Experiences with Co-Array Fortran on Hardware Shared Memory Platforms*

Yuri Dotsenko, Cristian Coarfa, John Mellor-Crummey, and Daniel Chavarría-Miranda

Rice University, Houston TX 77005, USA

Abstract. When performing source-to-source compilation of Co-array Fortran (CAF) programs into SPMD Fortran 90 codes for shared-memory multiprocessors, there are several ways of representing and manipulating data at the Fortran 90 language level. We describe a set of implementation alternatives and evaluate their performance implications for CAF variants of the STREAM, Random Access, Spark98 and NAS MG & SP benchmarks. We compare the performance of library-based implementations of one-sided communication with fine-grain communication that accesses remote data using load and store operations. Our experiments show that using application-level loads and stores for fine-grain communication can improve performance by as much as a factor of 24; however, codes requiring only coarse-grain communication can achieve better performance by using an architecture's tuned `memcpy` for bulk data movement.

1 Introduction

Co-array Fortran (CAF) [1] has been proposed as a practical parallel programming model for high-performance parallel systems. CAF is a global address space model for single-program-multiple-data (SPMD) parallel programming that consists of a small set of extensions to Fortran 90. To explore the potential of this programming model, we are building `cafc`—a multiplatform compiler for CAF. Our goal for `cafc` is to achieve *performance transparency*, namely, to deliver the full power of the hardware platform to the application on a wide range of parallel systems.

In this paper, we investigate how to generate efficient code for microprocessor-based scalable shared-memory multiprocessors with non-uniform shared memory access (NUMA). Such machines are organized as a set of nodes with each node containing one or more processors and memory. Nodes are connected using a low-latency, high-bandwidth interconnect. Each processor can access the memory on its node with low latency and memory on other nodes with higher latency. This class of systems includes platforms such as the SGI Altix [2], the SGI Origin [3]. Communication in

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

these systems occurs via cache line data transfers. Access to a data element on a remote node causes the cache line containing the data element to be fetched into the cache of the requesting node. On such systems, coarse-grain communication is accomplished by moving a group of cache lines individually.

`cafc` uses a source-to-source translation approach to code generation, transforming CAF into a Fortran 90 node program augmented with communication operations. This enables a separation of concerns: `cafc` can leave the details of back-end code optimization to a Fortran 90 compiler and focus on managing parallelism, communication and synchronization.

When transforming CAF into SPMD Fortran 90 node programs for shared-memory multiprocessors, there are several possible ways of representing and manipulating co-array data at the Fortran 90 language level. We explore several choices for representing shared data for co-arrays and accessing both local and remote co-array data. We evaluate the performance implications of these choices for several different codes including CAF variants of the STREAM [4], Random Access [5], Spark98 [6], and NAS MG & SP benchmarks [7]. We also compare the performance of the CAF versions against implementations of the same benchmarks written using MPI [8] and OpenMP [9], the most widely used parallel programming models.

In the next section, we briefly review the Co-array Fortran language and communication libraries used by our generated code. In Section 3, we describe the alternative code shapes that we investigate in this study. In Section 4, we describe the benchmark codes that we study and our experimental results comparing different strategies for representing and accessing shared data. We summarize our findings in Section 5.

2 Background

Co-Array Fortran. CAF is a global address space model for SPMD parallel programming that consists of a small set of extensions to Fortran 90. An executing CAF program consists of a static collection of asynchronous process images. CAF programs explicitly manage data locality and computation distribution. 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]`. CAF also includes synchronization primitives. Since both remote data access and synchronization are language primitives, they are amenable to compiler optimization. A more complete description of the CAF language can be found elsewhere [1].

Shared Memory Access Library (SHMEM). The SHMEM library [10], developed by SGI, provides an application programming interface (API) for NUMA machines such as the SGI Altix and Origin. For SPMD programs, SHMEM supports remote access to symmetric data objects—arrays or variables that exist with the same size, type

and relative address in all processes. Examples of symmetric data objects include Fortran COMMON block or SAVE variables and objects allocated from the symmetric heap [10]. The SHMEM API contains routines for data transfer using either contiguous or strided reads and writes, collective operations such as broadcast and reductions, barrier synchronization and atomic memory operations. SHMEM also supports remote pointers, which enable direct access to data objects owned by another process.

Aggregate Remote Memory Copy Interface. The `cafc` compiler generates code that uses the Aggregate Remote Memory Copy Interface (ARMCI) [11]—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. ARMCI supports non-contiguous data transfers. The latest version of ARMCI performs NUMA-aware memory allocation on the SGI Altix and Origin platforms using the SHMEM library's `shmalloc` primitive.

3 Implementing CAF on Shared Memory Architectures

The `cafc` compiler translates CAF programs into Fortran 90 node programs augmented with communication operations. In previous work [12], we described a translation strategy for generating portable code and performed a preliminary evaluation of the code's performance on several cluster architectures. The portable code we generate allocates memory for co-array data outside the Fortran 90 runtime system, initializes Fortran 90 pointers so that the node program can use them to access local co-array data, and performs communication using ARMCI `PUT` and `GET` operations.

As we experimented with `cafc`-generated code on more parallel architectures [13], we found that our generated code was not meeting our goal of performance transparency across the range of architectures and codes. While generating code to use Fortran 90 pointers to access local co-array data is a natural and portable approach, we found that in many cases the node performance of `cafc`-generated code using Fortran 90 pointers was often significantly slower than Fortran 90 code using arrays. Our experiments led us to conclude that performance irregularities we observed were a result of insufficient optimization of pointer-based codes by node compilers.

In [13], we described generating communication using ARMCI `PUT` and `GET` primitives. Though this approach is well-suited to cluster architectures, it fails to fully exploit the capabilities of shared-memory architectures. In contrast to clusters, shared memory architectures provide the ability to access remote memory directly via load and store instructions, which makes fine-grain remote accesses much more efficient. On shared-memory multiprocessors, Fortran 90 references can be used to access remote data directly, avoiding the overhead of calling library primitives for communication.

In this paper we compare Fortran 90 representations of COMMON block and SAVE co-arrays on scalable shared-memory multiprocessors to find the one that yields superior performance for both local computation and access to remote data. We report our findings for two NUMA SGI platforms (Altix 3000 and Origin 2000) and their corresponding compilers (Intel and SGI MIPSPro Fortran compilers). An important

conclusion of our study is that no single Fortran 90 co-array representation and code generation strategy yields the best performance across all architectures and Fortran 90 compilers. Moreover, two co-array representations can be used profitably together (one for effective local accesses, the other for effective remote accesses) to achieve the best results. An appealing characteristic of CAF is that a CAF compiler can automatically tailor code to a particular architecture and use whatever co-array representations, local data access methods, and communication strategies are needed to deliver the best performance.

3.1 Representing Co-arrays for Efficient Local Computation

To achieve the best performance for CAF applications, it is critical to support efficient computation on co-array data. Because `cafc` uses source-to-source translation into Fortran 90, this leads to the question of what is the best set of Fortran 90 constructs for representing and referencing co-array data. There are two major factors affecting the decision: (i) how well a particular back-end Fortran 90 compiler optimizes different kinds of data references, and (ii) hardware and operating system capabilities of the target architecture.

Most Fortran compilers effectively optimize references to COMMON block and SAVE variables, but fall short optimizing the same computation when data is accessed using Cray or Fortran 90 pointers. The principal stumbling block is alias analysis in the presence of pointers. COMMON block and SAVE variables as well as subroutine formal arguments in Fortran 90 cannot alias, while Cray and Fortran 90 pointers can. When compiling a CAF program, `cafc` knows that in the absence of Fortran EQUIVALENCE statements COMMON block and SAVE co-arrays occupy non-overlapping regions of memory; however, this information is not conveyed to a back-end compiler if `cafc` generates code to access local co-array data through pointers. Conservative assumptions about aliases cause back-end compilers to forgo critical performance optimizations such as software pipelining and unroll-and-jam, among others. Some, but not all, Fortran 90 compilers have flags that enable users to specify that pointers do not alias, which can ameliorate the effects of analysis imprecision.

Besides the aliasing problem, using Fortran 90 pointers to access data can increase register pressure and inhibit software prefetching. The shape of a Fortran 90 pointer is not known at compile time; therefore, bounds and strides are not constant and thus occupy extra registers, increasing register pressure. Also a compiler has no knowledge whether the memory pointed to by a Fortran 90 pointer is contiguous or strided, which complicates generation of software prefetch instructions.

The hardware and the operating system impose extra constraints on whether a particular co-array representation is appropriate. For example, on a shared-memory system a co-array should not be represented as a Fortran 90 COMMON variable if a COMMON block cannot be mapped into multiple process images. Below we discuss five possible Fortran 90 representations for the local part of a co-array variable `real a(10,20)[*]`.

Fortran 90 pointer. Figure 1(a) shows the representation of co-array data first used by `cafc`. At program launch, `cafc`'s run-time system allocates memory to hold `10` $\times$ `20` array of double precision numbers and initializes the `ca%local` field to point to it.

This approach enabled us to achieve performance roughly equal to that of MPI on an Itanium2 cluster with a Myrinet2000 interconnect using the Intel Fortran com-

```
type t1
  real, pointer :: local(:,:)
end type t1
type (t1) ca
```

(a) Fortran 90 pointer representation.

```
type t2
  real :: local(10,20)
end type t2
type (t2), pointer :: ca
```

(b) Pointer to structure representation.

```
real :: a_local(10,20)
pointer (a_ptr, a_local)
```

(c) Cray pointer representation.

```
real :: ca(10,20)
common /ca_cb/ ca
```

(d) COMMON block representation.

```
subroutine foo(...)
  real a(10,20)[*]
  common /a_cb/ a
  ...
end subroutine foo
```

(e) Original subroutine.

```
! subroutine-wrapper
subroutine foo(...)
  ! F90 pointer representation of a
  ...
  call foo_body(ca%local(1,1),...)
end subroutine foo

! subroutine-body
subroutine foo_body(a_local,...)
  real :: a_local(10,20)
  ...
end subroutine foo_body
```

(f) Parameter representation.

Fig. 1. Fortran 90 representations for co-array local data.

piler v7.0 (using a "no-aliasing" compiler flag) to compile `cafc`'s generated code [12]. Other compilers do not optimize Fortran 90 pointers as effectively. Potential aliasing of Fortran 90 or Cray pointers inhibits some high-level loop transformations in the HP Fortran compiler for the Alpha architecture. The absence of a flag to signal the HP Alpha Fortran compiler that pointers don't alias forced us to explore alternative strategies for representing and referencing co-arrays. Similarly, on the SGI Origin 2000, the MIPSPro Fortran 90 compiler does not optimize Fortran 90 pointer references effectively.

Fortran 90 pointer to structure. In contrast to the Fortran 90 pointer representation shown in Figure 1(a), the *pointer-to-structure* shown in Figure 1(b) conveys constant array bounds and contiguity to the back-end compiler.

Cray pointer. Figure 1(c) shows how a Cray pointer can be used to represent the local portion of a co-array. This representation has similar properties to the pointer-to-structure representation. Though the Cray pointer is not a standard Fortran 90 construct, many Fortran 90 compilers support it.

COMMON block. On the SGI Altix and Origin architectures, the local part of a co-array can be represented as a COMMON variable in each SPMD process image (as shown in Figure 1(d)) and mapped into remote images as symmetric data objects using SHMEM library primitives. References to local co-array data are expressed as references to COMMON block variables. This code shape is the most amenable to back-end compiler optimizations and results in the best performance for local computation on COMMON and SAVE co-array variables (see Section 4.1).

Subroutine parameter representation. To avoid pessimistic assumptions about aliasing, a *procedure splitting* technique can be used. If one or more COMMON block or SAVE co-arrays are accessed intensively within a procedure, the procedure can be split into wrapper and body procedures (see Figures 1(e) and 1(f)). The wrapper procedure passes all (non-EQUIVALENCEd) COMMON block and SAVE co-arrays used in the original subroutine to the body procedure as explicit-shape arguments[1]; within the body procedure, these variables are then referenced as routine arguments. This representation

[1] Fortran 90 argument passing styles are described in detail elsewhere [14].

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

(a) Remote element access

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

(b) General communication code

Fig. 2. General communication code generation.

enables `cafc` to pass bounds and contiguity information to the back-end compiler. The procedure splitting technique proved effective for both the HP Alpha Fortran compiler and the Intel Fortran compiler.

3.2 Code Generation for Remote Accesses

In CAF, communication events are expressed at the language level by using the bracket notation for co-dimensions to reference remote data. The CAF programming model is explicit enough that a user can perform communication optimizations such as vectorization or aggregation at the source level. To facilitate retargetability while enabling code to be tailored to a particular target system, `cafc` uses an abstract interface for instantiating one-sided communication operations. Currently, `cafc` does not vectorize communication and communication is placed adjacent to the statement in which a non-local reference appears.

Here we describe several candidate code shapes for co-array communication; these range from library-based platform-independent communication to several strategies for expressing fine-grain load/store communication on shared memory systems.

Communication generation for generic parallel architectures. To access data residing on a remote node, `cafc` generates ARMCI calls. Unless the statement causing communication is a simple copy, temporary storage is allocated to hold non-local data.

Consider the statement `a(:) = b(:)[p] + ...`, which reads co-array data for `b` from another process image. First, `cafc` allocates a temporary, `b_temp`, just prior to the statement to hold the value of `b(:)` from image `p`. `cafc` adds an ARMCI `GET` operation to retrieve the data from image `p`, rewrites the statement as `a(:) = b_temp(:) + ...` and inserts code to deallocate `b_temp` after the statement. For a statement containing a co-array write to a remote image, such as `c(:)[p] = ...`, `cafc` inserts allocation of a temporary `c_temp` prior to the statement. Then, `cafc` rewrites the statement to store its result in `c_temp`, adds an ARMCI `PUT` operation after the statement to perform the non-local write and inserts code to deallocate `c_temp`.

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

Fortran 90 pointers. With proper initialization, Fortran 90 pointers can be used to directly address non-local co-array data. The CAF runtime library provides the virtual address of a co-array on remote images; this is used to set up a Fortran 90 pointer for referencing the remote co-array. An example of this strategy is presented in Figure 3(a). The generated code accesses remote data by dereferencing a Fortran 90 pointer, for

which Fortran 90 compilers generate direct loads and stores. In Figure 3(a), the procedure `CafSetPtr` is called for every access; this adds significant overhead. Hoisting pointer initialization outside the loop as shown in Figure 3(b) can substantially improve performance. To perform this optimization automatically, `cafc` needs to determine that the process image number for a non-local co-array reference is loop invariant.

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

(a) Fortran 90 pointer to remote data

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

(b) Hoisted Fortran 90 pointer initialization

Fig. 3. Fortran 90 pointer access to remote data.

Vector of Fortran 90 pointers. An alternate representation that doesn't require pointer hoisting for good performance is to precompute a vector of remote pointers for all the process images per co-array. This strategy should work well for parallel systems of modest size. Currently, all shared memory architectures meet this requirement. In this case, the remote reference in the code example from Figure 2(a) would become:
`C(J) = ptrArrayA(p)%ptrA(J)`.

Cray pointers. We also explored a class of shared-memory code generation strategies based on the SHMEM library. After allocating shared memory with `shmalloc`, one can use `shmem_ptr` to initialize a Cray pointer to the remote data. This pointer can then be used to access the remote data. Figure 4(a) presents a translation of the code in Figure 2 using `shmem_ptr`. Without hoisting pointer initialization as shown in Figure 4(b), this code incurs a performance penalty similar to the code shown in Figure 3(a).

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

(a) Cray pointer to remote data

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

(b) Hoisted Cray pointer initialization

Fig. 4. Cray pointer access to remote data.

4 Experiments and Discussion

Currently, `cafc` generates code that uses Fortran 90 pointers for references to local co-array data. To access remote co-array elements, `cafc` can either generate ARMCI calls or initialize Fortran 90 pointers for fine-grain load/store communication. Initialization of pointers to remote co-array data occurs immediately prior to statements referencing non-local data; pointer initialization is not yet automatically hoisted out of loops. To evaluate the performance of alternate co-array representations and communication strategies, we hand-modified code generated by `cafc` or hand-coded them. For instance, to evaluate the efficiency of using SHMEM instead of ARMCI for communication, we hand-modified `cafc`-generated code to use `shmem_put`/`shmem_get` for both fine-grain and coarse-grain accesses.

We used two NUMA platforms for our experiments: an SGI Altix 3000[2] and an SGI Origin 2000[3]. We used the STREAM benchmark to determine the best co-array representation for local and remote accesses. To determine the highest-performing representation for fine-grain remote accesses we studied the Random Access and Spark98 benchmarks. To investigate the scalability of CAF codes with coarse-grain communication, we show results for the NPB benchmarks SP and MG.

4.1 STREAM

The STREAM [4] benchmark is a simple synthetic benchmark program that measures sustainable memory bandwidth in MB/s (10^6 bytes/s) and the corresponding computation rate for simple vector kernels. The top half of Figure 5 shows vector kernels for a Fortran 90 version of the benchmark. The size of each array should exceed the capacity of the last level of cache. The performance of compiled code for the STREAM benchmark also depends upon the quality of the code's instruction stream[4].

```
DO J=1, N
 C(J)=A(J)
END DO
```
(a) Copy

```
DO J=1, N
 B(J)=s*C(J)
END DO
```
(b) Scale

```
DO J=1, N
 C(J)=A(J)+B(J)
END DO
```
(c) Add

```
DO J=1, N
 A(J)=B(J)+s*C(J)
END DO
```
(d) Triad

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

```
DO J=1, N
 B(J)=s*C(J)[p]
END DO
```
(f) CAF Scale

```
DO J=1, N
 C(J)=A(J)[p]+B(J)[p]
END DO
```
(g) CAF Add

```
DO J=1, N
 A(J)=B(J)[p]+s*C(J)[p]
END DO
```
(h) CAF Triad

Fig. 5. The STREAM benchmark kernels (F90 & CAF).

We designed two CAF versions of the STREAM benchmark: one to evaluate the representations for local co-array accesses, and a second to evaluate the remote access code for both fine-grain accesses and bulk communication. Table 1 presents STREAM bandwidth measurements on the SGI Altix 3000 and the SGI Origin 2000 platforms.

Evaluation of local co-array access performance. To evaluate the performance of local co-array accesses, we adapted the STREAM benchmark by declaring `A`, `B` and `C` as co-arrays and keeping the kernels from the top half of Figure 5 intact. We used the Fortran 90 version of STREAM with the arrays A, B and C in a COMMON block as a baseline for comparison The results are shown in the local access part of the Table 1. The results for the COMMON block representation are the same as the results of the original Fortran 90. The Fortran 90 pointer representation without the "no-aliasing" compiler flag yields only 30% of the best performance for local access; it is not always possible to use no-aliasing flags because user programs might have aliasing unrelated to co-array usage. On both architectures, the results show that the most efficient representation for co-array local accesses is as COMMON block variables. This representation

[2] Altix 3000: 128 Itanium2 1.5GHz processors with 6MB L3 cache, and 128 GB RAM, running the Linux64 OS with the 2.4.21 kernel and the 8.0 Intel compilers

[3] Origin 2000: 16 MIPS R12000 processors with 8MB L2 cache and 10 GB RAM, running IRIX 6.5 and the MIPSpro Compilers version 7.3.1.3m

[4] On Altix, we use `-override_limits -O3 -tpp2 -fnoalias` for the Intel 8.0 compiler. On the Origin, we use `-64 -O3` for the MIPSpro compiler.

	SGI Altix 3000				SGI Origin 2000			
Program representation	Copy	Scale	Add	Triad	Copy	Scale	Add	Triad
Fortran, COMMON block arrays	3284	3144	3628	3802	334	293	353	335
Local access, F90 pointer, w/o no-aliasing flag	1009	929	1332	1345	323	276	311	299
Local access, F90 pointer	3327	3128	3612	3804	323	277	312	298
Local access, F90 pointer to structure	3209	3107	3629	3824	334	293	354	335
Local access, Cray pointer	3254	3061	3567	3716	334	293	354	335
Local access, split procedure	3322	3158	3611	3808	334	288	354	332
Local access, vector of F90 pointers	3277	3106	3616	3802	319	288	312	302
Remote access, general strategy	33	32	24	24	11	11	8	8
Remote access bulk, general strategy	2392	1328	1163	1177	273	115	99	98
Remote access, F90 pointer	44	44	34	35	10	10	7	7
Remote access bulk, F90 pointer	1980	2286	1997	2004	138	153	182	188
Remote access, hoisted F90 pointer	1979	2290	2004	2010	294	268	293	282
Remote access, shmem_get	104	102	77	77	72	70	57	56
Remote access, Cray pointer	71	69	60	60	26	26	19	19
Remote access bulk, Cray ptr	2313	2497	2078	2102	346	294	346	332
Remote access, hoisted Cray pointer, w/o no-aliasing flag	2310	2231	2059	2066	286	255	283	275
Remote access, hoisted Cray pointer	2349	2233	2057	2073	346	295	347	332
Remote access, vector of F90 pointers	2280	2498	2073	2105	316	291	306	280
Remote access, hybrid representation	2417	2579	2049	2062	350	295	347	333
Remote access, OpenMP	2397	2307	2033	2052	312	301	317	287

Table 1. Bandwidth for STREAM in MB/s on the SGI Altix 3000 and the SGI Origin 2000.

enables the most effective optimization by the back-end Fortran 90 compiler; however, it can be used only for COMMON and SAVE co-arrays; a different representation is necessary for allocatable co-arrays.

Evaluation of remote co-array access performance. We evaluated the performance of remote reads by modifying the STREAM kernels so that A,B,C are co-arrays, and the references on the right-hand side are all remote. The resulting code is shown in the bottom half of Figure 5. We also experimented with a bulk version, in which the kernel loops are written in Fortran 90 array section notation. The results presented in the Table 1 correspond to the following code generation options (for both fine-grain and bulk accesses): the library-based communication with temporary buffers using ARMCI calls, Fortran 90 pointers, Fortran 90 pointers with the initialization hoisted out of the kernel loops, library-based communication using SHMEM primitives, Cray pointers, Cray pointers with hoisted initialization without the no-aliasing flag, Cray pointers with hoisted initialization, and a vector of Fortran 90 pointers to remote data. The next result corresponds to a hybrid representation: using the COMMON block representation for co-array local accesses and Cray pointers for remote accesses. The last result corresponds to an OpenMP implementation of the STREAM benchmark coded in a similar style to the CAF versions; this is provided to compare the CAF versions against an established shared memory programming model.

The best performance for fine-grain remote accesses is achieved by the versions that use Cray pointers or Fortran 90 pointers to access remote data with the initialization of the pointers hoisted outside loops. This shows that hoisting initialization of pointers to remote data is imperative for both Fortran 90 pointers and Cray pointers. Using the vector of Fortran 90 pointers representation uses a simpler strategy to hoist pointer initialization that requires no analysis, yet achieves acceptable performance. Using a function call per each fine-grain access incurs a factor of 24 performance degradation on Altix and a factor of five on the Origin.

For bulk access, the versions that use Fortran 90 pointers or Cray pointers perform better for the kernels Scale, Add and Triad than the general version (1.5-2 times better on Altix and 2.5-3 times better on Origin), which uses buffers for non-local data. Copying into buffers degrades performance significantly for these kernels. For Copy, the general version does not use an intermediate buffer; instead, it uses `memcpy` to transfer the data directly into the `C` array and thus achieves high performance.

We implemented an OpenMP version of STREAM that performs similar remote data acesses. On Altix, the OpenMP version delivered performance similar to the CAF implementation for the Copy, Add, and Triad kernels, and 90% for the Scale kernel. On Origin, the OpenMP version achieved 86-90% of the performance of the CAF version.

In conclusion, for top performance on the Altix and Origin platforms, we need distinct representations for co-array local and remote accesses. For COMMON and SAVE variables, local co-array data should reside in COMMON blocks or be represented as subroutine dummy arguments; for remote accesses, `cafc` should generate communication code based on Cray pointers with hoisted initialization.

4.2 Random Access

To evaluate the quality of the CAF compiler generated code for applications that require fine-grain accesses, we selected the Random Access benchmark from the HPC Challenge benchmark suite [5], which measures the rate of random updates of memory, and implemented a CAF version.

The serial version of the benchmark declares a large main array `Table` of 64-bit words and a small substitution table `stable` to randomize values in the large array. The `Table` has the size `TableSize` = 2^n words. After initializing `Table`, the code performs a number of random updates on `Table` locations. The kernel of the serial benchmark is shown in Figure 6 (a).

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

(a) Sequential Random Access

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

(b) CAF Random Access

Fig. 6. Random Access Benchmark.

In the CAF implementation, the global table is a co-array. `TableSize` words reside on each image, so that the aggregate size is `GlobalTableSize = TableSize * NumberOfImages`. Each image has a private copy of the substitution table. All images concurrently generate random global indices and perform the update of the corresponding locations. No synchronization is used for concurrent updates (errors on up to 1% of the locations due to race conditions are acceptable). The kernel for all of our CAF variants of the benchmark is shown in Figure 6 (b).

A parallel MPI version [5] is available that uses a bucket sort algorithm to cache a number of remote updates locally. Compared to the CAF version, the bucket version

improves locality, increases communication granularity and decreases TLB misses for modest numbers of processors.

Our goal is to evaluate the quality of source-to-source translation for applications where fine-grain accesses are preferred due to the nature of the application. Previous studies have shown the difficulty of improving the granularity of fine-grain shared memory applications [15]. We use the Random Access benchmark as an analog of a complex fine-grain application. For this reason, we did not implement the bucket sorted version in CAF, but instead focused on the pure fine-grain version presented above.

The results of Random Access with different co-array representations and code generation strategies are presented in Table 2 for the SGI Origin 2000 architecture and in Table 3 for the SGI Altix 3000 architecture. The results are reported in MUPs, 10^6 updates per second, per processor for two main table sizes: 1MB and 256MB per image, simulating an application with modest memory requirements and an application with high memory requirements. All experiments were done on a power of two number of processors, so that we can replace `div`s and `mod`s with fast bit operations.

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

Table 2. Random Access performance on the Origin 2000 in MUPs per processor.

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

Table 3. Random Access performance on the Altix 3000 in MUPs per processor.

Each table presents results in MUPs per processor for seven variants of Random Access. *CAF vector of F90 ptrs.* uses a vector of Fortran 90 pointers to represent co-array data. *CAF F90 pointer* uses Fortran 90 pointers to directly access co-array data. *CAF Cray pointer* uses a vector of integers to store the addresses of co-array data. A Cray pointer is initialized in place to point to remote data and then used to perform an update. *CAF shmem* uses `shmem_put` and `shmem_get` functions called directly from Fortran. *CAF general comm.* uses the ARMCI functions to access co-array data. *MPI bucket 2048* implements a bucket sorted algorithm with a bucket size of 2048 words. *OpenMP* uses the same fine-grained algorithm as the CAF versions; it uses a private substitution table and performs first-touch initialization of the global table to improve memory locality.

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

4.3 Spark98

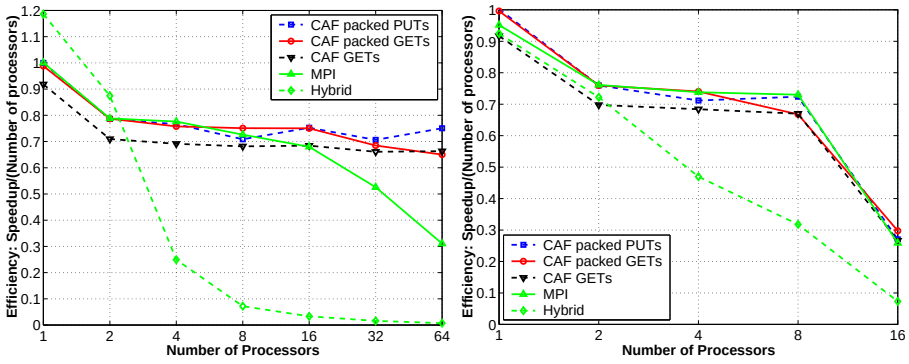
To evaluate the performance of more realistic fine-grain applications, we selected CMU's Spark98 [6] benchmark. This benchmark computes a sparse matrix-vector product of a symmetric matrix stored in compressed sparse row format, and is available in several versions: a sequential version, a highly tuned shared-memory threaded version (denoted as *hybrid* in [6]) and an MPI version. The original versions are written in C, we translated their computational kernels into Fortran 90 and derived a CAF version from the original MPI version.

All parallel versions of Spark98 use a sophisticated data partitioning which has been computed offline, to improve load balance between processors. The core of the benchmark computes a partial sparse matrix-vector product locally and then assembles the result across processors.

Our experimental results were collected on an Altix 3000 and an SGI Origin 2000. On the Altix 3000 architecture, we considered two different placements of a parallel job to the processors. The *single* placement corresponds to running one process per dual-processor node; in the *dual* placement two processes are run on both CPUs of a node, sharing the local memory bandwidth. To eliminate variations in local performance introduced by the backend Fortran or C compilers, the CAF and MPI versions use Fortran kernels for the local computation and result assembly. The threaded shared-memory version uses the original C kernels.

We evaluate three different CAF alternatives: the first (CAF packed PUTs) uses manual data packing and `PUTs` for communication, the second (CAF packed GETs) uses manual data packing and `GETs` for communication, the third version (CAF GETs) uses the Fortran 90 array section vector subscript notation to access remote data in place through a Cray pointer (during the assembly phase), but this notation is not currently handled automatically by our CAF compiler. We consider the third version to be written in a more natural style for CAF programs.

Figure 7 shows results for the Spark98 benchmark for the versions described previously for *dual* placement executions on the Altix 3000; similar results were observed for a *single* placement. The CAF and MPI versions have similar performance for a small number of processors (8-16). On the Altix 3000, for larger numbers of processors, the CAF versions outperform the MPI implementation. We observed that the time spent for the result assembly stage is 2.5 times higher on 32 processors and 5 times higher on 64 processors. While we do not know the implementation details of the proprietary MPI library, it appears that the single copy ARMCI data transfers are more efficient. In the hybrid version, a single thread allocates all data structures, thus reducing memory locality for the other threads resulting in poor load balance and non-scalable perfor-



(a) Altix 3000 (b) Origin 2000

Fig. 7. Comparison of parallel efficiencies (per iteration) for Spark98 (sf2 trace) for CAF, MPI and hybrid versions on an SGI Altix 3000 and an SGI Origin 2000.

mance. The CAF GETs version suffers up to a 13% performance penalty for the Altix 3000 and up to a 9% penalty on the Origin 2000 compared to the fastest CAF version (packed PUTs). This shows that this more natural programming style only has a small abstraction overhead.

4.4 NAS MG and SP

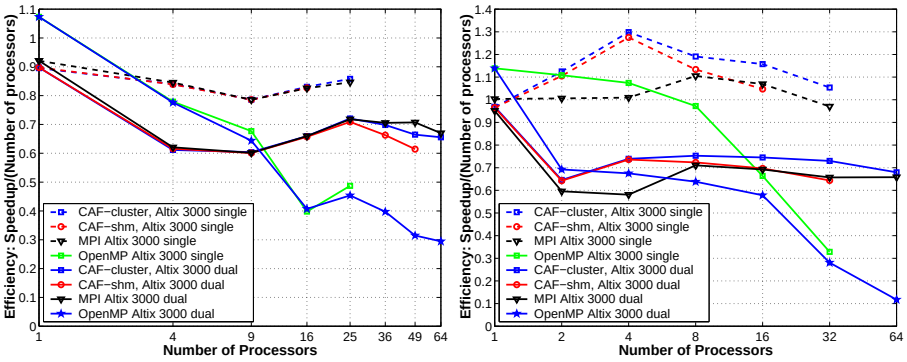
To evaluate our code generation strategy for hardware shared memory platforms for codes with coarse-grain communication, we selected two benchmarks, MG and SP, from the NAS Parallel Benchmarks [7, 16], widely used for evaluating parallel systems.

We compare four versions of the benchmarks: the standard 2.3 MPI implementation, two compiler-generated CAF versions based on the 2.3 distribution *CAF-cluster*, which uses the Fortran 90 pointer co-array representation and the ARMCI functions that rely on an architecture-optimized memory copy subroutine supplied by the vendor to perform data movement, and *CAF-shm*, which uses the Fortran 90 pointer co-array representation, but uses Fortran 90 pointers to access remote data, as well as the official 3.0 OpenMP [16] versions of SP and MG. The OpenMP version of SP incorporates structural changes made to the 3.0 serial version to improve cache performance on uniprocessor machines, such as fusing loops and reducing the storage size for temporaries; it also uses a 1D strategy for partitioning computation that is better suited for OpenMP.

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

For each benchmark, we present the parallel efficiency of the MPI, CAF and OpenMP implementations [5]. On an Altix, we evaluate these benchmarks for both the *single* and

[5] 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; P is the number of processors; $t_p(P,\rho)$ is the time for the parallel execution on P processors using parallelization ρ. Perfect speedup would yield efficiency 1.0 for each processor configuration.



(a) NAS SP (b) NAS MG

Fig. 8. Comparison of parallel efficiencies for NAS SP and NAS MG for MPI, CAF with general communication and CAF with shared memory communication, as well as OpenMP versions on an SGI Altix 3000.

dual processor configurations (see Section 4.3). The experimental results for problem size class C are shown on the figure 8. For SP, both CAF versions achieve similar performance—comparable to the standard MPI version. For MG, the CAF-cluster version performs better than the CAF-shm version. Since both versions use coarse-grain communication, the performance difference shows that the architecture-tuned memory copy subroutine performs better than the compiler scalarized data copy; it effectively hides the interconnect latency by keeping the optimal number of memory operations in flight. The CAF cluster version outperforms the MPI version for both the single and dual configurations. The results for the OpenMP versions are not directly comparable since they are based on the 3.0 source base, but they are known to be well designed and tuned for OpenMP execution. The OpenMP performance is good for a small number of processors (up to 8-9) but then tails off compared to the MPI and CAF versions.

5 Conclusions

We investigated several implementation strategies for efficiently representing, accessing and communicating distributed data in Fortran 90 source code generated by a CAF compiler for scalable shared memory systems. Generating fine-grain communication that uses direct loads and stores for the STREAM benchmark improved the performance by a factor of 24 on the Altix and a factor of five on the Origin. We found that for benchmarks requiring fine-grain communication, such as Random Access, a tailored code generation strategy that takes into account architecture and back-end compiler characteristics, provides better performance. In contrast, benchmarks requiring only coarse-grain communication deliver better performance by using an architecture's tuned `memcpy` routine for bulk data movement. Our current library-based code generation already enables us to achieve performance comparable to or better than that of hand-tuned MPI for benchmarks such as SP and MG, which use coarse-grain communication. The Spark98 experiments showed that programming in a natural CAF style by using remote data in place incurs an acceptable performance penalty compared to the fastest CAF version, which manages buffers explicitly.

Based on our study, we plan to develop suport for automatic shared memory code generation using the COMMON block representation for local co-array accesses and using a pointer-based representation for remote accesses in conjunction with pointer initialization hoisting. We will add support for automatic recognition of contiguous remote data transfers and implement them using calls to optimized system primitives. These strategies will enable `cafc` to generate code with high performance for both local and remote accesses on scalable shared-memory systems.

Acknowledgments

We thank J. Nieplocha and V. Tipparaju for their collaboration on ARMCI. We thank F. Zhao for her work on the compiler and K. Feind for his insights about the Altix.

References

1. Numrich, R.W., Reid, J.K.: Co-Array Fortran for parallel programming. ACM Fortran Forum **17** (1998) 1–31
2. Silicon Graphics, Inc.: The SGI Altix 3000 Global Shared-Memory Architecture. `http://www.sgi.com/servers/altix/whitepapers/tech_papers.html` (2004)
3. Laudon, J., Lenoski, D.: The SGI Origin: a ccNUMA highly scalable server. In: Proceedings of the 24th Intl. Symposium on Computer Architecture, ACM Press (1997) 241–251
4. McCalpin, J.D.: Sustainable Memory Bandwidth in Current High Performance Computers. Silicon Graphics, Inc., MountainView, CA. (1995)
5. HPC Challenge Developers: HPC Challenge Benchmark. `http://icl.cs.utk.edu/projectsdev/hpcc` (2003)
6. O'Hallaron, D.R.: Spark98: Sparse matrix kernels for shared memory and message passing systems. Technical Report CMU-CS-97-178, School of Computer Science, Carnegie Mellon University (1997)
7. 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)
8. Snir, M., Otto, S.W., Huss-Lederman, S., Walker, D.W., Dongarra, J.: MPI: The Complete Reference. MIT Press (1995)
9. Dagum, L., Menon, R.: OpenMP: An Industry-Standard API for Shared-Memory Programming. IEEE Comput. Sci. Eng. **5** (1998) 46–55
10. Silicon Graphics, Inc.: MPT Programmer's Guide, mpi man pages, intro_shmem man pages. `http://techpubs.sgi.com` (2002)
11. Nieplocha, J., Carpenter, B. In: ARMCI: A Portable Remote Memory Copy Library for Distributed Array Libraries and Compiler Run-Time Systems. Volume 1586 of Lecture Notes in Computer Science. Springer-Verlag (1999) 533–546
12. Coarfa, C., Dotsenko, Y., Eckhardt, J., Mellor-Crummey, J.: Co-array Fortran Performance and Potential: An NPB Experimental Study. Number 2958 in LNCS, Springer-Verlag (2003)
13. Dotsenko, Y., Coarfa, C., Mellor-Crummey, J.: A Multiplatform Co-Array Fortran Compiler. In: Proceedings of the 13th Intl. Conference of Parallel Architectures and Compilation Techniques, Antibes Juan-les-Pins, France (2004)
14. Adams, J.C., Brainerd, W.S., Martin, J.T., Smith, B.T., Wagener, J.L.: Fortran 90 Handbook: Complete ANSI/ISO Reference. McGraw Hill (1992)
15. Woo, S.C., Ohara, M., Torrie, E., Singh, J.P., Gupta, A.: The SPLASH-2 programs: Characterization and methodological considerations. In: Proceedings of the 22th International Symposium on Computer Architecture, Santa Margherita Ligure, Italy (1995) 24–36
16. Jin, H., Frumkin, M., Yan, J.: The OpenMP implementation of NAS parallel benchmarks and its performance. Technical Report NAS-99-011, NASA Ames Research Center (1999)