, two `sync_notify` are necessary to signal our left and right neighbors that our receive buffers are ready to receive data from them; two following `sync_wait` ensure that the remote buffers on the left and the right neighbors are ready for us to send data. The `sync_notify` following each `give3` call is matched by the neighbor's `sync_wait` and signals the completion of the put. Similarly, our `sync_wait` matches the neighbor's `sync_notify` signaling that the data transfer from the neighbor is complete and we can proceed to the unpacking phase in `take3`.

As the performance graph on Figure 4 (b) illustrates, our CAF version of NAS MG achieves comparable performance to that of the MPI version.

6.3 NAS SP and BT

As described in a NASA Ames technical report [9], the NAS benchmarks BT and SP are two simulated CFD applications that solve systems of equations resulting from an approximately factored implicit finite-difference discretization of three-dimensional Navier-Stokes equations. The principal difference between the codes is that BT solves block-tridiagonal systems of 5x5 blocks, whereas SP solves scalar penta-diagonal systems resulting from full diagonalization of the approximately factored scheme [9]. Both consist of an initialization phase followed by iterative computations over time steps. In each time step, boundary conditions are first calculated. Then the right hand sides of the equations are calculated. Next, banded systems are solved in three computationally intensive bi-directional sweeps along each of the x, y, and z directions. Finally, flow variables are updated. During each timestep, loosely-synchronous communication is required before the boundary computation, and tightly-coupled communication is required during the forward and backward line sweeps along each dimension.

Because of the line sweeps along each of the spatial dimensions, traditional block distributions in one or more dimensions would not yield good parallelism. For this reason, SP and BT use a skewed block distribution called multipartitioning [9, 14]. With

```
lhs( 1:BLOCK_SIZE, 1:BLOCK_SIZE,
     cc, -1,
     0:JMAX-1, 0:KMAX-1,
     cr) [successor(1)] =
lhs( 1:BLOCK_SIZE, 1:BLOCK_SIZE,
     cc, cell_size(1,c)-1,
     0:JMAX-1, 0:KMAX-1, c)
```

(a) NAS BT

```
.... pack into out_buffer_local......

out_buffer(1:p, stage+1:stage+1)
     [successor(1)] =
out_buffer_local(1:p, 0:0)

.... unpack from out_buffer..........
```

(b) NAS SP

Fig. 5. Forward sweep communication in NAS BT and NAS SP.

multi-partitioning, each processor handles several disjoint blocks in the data domain. Blocks are assigned to the processors so that there is an even distribution of work for each directional sweep, and that each processor has a block on which it can compute in each step of every sweep. Using multipartitioning yields full parallelism with even load balance while requiring only coarse-grain communication.

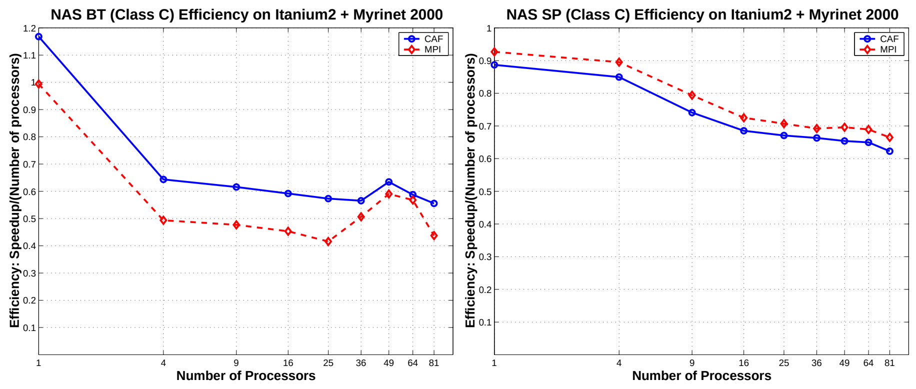
The MPI implementation of NAS BT and SP attempts to hide communication latency by overlapping communication with computation, using non-blocking communication primitives. For example, in the forward sweep, except for the last tile, non-blocking sends are initiated in order to update the ghost region on the next tile. Afterwards, each process advances to the next tile it is responsible for, posts a non-blocking receive, performs some local computation, then waits for the completion of both non-blocking send and receive. The same pattern is present in the backward sweep.

The CAF implementation for BT and SP inherits the multipartitioning scheme used by the MPI version. In BT, the main working data resides in co-arrays, while in SP it resides in non-shared arrays. For BT, during the boundary condition computation and during the forward sweep for each of the axes, no buffers are used for packing and unpacking, as shown in Figure 5(a). On the contrary, in SP all the communication is performed via co-array buffers (see Figure 5(b)). This shows that when an application is written in the spirit of the Co-array Fortran programming model, it might require less memory copies. In the backward sweep, both BT and SP use auxiliary co-array buffers to communicate data.

In our CAF implementations, we had to consider the trade-off between the amount of memory used for buffers and the amount of necessary synchronization. By using more buffer storage we were able to eliminate both output and anti-dependences due to buffer reuse, thus obviating the need for extra synchronization. We used a dedicated buffer for each communication event during the sweeps, for a total buffer size increase by a factor of square root of the number of processors. Experimentally we found that this was beneficial for performance while the memory increase was acceptable.

The performance graphs in figure 6 show that the CAF version performs consistently better than the MPI version for BT, but is about 5% slower for SP. For both benchmarks, our compiler uses blocking communication primitives. By applying hand optimization to the code generated by our CAF compiler for SP, we discovered that using non-blocking communication enables us to achieve performance comparable to that of MPI.

We observed that even though we used blocking communication for both BT and SP, we only paid a performance penalty for SP. This difference is due to the computation



(a) NAS BT (b) NAS SP

Fig. 6. Comparison of MPI and CAF parallel efficiency for NAS SP and BT.

and communication characteristics of the benchmarks. Measurements showed that BT communicates half the number of messages that SP does, whereas the communication volume is about 2/3 of the communication volume for SP. Therefore, in BT the communication is less frequent than in SP, and consists of larger messages. As a consequence, overlapping computation with communication is more critical for performance in SP.

6.4 Discussion

In the course of our experimentation on our Itanium2+Myrinet2000 cluster, we observed that allocating co-arrays and temporary communication buffers in registered memory provides a noticeable boost in performance. Myrinet is only able to perform DMA on registered (pinned) pages. If all local variables involved in communication are allocated in registered memory, they can be used by the communication library directly, without copying into temporary buffers allocated from a pool of registered memory. In our prototype compiler, we don't automatically migrate local variables involved in communication into pinned memory; instead, we accomplished this by modifying the source code to turn them into co-arrays that are never referenced remotely. Automatically migrating local variables into co-array storage can be complex to do because of the need to preserve sequence association among local variables in common blocks and Fortran data initialization statements.

Our experiments showed that using split-phase, non-blocking communication and overlapping computation with DMA transfers significantly boosts performance. Our prototype compiler implements communication by an in-place conversion of language-level communication constructs into blocking gets and puts. Once we have a framework for data-flow and dependence analysis in place, we will be able to automatically translate blocking communication into split-phase, non-blocking equivalents to effectively overlap communication with computation.

At a higher level, the original semantics of CAF as defined by Numrich and Reid [4, 5] require an implicit `sync_memory` at each procedure call boundary to complete any outstanding gets or puts. This requirement makes it impossible to overlap communication with a procedure call that does not use any of the data involved in communication. Requiring this implicit `sync_memory` for SP would remove a significant opportunity for latency hiding that is exploited by the MPI hand-coded parallelization, so we believe that this language requirement should be dropped.

In the first draft CAF versions of the NAS benchmarks, we found that there were frequent references to read-only data that was stored off-processor. For example, there were frequent off-processor references to the variable `reduce_send_start` in the initial CAF version of CG, to the communication buffer offsets for each face in SP, and to the cell size information of neighboring processors in BT. We improved code performance by fetching these values once after they had been initialized and storing them locally. With interprocedural analysis to determine that these variables are essentially run-time constants, we could potentially apply this transformation automatically.

7 Conclusions

This paper presents an overview of the issues that we have been grappling with as we work on (1) refinement of the CAF language to make it the programming model of choice for portable, high-performance scientific programming in Fortran and (2) design and implementation of a portable and retargetable CAF compiler. Preliminary performance results for several NAS benchmarks on a cluster of workstations show that CAF is capable of achieving good performance despite the current lack of automatic communication and synchronization optimizations in our prototype CAF compiler. We were able to achieve performance comparable to highly-tuned, hand-coded MPI versions of the same benchmarks. The expressive syntax and explicit, one-sided communication model enabled us to manually perform key optimizations such as communication vectorization and synchronization strength reduction in the CAF source code. The CAF model is expressive enough to allow the user to perform these transformations manually when a compiler cannot. This is in contrast to HPF, for example, where it is more difficult for a user to improve code performance through source code adjustments.

While we performed optimizing transformations manually on our CAF source code in this preliminary work, it is our intention to improve our prototype CAF compiler to perform automatically most of the optimizing transformations described in this paper. An advantage of writing parallel programs in CAF over MPI is that because communication and synchronization are expressed at the language level, it is possible for a compiler to analyze and tailor code to whatever target platform is at hand. On a shared memory architecture, CAF accesses to remote data can simply be turned into loads and stores; performing such a radical transformation on an MPI program would be exceedingly difficult. While it may be possible to annotate MPI libraries so that compilers could understand the semantics of the communication expressed by library calls, CAF offers a simpler, coherent model for parallel programming.

Because CAF is amenable to automatic analysis and transformation, it is possible and desirable to express computation and communication in a natural and general way, leaving the burden of platform-specific code tuning to the compiler. This is important because user-applied optimizations that perform well on one architecture may actually be counter-productive on a different architecture.

Acknowledgments

We thank D. Chavarría-Miranda for explaining the intricacies of the NAS benchmarks. We thank J. Nieplocha and V. Tipparaju for collaborating on the refinement and tuning of ARMCI on Myrinet. We thank R. Numrich and A. Wallcraft for providing us with draft CAF versions of the BT, CG, MG, and SP NAS parallel benchmarks. We thank F. Zhao for her work on the Open64/SL Fortran front-end.

References

1. Mattson, T.G.: An introduction to OpenMP 2.0. In: High Performance Computing. Volume 1940 of Lecture Notes in Computer Science. Springer-Verlag (2000) 384–390
2. Koelbel, C., Loveman, D., Schreiber, R., Steele, Jr., G., Zosel, M.: The High Performance Fortran Handbook. The MIT Press, Cambridge, MA (1994)
3. Gropp, W., Snir, M., Nitzberg, B., Lusk, E.: MPI: The Complete Reference. Second edn. MIT Press (1998)
4. Numrich, R.W., Reid, J.K.: Co-Array Fortran for parallel programming. Technical Report RAL-TR-1998-060, Rutheford Appleton Laboratory (1998)
5. Numrich, R.W., Reid, J.K.: Co-Array Fortran for parallel programming. ACM Fortran Forum **17** (1998) 1–31
6. Carlson, W.W., Draper, J.M., Culler, D.E., Yelick, K., E. Brooks, K.W.: Introduction to UPC and language specification. Technical Report CCS-TR-99-157, IDA Center for Computing Sciences (1999)
7. Silicon Graphics: CF90 co-array programming manual. Technical Report SR-3908 3.1, Cray Computer (1994)
8. Nieplocha, J., Carpenter, B.: ARMCI: A portable remote memory copy library for distributed array libraries and compiler run-time systems. In: Parallel and Distributed Processing. Volume 1586 of Lecture Notes in Computer Science. Springer-Verlag (1999) 533–546
9. Bailey, D., Harris, T., Saphir, W., van der Wijngaart, R., Woo, A., Yarrow, M.: The NAS parallel benchmarks 2.0. Technical Report NAS-95-020, NASA Ames Research Center (1995)
10. Nieplocha, J., Tipparaju, V., Saify, A., Panda, D.: 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 (2002)
11. Open64/SL Developers: Open64/SL compiler and tools. `http://hipersoft.cs.rice.edu/open64` (2003)
12. Open64 Developers: Open64 compiler and tools. `http://sourceforge.net/projects/open64` (2001)
13. Prakash, S., Dhagat, M., Bagrodia, R.: Synchronization issues in data-parallel languages. In: Proceedings of the 6th Workshop on Languages and Compilers for Parallel Computing. Volume 768. Springer-Verlag (1994) 76–95
14. Naik, V.: A scalable implementation of the NAS parallel benchmark BT on distributed memory systems. IBM Systems Journal **34** (1995)