A Multi-platform Co-Array Fortran Compiler*

Yuri Dotsenko Cristian Coarfa John Mellor-Crummey
Dept. of Computer Science, Rice University, 6100 Main, Houston, TX 77005
{dotsenko,ccristi,johnmc}@cs.rice.edu

ABSTRACT

Co-array Fortran (CAF)—a small set of extensions to Fortran 90—is an emerging model for scalable, global address space parallel programming. CAF's global address space programming model simplifies the development of single-program-multiple-data parallel programs by shifting the burden for managing the details of communication from developers to compilers. This paper describes `cafc`—a prototype implementation of an open-source, multiplatform CAF compiler that generates code well-suited for today's commodity clusters. The `cafc` compiler translates CAF into Fortran 90 plus calls to one-sided communication primitives. The paper describes key details of `cafc`'s approach to generating efficient code for multiple platforms. Experiments compare the performance of CAF and MPI versions of several NAS parallel benchmarks on an Alpha cluster with a Quadrics interconnect, an Itanium 2 cluster with a Myrinet 2000 interconnect and an Itanium 2 cluster with a Quadrics interconnect. These experiments show that `cafc` compiles CAF programs into code that delivers performance roughly equal to that of hand-optimized MPI programs.

1. INTRODUCTION

Parallel languages and parallelizing compilers have been the focus of compiler research for many years. Over the last decade, OpenMP [4] and High Performance Fortran (HPF) [9] are the parallel programming models that have received the most attention. However, neither of these models has been accepted for writing scalable, high-performance parallel programs for commodity clusters. OpenMP provides programmers with little control over data layout; as a result, OpenMP programs are difficult to map efficiently to distributed memory platforms. HPF, in contrast, enables programmers to explicitly control the mapping of data to processors; however, its abstract programming model makes compilation challenging and, 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) [8]—a library-based programming model that enables application developers to control the placement of data, computation and communication—has become the *de facto* standard for scalable parallel programming.

Dissatisfaction with the complexity of writing MPI programs has spurred a resurgence of research into language-based parallel programming models. Three languages have been the focus of recent attention as promising near-term alternatives to MPI: Co-array Fortran (CAF) [12, 13], Unified Parallel C (UPC) [2] and Titanium [18]. Each of these languages supports a Global Address Space (GAS) model for single-program-multiple-data (SPMD) parallel programming. These language models have three key strengths. First, they abstract away much of the complexity of communication that frustrates MPI programmers; one simply reads and writes shared variables to communicate. With communication and synchronization as language primitives, programs written in SPMD GAS languages are also more amenable to compiler-directed communication optimization than MPI programs. Second, like MPI, SPMD GAS languages enable programmers to hand-craft locality-aware parallelizations by providing them with control over placement of data, computation and communication. This approach provides better locality control than OpenMP and avoids HPF's vulnerability to compiler shortcomings. Third, since SPMD GAS models are less abstract than HPF, they are easier to compile effectively.

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 [17]. In this paper, we describe and evaluate the implementation of `cafc`—a portable, multi-platform, open-source compiler for a subset of the CAF language. The language subset currently supported by `cafc` is sufficiently broad that we can generate

*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 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.

code for non-trivial programs, including CAF versions of the NAS benchmarks [1]. We are working to complete remaining language features. Our aim is to create a high-quality CAF compiler that can deliver excellent performance for production codes on a wide range of scalable platforms. `cafc` performs source-to-source translation of CAF into Fortran 90 plus calls to a multi-platform library for one-sided communication. Today, `cafc` automatically applies two optimizations: procedure splitting to improve computation performance and run-time use of non-blocking communication guided by user hints. To date, we have run code generated by `cafc` on several platforms, including Itanium2+Myrinet, Itanium2+Quadrics, Alpha+Quadrics, Pentium+Ethernet, SGI Altix and SGI Origin 2000. Here, we evaluate the quality of `cafc`-generated code using the NAS benchmarks on Alpha+Quadrics, Itanium2+Myrinet and Itanium2+Quadrics platforms to assess the ability of our multi-platform strategy to deliver high performance. Our results show that even though `cafc` currently lacks communication optimization, with careful coding one can already use it to achieve performance that is roughly equal to hand-tuned MPI.

In section 2, we briefly describe the CAF language. In section 3, we outline key features of the implementation of `cafc`. In Section 4, we describe performance optimizations implemented in `cafc`. In Section 5, we report results of experiments using versions of the NAS CG, MG, BT, SP and LU benchmarks to compare the performance of CAF and MPI and evaluate the effectiveness of `cafc`'s optimizations. Section 6 presents our conclusions and future plans for `cafc`.

2. CO-ARRAY FORTRAN

CAF is a global address space programming model that supports SPMD parallel programming through a small set of language extensions to Fortran 90. An executing CAF program consists of a static collection of asynchronous *process images*. CAF provides a two-level memory model: data is explicitly local or remote. This enables CAF programs to explicitly manage locality by carefully controlling the distribution of data and computation.

In CAF, one declares shared, distributed data—called co-arrays–using an extension to Fortran 90 declaration syntax. For example, the declaration `integer :: x(n,m)[*]` declares a shared co-array with a `n` × `m` integer array local to each process image. The dimension inside brackets is called a co-dimension; it is used to index data on remote process images. The upper bound of the last co-dimension must always be `*` to support an arbitrary number of process images. Any co-array reference that lacks co-dimension subscripts refers to a process image's local co-array data. A CAF process image can directly reference non-local co-array 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]`. A co-array can be declared with multiple co-dimensions; the declaration `integer :: x(n,m)[k,*]` logically organizes process images into `k` rows. Co-arrays may 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 (scalar co-array) or an array of type instances.

CAF includes several synchronization primitives; the most important of them are `sync_all`, which implements a synchronous barrier, and `sync_team`, which is used for barrier-style synchronization among dynamically-formed *teams* of two or more processes. For better performance, we extended CAF to include support for point-to-point notify and wait synchronization primitives [3].

Since both remote data access and synchronization are language primitives in CAF, they are amenable to compiler-based optimization. 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 [13].

3. IMPLEMENTATION STRATEGY

We designed the `cafc` compiler for CAF with the major goals of being portable and delivering high-performance on a plethora of platforms.To support multiple platforms effectively, `cafc` performs source-to-source transformation of CAF code into Fortran 90 code augmented with platform-specific communication generated using an abstract communication generation interface. Internally, `cafc`'s abstract communication generation interface enables it to separate analysis and optimization of communication from the details of any particular communication layer. Currently, we have two instantiations of this interface. One generates multi-platform code using the Aggregate Remote Memory Copy Interface (ARMCI) [11] library for one-sided communication [5]. The second generates code for shared-memory multiprocessors using loads and stores for communication. `cafc`'s source-to-source code generation strategy enables it to optimize local computation for the target platform using the best Fortran 90 compiler available. The `cafc` compiler is implemented using Open64/SL [15], a version of the Open64 compiler infrastructure [14] that we have modified to support source-to-source transformation of Fortran 90. Members of our research group modified a version of the Open64 infrastructure so that `cafc` can run natively on multiple architectures and operating systems.

3.1 Memory management

To support efficient access to remote co-array data on the broadest range of platforms, memory for co-arrays is managed by the communication substrate separately from memory managed conventionally by a Fortran 90 compiler's language run-time system. Having the communication substrate control allocation of co-array memory enables our generated code to use the most appropriate allocation strategy for that platform. For instance, on a Myrinet 2000-based cluster, `cafc` generates code that uses ARMCI to allocate data for co-arrays in pinned physical memory; this enables ARMCI to perform data transfers on the memory directly using the Myrinet adapter's DMA engine.

Separately managing co-array data requires special mechanisms for allocating and initializing SAVE and COMMON co-array variables. When generating ARMCI-based communication, `cafc` replaces declarations of static co-arrays with descriptors for the separately allocated co-array storage. (The descriptors themselves are explained in the next section.)

When `cafc`-generated code begins execution, it performs a two-step initialization process. First, it allocates storage for co-arrays. Second, it initializes procedure-level views of SAVE and COMMON co-arrays by associating co-array descriptors with the allocated memory and the communication substrate's run-time state.

For each procedure containing SAVE co-arrays, `cafc` generates an initialization routine that allocates memory for each SAVE co-array and sets up a descriptor for the co-array. For COMMON blocks, the process is similar, except that separate routines are generated for allocating storage for COMMON co-arrays and initializing co-array descriptors. We explain the handling of COMMON blocks in more detail in Section 3.3 in the context of an explanation of how `cafc` supports sequence association for COMMON co-arrays. When linking a CAF program, `cafc` first examines the object files to collect the names of all storage allocators and co-array descriptor initializers. Next, `cafc` synthesizes a global initializer that calls each allocator and initializer. The global initializer is called once at program launch before any user-written code executes.

3.2 Co-array descriptors

For CAF programs to perform well, access to the local portions of co-arrays must be efficient. Since co-arrays are not natively supported in Fortran 90, `cafc` must translate references to local co-array data into Fortran 90 syntax that will enable them to access local co-array data with loads and stores. In `cafc`'s generated code, co-arrays are represented using *co-array descriptors*. Co-array descriptors reside in the local memory of each process image.

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

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

When using ARMCI, a co-array's `handle` refers to a run-time representation that currently contains the base virtual address of the co-array storage for each process image and the raw size of each process's local co-array data. Co-array shape and co-shape information is not represented explicitly in the run-time layer.

Since co-array data in `cafc`'s generated code is allocated by the communication substrate outside control of the Fortran 90 run-time system, `cafc`'s run-time library needs the ability to initialize and manipulate compiler-dependent Fortran 90 pointer representations (known as dope vectors) on a variety of target platforms. Currently, `cafc`'s run-time library uses a home-grown approach for initializing dope vectors; eventually, we will update it to use the CHASM library [16] from Los Alamos National Laboratory for this purpose.

3.3 Sequence association and reshaping

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

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

3.4 Parameter passing

CAF allows programmers to pass co-arrays as arguments to procedures. According to CAF specification [12, 13], there are two types of co-array argument passing: by-value and by-co-array.

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

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

We propose an extension to co-array parameter passing that we found to be useful in the NPB benchmark codes. We support a pass by-reference convention, in which the callee

receives the local part of a co-array as array argument and treats it as a regular array. This enables the programmer to reuse subroutines that compute over arrays for processing local parts of co-arrays. Fortran 90 interfaces are used to differentiate what type of calling convention should be used. An example is shown in Figure 1.

```
interface
  subroutine foo(a)
    double precision a[*]
  end subroutine foo

  subroutine bar(b)
    double precision b
  end subroutine bar
end interface

double precision x(10,10)[5,*]

...

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

Figure 1: Using Fortran 90 interfaces to specify by-co-array and by-reference argument passing styles.

When declaring a procedure interface that receives a co-array by reference, the dummy argument's shape (and co-shape) information is omitted. This provides symmetry for specifying by-reference argument passing for arrays and co-arrays. A callee receiving a co-array argument declares fresh shape and co-shape information; this can be used to reshape a co-array in the callee if desired.

3.5 Supporting multiple co-dimensions

The CAF programming model does not limit the programmer to using a flat co-space. Instead, the user can specify a multidimensional co-space, with the same column-major convention as regular Fortran code. This feature is of most use when the processor space of a problem is logically mapped onto a processor grid. The programmer has the ability to mold the co-space to fit the logical processor grid. Indexing of a multi-dimensional organization of remote images is then straightforward using this feature.

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

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

$$\mathtt{m}_j = \prod_{k=1}^{j-1} (\mathtt{ub}_k - \mathtt{lb}_k + 1) \ , \ 2 \le j \le n+1 \tag{2}$$

Section 3.2 explains co-array descriptors. In order to support co-arrays with multiple co-dimensions, we augment the co-array metadata used in `cafc`-generated code with several co-space variables. For a co-array `a` with the co-space definition $[\mathtt{lb}_1 : \mathtt{ub}_1, \mathtt{lb}_2 : \mathtt{ub}_2, \ldots, \mathtt{lb}_n : \mathtt{ub}_n, \mathtt{lb}_{n+1} : *]$, we added the following variables:

- `a_coLB_i`, for $1 \le i \le n+1$
- `a_coUB_i`, for $1 \le i \le n$
- `a_ThisImage_i`, for $1 \le i \le n+1$
- `a_CoIndexMultiplier_i`, for $1 \le i \le n+1$
- `a_ThisImageVector`

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

We extend the initialization routines mentioned in section 3.1 to set up the co-space metadata variables presented above. `a_coLB_i`, `a_coUB_i` are trivially assigned using the co-array definition. The variables `a_CoIndexMultiplier_i` are computed iteratively using the formulas (1) and (2) for `m_i`. To compute `a_ThisImage_i` we use the process image number returned by `this_image` as follows:

`a_ThisImage_i` =
$\mathtt{mod}((\mathtt{this\_image}() - 1), \mathtt{a\_coIndexMultiplier\_i}) + \mathtt{a\_coLB}_i$.

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

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

3.6 Communication code generation

Communication events expressed with CAF's bracket notation must be converted into Fortran 90; however, this is not straightforward because the remote memory may be 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 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 [5]. The situation is more complicated for cluster-based systems with distributed memory.

To perform data movement on clusters, the compiler must generate calls to a communication library to access data that resides on a remote node. Local temporary storage is sometimes needed for off-processor data. For example, in the case of a read reference of a co-array on another image, `arr(:)=coarr(:)[p] + ...`, a temporary, `tmp`, is allocated just prior to the statement to hold the value of the `coarr(:)` array section from image `p`. Then, a call to get data from image `p` is issued to the run-time library. The statement is rewritten as `arr(:) = tmp(:) + ...`. The temporary is deallocated immediately after the statement. For a write to a remote image, such as `coarr(:)[p1,p2]=...`, a temporary `tmp` is allocated prior to the remote write statement; the result of the evaluation of the right-hand side is stored in the temporary; next, a call to the abstract communication interface is issued to perform the write; finally, the temporary is deallocated.

When possible, the generated code avoids using unneeded temporary buffers. For example, when both the left hand-side and the right-hand side of an assignment are co-arrays, `cafc` generates calls to the abstract communication interface that directly access the involved co-arrays. Besides avoiding the cost of allocating and copying into the temporary buffer (and potential cache conflicts), this also enables us to exploit zero-copy data transfer capabilities of the underlying communication library, whenever they are supported.

In section 4.2 we present a practical run-time strategy that enables use of non-blocking communication.

3.7 Synchronization

The original CAF synchronization model used several primitives, such as `sync_all` and `sync_team`. The semantics of these primitives are described in Section 2. We have implemented `sync_all` by issuing a call to the barrier provided by ARMCI. The intrinsic `sync_team(team,[wait])` was implemented by issuing a series of notification to the team members (specified by the integer vector `team`), then waiting for notifications from either the `team` process images (if the `wait` argument is absent), or from the `wait` process images. These primitives sufficed to fully express the NAS benchmarks MG, CG, BT, SP and LU.

An issue that arose during our application evaluation was that using the synchronization primitives provided by CAF reduced the performance of the applications we studied. 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.

In [3], we proposed `sync_notify(q)` and `sync_wait(p)` as two new primitives for point-to-point synchronization. When a process executes a `sync_notify`, it initiates notification of the specified process image and then continues immediately. When a process executes a `sync_wait(p)`, it blocks until it is notified by the process image `p`. 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.

3.8 Intrinsic functions

`cafc` supports the CAF intrinsic functions: `log2_images()`, `this_image()`, `num_images()` and `rem_images()`. To implement them efficiently, we precompute their values at program launch and store them into scalars. At compile time, calls to these functions are replaced by references to the corresponding scalars. A more complicated strategy is employed to support `this_image` relative to a co-array. As mentioned in section 3.5, we compute the components of `this_image` once at program initialization for SAVE and COMMON co-arrays, and once per procedure invocation for dummy co-arrays. We replace calls to `this_image(a)` with a reference to `a_ThisImageVector`. We replace calls to `this_image(a,i)` with a scalar variable if `i` is a compile-time constant, and if `i` is a variable, we use an array reference into `a_ThisImageVector`.

3.9 Ongoing work

The following features of CAF are currently not supported: user-defined type co-arrays, allocatable co-arrays, allocatable co-array components, triplets in co-dimensions, parallel I/O and remaining Co-array Fortran intrinsic functions. Ongoing work is aimed at removing these limitations.

4. OPTIMIZATIONS

In this section, we describe two optimizations implemented in `cafc`: procedure splitting and support for overlapping communication with computation and/or other communication. Finally, we discuss a third optimization—packing of strided communication—that we only experimented with manually.

4.1 Procedure splitting

In early experiments comparing the performance of CAF programs compiled by `cafc` with the performance of Fortran+MPI versions of the same programs, we observed that loops accessing local co-array data in the CAF programs were often significantly slower than the corresponding loops in the Fortran+MPI code, even though the source code for the computational loops were identical. Consider the following lines that are common to both the CAF and Fortran+MPI versions of the `compute_rhs` subroutine of the NAS BT benchmark. (NAS BT is described in Section 5.3.)

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

In both the CAF and Fortran+MPI sources, `u` and `rhs` reside in a single COMMON block. The CAF and Fortran+MPI versions of the program declare identical data dimensions for these variables, except that the CAF code adds a single co-dimension to `u` and `rhs` by appending a "`[*]`" to the end of its declaration. As described in Section 3.2, `cafc` rewrites the declarations of the `u` and `rhs` co-arrays with co-array descriptors that use a deferred-shape representation for co-array data. References to `u` and `rhs` are rewritten to use Fortran 90 pointer notation as shown here:

```
rhs%ptr(1,i,j,k,c) = rhs%ptr(1,i,j,k,c) + dx1tx1 * &
```

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

Our experiments showed that the performance differences we observed between the `cafc`-generated code and its Fortran+MPI counterpart result in part from the fact that the Fortran 90 compilers we use to compile `cafc`'s generated code conservatively assume that the pointers `rhs%ptr` and `u%ptr` might alias one another.[1] Overly conservative assumptions about aliasing inhibit optimizations.

We addressed this performance problem by introducing an automatic, demand-driven procedure-splitting transformation. We split each procedure that accesses SAVE or COMMON co-array variables into a pair of outer and inner procedures[2]. We apply this transformation prior to any compilation of co-array features. Pseudo-code in Figure 2 illustrates the effect of the procedure-splitting transformation.

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

(a) Original procedure

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

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

(b) Outer and inner procedures after splitting.

Figure 2: Procedure splitting transformation.

The outer procedure retains the same procedure interface as the original procedure. The outer procedure's body contains solely its data declarations, an interface block describing the inner procedure, and a call to the inner procedure. The inner procedure is created by applying three changes to the original procedure. First, its argument list is extended to account for the SAVE and COMMON co-arrays that are now received as arguments. Second, explicit-shape co-array declarations are added for each additional co-array received as an argument. Third, each reference to any SAVE or COMMON co-array now also available as a dummy argument is replaced to use the dummy argument version instead. In Figure 2, this has the effect of rewriting the reference to `c(50)` in `f` with a reference to `c_arg(50)` in `f_inner`.

[1]Compiling the `cafc`-generated code for the Itanium2 using Intel's `ifort` compiler (version 8.0) with the `-fno-alias` flag removed some of performance difference in computational loops between the CAF and Fortran+MPI codes.

[2]Our prototype currently supports procedure splitting only for subroutines; splitting for functions will be added soon.

After procedure splitting, the translation process for implementing co-arrays, as described in Section 3, is performed. The net result after splitting and translation is that within the inner procedure, SAVE and COMMON co-arrays that are now handled as dummy arguments are represented using explicit-shape arrays rather than deferred-shape arrays. Passing these co-arrays as arguments to the inner procedure to avoid accessing SAVE and COMMON co-arrays using Fortran 90 pointers has several benefits. First, Fortran compilers may assume that dummy arguments to a procedure do not alias one another; thus, these co-arrays are no longer assumed to alias one another. Second, within the inner procedure, the explicit-shape declarations for co-array dummy arguments retain explicit bounds that are otherwise obscured when using the deferred-shape representation for co-arrays in the generated code that was described in Section 3.2. Third, since local co-array data is referenced in the inner procedure as an explicit-shape array, it is known to be contiguous, whereas co-arrays referenced through Fortran 90 pointers may be strided. Our experiments also showed that knowing that data is contiguous improves software prefetching (as well as write hinting in Compaq's Fortran 90 compiler). The overall performance benefits of this transformation are evaluated in Section 5.

4.2 Hints for non-blocking communication

Overlapping communication and computation is an important technique for hiding interconnect latency as well as a means for tolerating asynchrony between communication partners. However, as CAF was originally described [13], all communication must complete before each procedure call in a CAF program. In a study of our initial implementation of `cafc`, we found that obeying this constraint and failing to overlap communication with independent computation hurt performance [3].

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

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

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

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

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

```
call complete_nb_put_region(region_id)
```

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

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

4.3 Strided vs. contiguous transfers

It is well-known that transferring one large message instead of many small messages in general is much cheaper on loosely-coupled architectures. With the column-major layout of co-arrays, a single language-level communication event, such as `a(i,1:n)[p]=b(j,1:n)`, might lead to `n` one-element transfers, which can be very costly. To overcome this performance hurdle, an effective solution is to pack strided data on the source, and unpack it on the destination. There can be several levels in the runtime environment where the data can be packed and unpacked to ensure efficient transfers.

In the CAF program This approach requires some effort on the programmer's side and can preclude CAF compiler from optimizing code for tightly-coupled architectures, such as the Cray X1.

By the CAF compiler In a one-sided communication programming paradigm, a major difficulty to pack / unpack data on this level is to transform one-sided communication into two-sided. For a PUT, the CAF compiler can easily generate packing code, but it is difficult to infer where in the program to insert the unpacking code so the receiving image unpacks data correctly. Similar complications arise for GETs. If Active Messages [6] are supported on a target platform, `cafc` could potentially generate packing code for the source process and an unpacking code snippet to execute on the destination.

In the runtime library This is the most convenient level in the runtime environment to perform packing / unpacking of strided communication. An optimized runtime library can use a cost model to decide if it is beneficial to pack data for a strided transfer. The ARMCI library used by our CAF compiler runtime library already performs packing / unpacking of data for Myrinet. However, we discovered that it does not currently do packing for Quadrics. Instead, ARMCI relies on Quadrics driver support for strided transfers, which deliver poor performance. We are coordinating with the ARMCI developers regarding this issue.

5. EXPERIMENTAL EVALUATION

In this section we compare the performance of the code `cafc` generates from CAF with hand-coded MPI implementations of the NAS MG, CG, BT, SP and LU parallel benchmark codes. The NPB codes are widely regarded as useful for evaluating the performance of compilers on parallel systems. 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.

For each benchmark, we compare the parallel efficiency of MPI and `cafc`-generated code for each benchmark. We compute parallel efficiency as follows. For each parallel version 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.

To evaluate the performance of CAF programs optimized by `cafc` we performed experiments on three cluster 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 a Quadrics interconnect (Elan3). We used the Compaq Fortran 90 compiler V5.5. The second platform was a cluster of 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 third platform was a cluster of 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 Linux (kernel version 2.4.20). The back-end compiler is the Intel Fortran compiler version 8.0. For all three platforms we used only one CPU per node to avoid memory contention.

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 on both architectures studied.

5.1 NAS CG

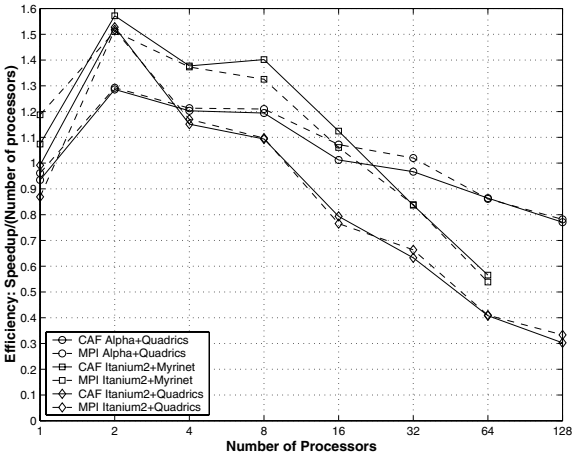

Figure 3: Comparison of MPI and CAF parallel efficiency for NAS CG on Alpha+Quadrics, Itanium2+Myrinet and Itanium2+Quadrics clusters.

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 [1]. 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.

Our previous study [3] revealed that the important CAF optimizations are: communication vectorization, synchronization strength-reduction and data layout management for co-array and non-coarray data. Here we describe experiments with NAS CG class C (size 150000, 75 iterations). Figure 3 shows that on the Alpha+Quadrics and the Itanium2+Quadrics clusters our CAF version of CG achieves comparable performance to that of the MPI version. The CAF version of CG consistently outperforms the MPI version for all the parallel runs on Itanium2+Myrinet.

Experiments with CG have showed that using PUTs instead of GETs on the Quadrics platforms yields performance improvements of up to 8% for large scale jobs on the Alpha + Quadrics platform and up to 3% on the Itanium2+Quadrics platform.

5.2 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 $\mathtt{n} \times \mathtt{n} \times \mathtt{n}$ grid with periodic boundary conditions [1]. The communication is highly structured and goes through a fixed sequence of regular patterns.

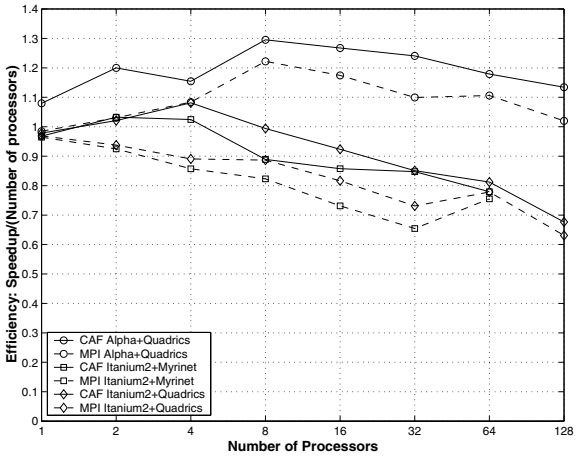

Figure 4: Comparison of MPI and CAF parallel efficiency for NAS MG on Alpha+Quadrics, Itanium2+Myrinet and Itanium2+Quadrics clusters.

Our previous study [3] revealed that the important CAF optimizations for MG are: communication vectorization, synchronization strength-reduction and conversion of GETs into PUTs. Figure 4 illustrates that our CAF version of NAS MG class C (512^3, 20 iterations) achieves performance superior to that of the MPI version on all three platforms. On the Alpha+Quadrics cluster, our CAF version outperforms MPI by up to 16% (11% on 128 processors); on the Itanium2+Myrinet cluster, the CAF version of MG exceeds the MPI performance by up to 30% (3% on 64 processors); on the Itanium2+Quadrics cluster, MG CAF surpasses MPI by up to 18% (7% on 128 processors). The best-performing CAF version uses procedure splitting and non-blocking communication.

5.3 NAS SP and BT

As described in a NASA Ames technical report [1], 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 [1]. SP and BT use skewed block distribution called multipartitioning [1, 10].

The MPI implementation of NAS BT and SP attempts to hide communication latency by overlapping communication with computation, using non-blocking communication prim-

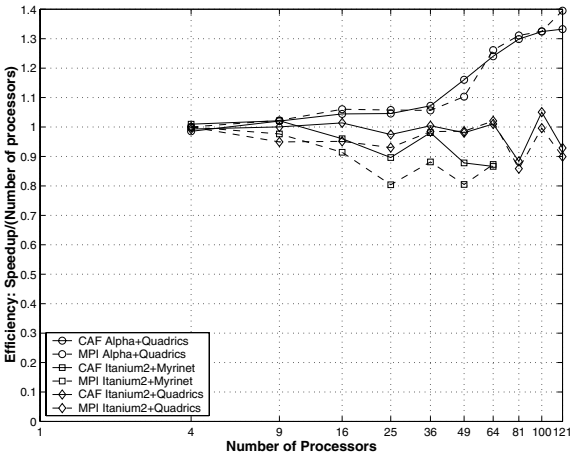

Figure 5: Comparison of MPI and CAF parallel efficiency for NAS BT on Alpha+Quadrics, Itanium2+Myrinet and Itanium2+Quadrics clusters.

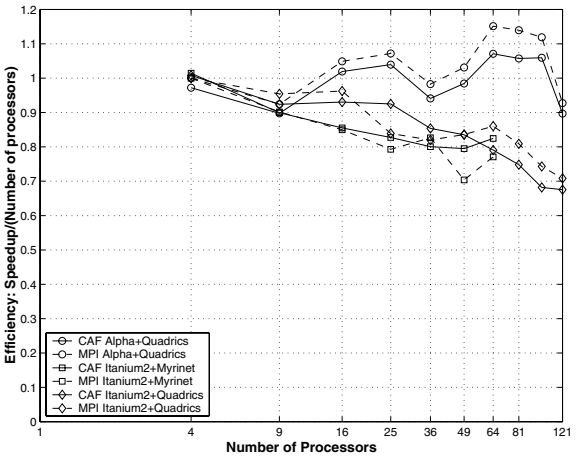

Figure 6: Comparison of MPI and CAF parallel efficiency for NAS SP on Alpha+Quadrics, Itanium2+Myrinet and Itanium2+Quadrics clusters.

itives. We explained our approach to implement CAF versions of BT and SP in our previous study [3]. The high-payoff code transformations were communication vectorization and trade-off between communication buffer space and amount of necessary synchronization. This study discovered that for some architectures the code shape of the input CAF program and the Fortran 90 translation is important for achieving high-performance.

The performance achieved by the CAF versions of BT class C (162^3, 200 iterations) and SP class C (162^3, 400 iterations) are presented in Figures 5 and 6. On the Alpha+Quadrics cluster, the performance of the CAF version of BT is comparable to that of the MPI version. On the Itanium2+Myrinet cluster, CAF BT outperforms the MPI versions by as much as 8% (and is comparable for 64 processors); on the Itanium2+Quadrics cluster, our CAF version of BT exceeds the MPI performance by up to 6% (3% on 121 processors). The CAF versions of SP is outperformed by MPI on the Alpha+Quadrics cluster by up to 8% and Itanium2+Quadrics clusters by up to 9%. On the Itanium2+Myrinet cluster, SP CAF exceeds the performance of MPI CAF by up to 7% (7% on 64 processors). The best performing CAF versions of BT and SP use procedure splitting, packed PUTs and non-blocking communication generation.

5.4 NAS LU

LU solves the 3D Navier-Stokes equation as do SP and BT. LU implements the solution by using a Successive Over-Relaxation (SSOR) algorithm which splits the operator of the Navier-Stokes equation into a product of lower-triangular and upper-triangular matrices (see [1] and [7])). The algorithm solves five coupled nonlinear partial differential equations, on a 3D logically structured grid, using an implicit pseudo-time marching scheme. The MPI code requires a power-of-two number of processors. The problem is partitioned on processors by repeatedly halving the grid in the dimensions x and y, alternately, until all power-of-two processors are assigned. This results in vertical pencil-like grid partitions on processors. The computations perform a sweep starting with one corner in a z plane to the opposite corner of the same z-plane; next it proceeds to the following z-plane. The communication of partition boundaries occurs after the computation is complete on all diagonals that contact an adjacent partition. This method effectively performs a diagonal pipelining and is called "a wavefront" by its authors; it has the potential of generating a relatively large number of small messages of 5 words each.

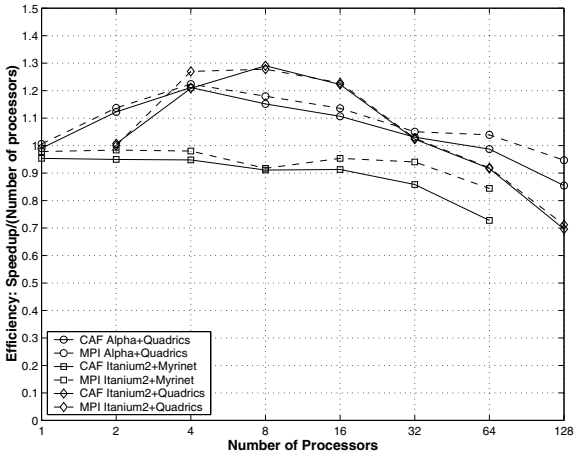

Figure 7: Comparison of MPI and CAF parallel efficiency for NAS LU on Alpha+Quadrics, Itanium2+Myrinet and Itanium2+Quadrics clusters.

Our CAF implementation follows closely the MPI implementation. We have transformed into co-arrays the grid parameters, the field variables and residuals, the output control parameters and the Newton-Raphson iteration control parameters. Local computation is similar to that of MPI. The various exchange procedures use co-arrays with two co-dimensions in order to naturally express communication with neighbors in four directions: north, east, south and west. For example, a processor with the co-indices `[row,col]` will send data to `[row+1,col]` when it needs to communicate to the south neighbor and to `[row,col-1]` for the west neighbor.

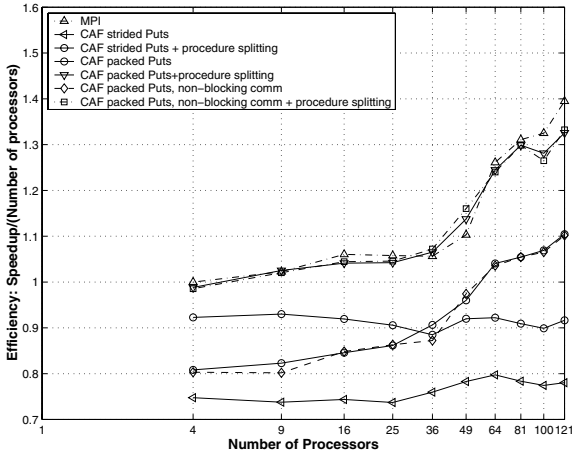

Figure 8: Parallel efficiency for several CAF versions of NAS BT on an Alpha+Quadrics cluster.

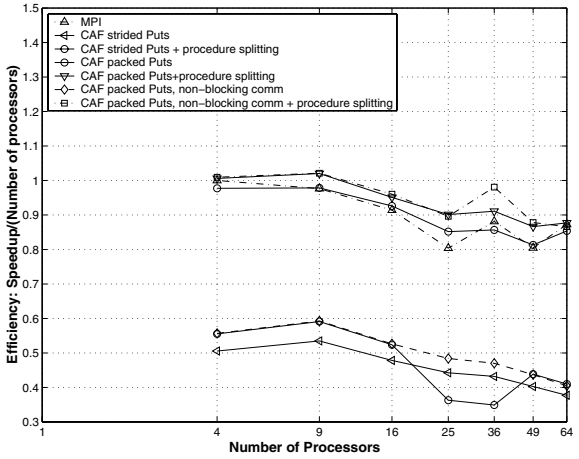

Figure 9: Parallel efficiency for several CAF versions of NAS BT on an Itanium2+Myrinet cluster.

The experimental results for the MPI and CAF versions of LU class C (162^3, 250 iterations) on all platforms are presented in Figure 7. On the Alpha+Quadrics cluster the MPI version outperforms the CAF version by up to 9%; on the Itanium2+Myrinet cluster, MPI LU exceeds the performance of CAF LU by as much as 13%. On the Itanium2+Quadrics cluster, the CAF and MPI versions of LU achieve comparable performance. The best performing CAF version of LU uses packed PUTs and procedure splitting.

5.5 Impact of optimizations

In section 4, we describe several optimizations to improve the performance of CAF programs: procedure splitting, issuing of non-blocking communication and communication packing. To experimentally evaluate the impact of each optimization, we implemented several versions of each of the NPB benchmarks presented above. In Figures 8 and 9, we present results on the Alpha+Quadrics and the Itanium2 + Myrinet clusters for the MPI version of BT and the following BT CAF versions: strided PUTs, strided PUTs with procedure splitting, packed PUTs, packed PUTs with procedure splitting, packed non-blocking PUTs and packed non-blocking PUTs with procedure splitting. In Figures 10 and 11, we present results on the Alpha+Quadrics and the Itanium2 + Myrinet clusters for the MPI version of LU and the following CAF versions: strided PUTs, strided PUTs with procedure splitting, packed PUTs and packed PUTs with procedure splitting. For both BT and LU the communication packing is performed at source level.

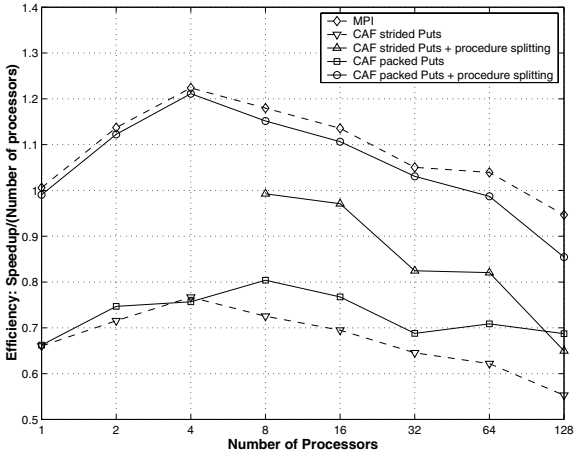

Figure 10: Parallel efficiency for several CAF versions of NAS LU on an Alpha+Quadrics cluster.

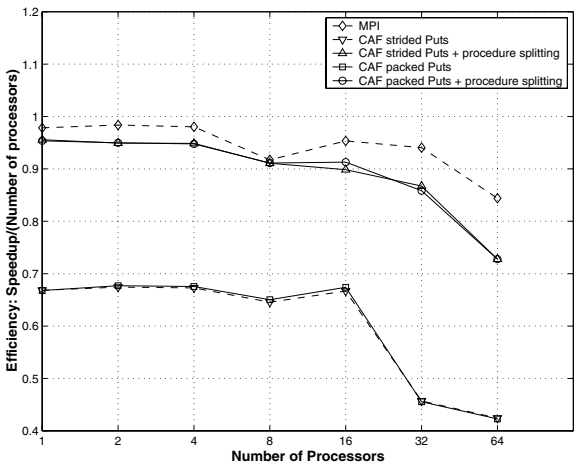

Figure 11: Parallel efficiency for several CAF versions of NAS LU on an Itanium2+Myrinet cluster.

For BT, procedure splitting is a high-impact transformation: it improves the performance by 13–20% on the Alpha+Quadrics cluster and by 42–60% on the Itanium2 + Myrinet cluster. For LU, procedure splitting yields an improvement of 15–33% on Alpha+Quadrics and 29–42% on Itanium2 + Myrinet. The CAF versions of BT and LU benefit significantly from the procedure splitting optimization because SAVE and COMMON co-arrays are heavily used in local computations. For benchmarks such as CG, MG and

SP, where co-arrays are used solely for data movement (by packing data, sending it and unpacking it on the destination) the benefits of the procedure splitting are modest. In addition, procedure splitting doesn't degrade performance for any of the programs we used in our experiments.

For BT, non-blocking PUTs improved performance by up to 2% on the Alpha+Quadrics platform, by up to 7% on the Itanium2+Myrinet platform and by up to 5% on the Itanium2+Quadrics platform. For MG, non-blocking PUTs improved performance by up to 3% on all platforms. For SP, non-blocking communication improved performance as much as 8% on Itanium2+Myrinet, though only up to 2% on the Quadrics clusters.

Packing data and performing contiguous rather than strided PUTs yields a performance improvment on both Quadrics platforms, on which the ARMCI library does not provide automatic packing. On the Myrinet platform, ARMCI supports data packing for communication, and thus there is no improvement from packing data at source level in CAF applications. For BT CAF, the execution time is improved up to 31% on the Alpha+Quadrics cluster and up to 30% on the Itanium2+Quadrics cluster. For LU CAF, the improvement is up to 24% on the Alpha+Quadrics cluster and up to 37% on the Itanium2+Quadrics cluster.

Our best guess for why the efficiencies of applications we studied degrade more sharply on Itanium2 platforms as the number of processors is increased is that the Itanium2 processors are much faster than the EV68 processors and thus the applications become communication bound earlier.

6. CONCLUSIONS

Co-array Fortran'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. Since the details of communication implementation are not embedded in Co-array Fortran programs, we believe that Co-array Fortran holds promise as a high performance programming model suitable for a wide range of platforms.

This paper describes the first implementation of an open-source, multiplatform compiler for CAF that generates code well-suited for today's commodity clusters. Our experiments with `cafc`-generated code for the NAS MG, CG, SP, BT, and LU benchmarks on several cluster architectures show that `cafc` delivers performance comparable to that of MPI. Our experiments showed that packed communication, procedure splitting and non-blocking communication are necessary to deliver high performance across a range of applications and platforms. The `cafc` compiler automatically applies two of these optimizations: procedure splitting and run-time use of non-blocking communication guided by user hints. However, without compile-time optimization of communication, including vectorization and aggregation, we have not yet realized our vision of supporting portable high-performance applications written in a natural style. A fundamental lesson is that *all* of these transformations are important for achieving high-performance on a wide range of codes and architectures.

Future work will focus on completing the CAF language implementation, communication optimization and harnessing the performance of emerging extreme-scale systems, such as IBM's Blue Gene.

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. We thank R. Numrich and A. Wallcraft for providing us with draft CAF versions of the BT, CG, MG, LU and SP NAS parallel benchmarks. We thank F. Zhao for her work on the Open64/SL Fortran front-end.

7. REFERENCES

[1] D. Bailey, T. Harris, W. Saphir, R. van der Wijngaart, A. Woo, and M. Yarrow. The NAS parallel benchmarks 2.0. Technical Report NAS-95-020, NASA Ames Research Center, Dec. 1995.

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

[3] C. Coarfa, Y. Dotsenko, J. Eckhardt, and J. Mellor-Crummey. Co-array Fortran Performance and Potential: An NPB Experimental Study. In *Proceedings of the 16th International Workshop on Languages and Compilers for Parallel Computing*, number 2958 in Lecture Notes in Computer Science, College Station, Texas, October 2-4, 2003. Intel Corp. and the Portland Group, Inc., Springer-Verlag. Published in 2004.

[4] L. Dagum and R. Menon. OpenMP: An Industry-Standard API for Shared-Memory Programming. *IEEE Comput. Sci. Eng.*, 5(1):46–55, 1998.

[5] Y. Dotsenko, C. Coarfa, J. Mellor-Crummey, and D. Chavarría-Miranda. Experiences with Co-Array Fortran on Hardware Shared Memory Platforms, 2004. Submitted to publication to LCPC2004, available at `http://hipersoft.cs.rice.edu/caf/publications/caf-lcpc2004.pdf`.

[6] T. v. Eicken, D. Culler, S. Goldstein, and K. Schauser. Active messages: A mechanism for integrated communication and computation. In *Proceedings of the 19th International Symposium on Computer Architecture*, Gold Coast, Australia, May 1992.

[7] M. Frunmkin, H. Jin, and J. Yan. Implementation of the nas parallel benchmarks in high performance fortran. Technical Report NAS-98-009, NAS Parallel Tools Groups, NASA Ames Research Center, Moffet Field, CA 94035, September 1998.

[8] W. Gropp, M. Snir, B. Nitzberg, and E. Lusk. *MPI: The Complete Reference*. MIT Press, second edition, 1998.

[9] C. Koelbel, D. Loveman, R. Schreiber, G. Steele, Jr., and M. Zosel. *The High Performance Fortran Handbook*. The MIT Press, Cambridge, MA, 1994.

[10] V. Naik. A scalable implementation of the NAS parallel benchmark BT on distributed memory systems. *IBM Systems Journal*, 34(2), 1995.

[11] 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.

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

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

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

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

[16] C. Rasmussen, M. Sottile, and T. Bulatewicz. CHASM language interoperability tools. `http://sourceforge.net/projects/chasm-interop`, July 2003.

[17] Silicon Graphics. CF90 co-array programming manual. Technical Report SR-3908 3.1, Cray Computer, 1994.

[18] K. A. Yelick, L. Semenzato, G. Pike, C. Miyamoto, B. Liblit, A. Krishnamurthy, P. N. Hilfinger, S. L. Graham, D. Gay, P. Colella, and A. Aiken. Titanium: A high-performance Java dialect. *Concurrency: Practice and Experience*, 10(11–13), September–November 1998.