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. Source-to-source translation is an important code generation strategy commonly used by parallelizing compilers and compilers for parallel languages. In this paper, we investigate the performance impact of using different Fortran 90 representations for local and remote accesses on scalable shared-memory multiprocessors when generating SPMD code for Co-array Fortran. Our aim is to deliver the full power of the hardware platform to the application when operating on local data and when accessing remote data using either coarse-grain or fine-grain communication. We explored the performance impact of several different representations for shared data and several different ways of implementing communication. Using CAF variants of the STREAM, Random Access, and NAS MG & SP benchmarks, we compared the performance of library-based implementations of one-sided communication with fine-grain communication that uses pointers to access remote data using load and store operations. Our experiments showed that using pointer-based fine-grain communication improved performance as much as a factor of 24 on an SGI Altix and as much as a factor of five on an SGI Origin, when pointer initialization is hoisted out of a loop performing data accesses.

1 Introduction

To facilitate retargetability, parallelizing compilers and compilers for parallel programming languages primarily use a source-to-source strategy for code generation. This approach enables a separation of concerns: a parallel compiler can leave the details of back-end code optimization to a sequential node compiler (usually one for Fortran or C) and focus on managing parallelism, communication and synchronization. To achieve the highest possible performance when using source-to-source code generation, one must tailor a parallel compiler's generated code to the characteristics of the target platform.

* 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.

Parallel programs use either fine-grain or coarse-grain communication. Commodity interconnects for clusters are optimized for coarse-grain communication. To tailor communication for clusters, parallel compilers can transform fine-grain into coarse-grain communication through vectorization and other techniques. On platforms with hardware support for shared memory, the range of hardware solutions lends itself to a variety of different approaches. Today's microprocessor-based shared-memory multiprocessors, ranging from multiple processors on a shared bus to systems with scalable interconnects such as the SGI Altix [1, 2] and SGI Origin [3], are designed to support fine-grained communication at the cache-line level exclusively; coarse-grain communication is accomplished by moving a group of cache lines individually. In contrast, the Cray X1 [4] is a shared-memory architecture designed to stream data between the processors using vector operations. Clearly, achieving the top performance on such different systems will require different communication code.

In this paper we investigate the performance impact of using different Fortran 90 array representations and communication strategies on scalable shared-memory multiprocessors with non-uniform shared memory access (NUMA). Our study was motivated by our desire to build a retargetable compiler for Co-array Fortran (CAF) [5, 6]—a global address space model for single-program-multiple-data (SPMD) parallel programming that consists of a small set of extensions to Fortran 90—that generates fast Fortran 90 node code augmented with communication for such platforms. Our goal is to achieve *performance transparency*, namely, to deliver the full power of the hardware platform to the application. To achieve this goal, code generated by our CAF compiler must yield the best possible performance when operating on local data and when accessing remote data in a fashion suited for either coarse-grain or fine-grain communication.

When generating Fortran 90 node code from CAF, there are several candidate ways of representing and manipulating co-array data at the Fortran 90 language level. When co-array data is allocated at program launch time to ensure it will be shareable (e.g. in a shared-memory region), a pointer-based representation is necessary. We explore several choices for representing co-arrays and accessing local & remote co-array data. We evaluate the performance implications of these choices for several different codes including CAF variants of the STREAM [7], Random Access [8], and NAS MG & SP benchmarks [9].

For STREAM, our experiments show that instead of using function calls to access remote data elements, carefully generating fine-grain communication that uses pointers to access remote data using load and store can improve performance by more than a factor of 24 on the Altix and a factor of five on the Origin. These dramatic improvements are possible when initialization of pointers to remote data is hoisted out of loops performing remote data accesses. For Random Access, using load/store for fine-grain access to all shared data always improves performance, though not as much. NAS SP and MG use coarse-grain communication; our findings show that using memcpy to transfer data was more effective than compiler-generated code for copying data.

In the next section, we briefly review the Co-array Fortran language, shared-memory architectures and communication libraries. In section 3, we describe the alternative code shapes we investigate in this study. In section 4, we describe the benchmark codes and

our experimental results using different code shapes for accessing shared data. We summarize our findings in section 5.

2 Background

Co-Array Fortran (CAF) supports SPMD parallel programming through a small set of language extensions to Fortran 90. Like MPI programs, an executing CAF program consists of a static collection of asynchronous process images. CAF programs explicitly manage data locality and computation distribution; however, CAF is a global address space programming model. CAF supports distributed data using a natural extension to Fortran 90 syntax. For example, the declaration `integer :: x(n,m)[*]` declares a shared co-array with `n` $\times$ `m` integers local to each process image. The dimensions inside brackets are called co-dimensions. Co-arrays may also be declared for user-defined types as well as primitive types. A local section of a co-array may be a singleton instance of a type rather than an array of type instances. Instead of explicitly coding message exchanges to obtain data belonging to other processes, a CAF program can directly reference non-local values using an extension to Fortran 90 syntax for subscripted references. For instance, process `p` can read the first column of data in co-array `x` from process `p+1` with the right-hand side reference to `x(:,1)[p+1]`. The CAF language also includes synchronization primitives. Since both remote data access and synchronization are language primitives in CAF, they are amenable to compiler-based optimization. A more complete description of the CAF language can be found elsewhere [5, 6].

2.1 Shared Memory Architectures

Shared memory architectures provide an abstraction of globally addressable physical memory. Global address spaces significantly simplify programming, e.g., using OpenMP directives, compared to cluster-based architectures, where a communication library, such as MPI [10] or ARMCI [11], must be used to access remote data.

Hardware shared memory architectures are widely used for parallel computing because they are easy to program. There are two main classes of hardware shared memory architectures: machines with uniform memory access (UMA) times and machines with non-uniform memory access (NUMA) times. Both classes of architectures allow any processor in the system to address any word in the overall global memory. Common bus-based SMP systems fall into the UMA category, but the number of processors they can support is usually small (up to 16). Scalable UMA machines such as the Sun Fire Server series [12] use a sophisticated interconnect between processors and the memory subsystem.

Most scalable shared memory parallel architectures are NUMA machines, such as the Cray X1 [4],IBM pSeries 690 [13], the SGI Origin2000 [3] and the SGI Altix 3000 [2]. These machines are organized using a set of nodes; each node contains processing elements and memory. These nodes are connected using a low-latency, high-bandwidth interconnect. Each processor can access the local memory on its node with low latency. The memory on other nodes can also be accessed by every processor with a higher latency.

2.2 Shared Memory Access Library

The Shared Memory Access Library (SHMEM) [14], developed by Silicon Graphics, Inc., provides an 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 on all processes, such as common block or save variables and objects allocated from the symmetric heap [14]. The API contains routines for remote data transfer, for both contiguous and strided reads and writes, collective operations such as broadcast and reductions, barrier synchronization and atomic memory operations. One useful feature is the availability of remote pointers, which enables direct references to data objects owned by another process. The SHMEM memory allocation routines provide a NUMA-aware memory placement; the SHMEM startup processing routines ensure collocation between the processor on which a process runs and the memory associated with that process; such collocation is important for performance.

2.3 Aggregate Remote Memory Copy Interface

The CAF compiler we describe in this paper 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 [15]. The latest version of ARMCI performs NUMA-aware memory allocation on SGI ALtix and Origin platforms using the SHMEM library's `shmalloc` primitive.

3 Source-level Strategies for Implementing CAF on Shared Memory Architectures

Currently, the Rice CAF compiler (`cafc`) performs source-to-source transformation of CAF code into Fortran 90 code using two strategies: one uses an abstract communication interface, presently instantiated for ARMCI, the other uses loads and stores to access remote data. This strategy was designed to leverage the best back-end compiler available on the target platform to optimize local computation. `cafc` is implemented on top of OPEN64/SL [16], a version of the OPEN64 compiler infrastructure [17] that we modified to support source-to-source transformation of Fortran 90 and CAF.

A preliminary study [18] presented a general solution for co-array representation using Fortran 90 pointers and code generation for an Itanium2 cluster with a Myrinet2000 interconnect using ARMCI. Extending our compiler to support more parallel architectures revealed the shortcomings of that strategy in our quest to achieve the best performance across a wide range of architectures for a variety of codes. A motivation behind using Fortran 90 pointers was its portability across all Fortran 90 compilers. To use this representation, we need the capability to manipulate Fortran 90 array

descriptors outside the compiler runtime; the CHASM library [19] provides this service. Unfortunately, the Fortran 90 pointer representation does not deliver *performance portability*, even across cluster-based architectures. The explanation lies in differences between back-end compiler optimization of codes with Fortran 90 pointers.

In the same study, we described communication generation strategy that generates code using library-based communication primitives. Though this approach is well-suited to cluster architectures, it fails to fully exploit the capabilities of hardware shared-memory architectures. In contrast to cluster-based architectures, shared memory architectures provide the ability to access remote memory directly via load/store instructions, making fine-grain remote accesses much more efficient than those on cluster-based architectures. On such systems, Fortran 90 data structures can be used to access remote data directly, avoiding the overhead of calling the abstract communication interface functions.

In this paper we describe an extensive study in which we compare Fortran 90 representations of common block and save co-arrays to find the one that yields superior performance for both local computation and computation with remote references, as well as to find the code shape most amenable for optimization by back-end Fortran 90 compilers. An important conclusion is that no such 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 has the ability to automatically tailor code to a particular architecture and use whatever co-array representations and code generation are needed to deliver the best performance.

In this paper we report our findings for two NUMA SGI platforms: Altix 3000 and Origin 2000, and the corresponding vendor compilers: Intel Fortran Compiler and SGI MIPSPro Fortran compiler.

3.1 Representing Local Co-array Data for Efficient Local Computation

To achieve the best performance for CAF applications, it is imperative to optimize local computation with co-array references - *local co-array references*. Because we use a source-to-source translation into Fortran 90, this leads to the question of what is the best Fortran 90 representation of the co-array local data. There are two major factors affecting the decision: (i) how well a particular back-end Fortran 90 compiler optimizes the representation, (ii) hardware and operating system capabilities of the target architecture.

Most vendor compilers do an excellent job optimizing accesses to common block and save variables, but fall short optimizing the same computation expressed with Cray or Fortran 90 pointers. The main reason is the challenge of 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. During the compilation of a CAF program, the CAF compiler knows that common block and save co-arrays occupy non-overlapping regions of memory, but this information cannot be conveyed to the back-end compiler if the co-array local part is expressed as a pointer. Conservative assumptions about aliases cause back-end compilers to forgo optimizations such as software pipelining, software prefetching, and unroll-and-jam among others, that are

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

(a) F90 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
```

(f) common block representation

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

(d) 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
```

(e) parameter representation

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

crucial for efficient local computation. Some but not all Fortran 90 compilers have flags enabling the user to specify that pointers do not alias, ameliorating the effects of analysis imprecision. Besides the aliasing problem, the Fortran 90 pointer representation can increase register pressure and inhibit software prefetching. The shape of an Fortran 90 pointer is not known at the time of compilation, therefore, the bounds and strides are not constant and occupy extra registers, thus increasing register pressure. Also an Fortran 90 compiler has no knowledge whether the memory pointed to by a pointer is contiguous or strided, which complicates generation of software prefetch instructions. Alternative representations might cause inefficiencies as well, e.g., co-arrays passed as procedure arguments to avoid assumptions about aliasing do not have the aliasing problem, but require adding extra procedure calls.

The hardware and the operating system impose extra constraints on whether a particular co-array representation is feasible. For example, the common block representation of a co-array is not feasible if a common block cannot be mapped into multiple process images.

The following sections describe five Fortran 90 representations for the local part of the co-array variable `real a(10,20)[*]`.

Fortran 90 pointer. Figure 1(a) shows the original representation implemented in `cafc`. At program start-up, memory is allocated to hold `10*20 = 200` double precision numbers and the `ca%local` field is initialized to point to it.

This approach enabled us to achieve performance roughly equal to that of MPI on an Itanium2 cluster with the Myrinet2000 interconnect and the Intel Fortran compiler v7.0 as the back-end compiler [18]. Some other vendor compilers do not optimize the Fortran 90 pointer representation effectively. For instance, possible 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 compiler flag to indicate the lack of pointer aliasing forced us to seek an alternative co-array Fortran 90 representation. Similarly, on the SGI Origin 2000, the MIPSPro Fortran 90 compiler does not optimize Fortran 90 pointer references in an efficient way.

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 the constant array bounds and contiguity information to the back-end compiler.
Cray pointer. Figure 1(c) shows how a Cray pointer can be used to represent the local portion of co-array `a`. This representation has similar properties to the pointer-to-structure representation. Though the Cray pointer is not a standard Fortran 90 construct, many vendor Fortran 90 compilers support it.
Subroutine parameter representation. To avoid the pointer aliasing problem, the *procedure splitting* technique can be used. If a common block or save co-array is intensively accessed in a subroutine, the local part of the co-array can be converted to an array parameter to the subroutine. Figure 1(d) shows the original subroutine `foo`, while figure 1(e) shows how it is transformed into a pair of subroutines

This representation allows passing of bounds and contiguity information to the back-end compiler, but it requires to create a *subroutine-wrapper* for each original procedure. The subroutine-wrapper has the same name and the same argument list to provide for the separate compilation. The subroutine-wrapper calls the *subroutine-body* and passes all common block and save co-arrays used in the original subroutine `foo` as parameters using Fortran 77 calling conventions ([20]). Besides local parts for each co-array, either a co-array handle or a vector of pointers has to be passed, depending on the representation for remote parts (see section 3.2). The subroutine-body performs the same computation as the original subroutine `foo`, but local co-array references are replaced with references to the parameters. The procedure splitting technique proved effective for the HP Fortran compiler.
Common block. The two architectures addressed in this paper have the means to represent the co-array local part as a common block variable, as presented in figure 1(f). Memory is allocated by the linker. Local co-array accesses are expressed as references to a common block array variable `ca`. This code shape is the most amenable to back-end compiler optimizations and results in the best performance for local computation (see section 4.1). The SHMEM library described in section 2.2 allows accessing symmetric objects, such as common block and save variables, from other processes of a SPMD program.

3.2 Remote access code generation

Communication events are expressed in CAF at the language level, using the bracket notation. The compiler has the ability to generate communication code best suited to the target architecture. Currently, `cafc` generates communication calls in place, without moving or vectorizing communication. The programming model is powerful enough that a CAF user could express such communication optimizations as vectorization or aggregation at the source level. In order to achieve portability and to generate code best suited for particular parallel systems, we designed a generic one-sided communication interface; the communication code generated by the CAF compiler issues calls to this interface.

We explored several communication code generation strategies, ranging from the most general ones (but potentially suboptimal on shared memory architectures) to com-

munication tailored to exploit the efficient support for fine-grain communication present in shared memory machines.

Communication generation for generic parallel architectures. To access data that resides on a remote node, `cafc` generates calls ARMCI. In addition, temporary storage is allocated to hold off-processor data unless the statement is a simple copy.

For example, in the case of a read reference of a co-array on another image, `a(:) = b(:)[p] + ...`, a temporary, `b_temp`, is allocated just prior to the statement to hold the value of the array section `b(:)` from image `p`. Then, a call to get data from image `p` is issued to the runtime library. The statement is rewritten as `a(:) = b_temp(:) + ...`. The temporary is deallocated immediately after the statement. For a write to a remote image, such as `c(:)[p] = ...`, a temporary `c_temp` 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.

To evaluate the efficiency of using SHMEM instead of ARMCI for communication, we hand-modified our generated code to use `shmem_put`/`shmem_get` for both fine-grain and coarse-grain accesses.

Efficient communication generation for shared memory architectures. Library-based communication unnecessarily adds 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 co-array representations for performing efficient fine-grain accesses.

Fortran 90 pointers. The first representation of remote accesses we considered was using Fortran 90 pointers for remote co-array parts. The CAF runtime library provides the virtual address of a co-array on remote images; next, we set up a Fortran 90 pointer to point to the remote co-array. An example of this strategy is presented in figure 3(a). In the generated code accesses to remote data are performed by dereferencing the Fortran 90 pointers, for which vendor Fortran 90 compilers generate direct loads and stores. Note that the procedure `CafSetPointerAddress` is called for every access, this adds significant overhead.

Pointer initialization outside the loop can reduce overhead as shown in figure 3(b). To perform pointer hoisting automatically, the compiler needs to determine that all co-array accesses through a reference refer to a process image that is loop invariant.

Vector of Fortran 90 pointers. An alternative representation, that doesn't require the CAF compiler to perform pointer hoisting, is to precompute a vector of remote pointers for all the process images per co-array. In this case the code example from figure 2(a) would become

```
... initialization ...
DO J=1,N
  C(J)=ptrArrayA(p)%ptrA(J)
END DO
```

This strategy should work well for parallel systems of modes size. Currently, all shared memory architectures meet this requirement.

Cray pointers. We also explored a class of code generation strategies based on the SHMEM library. After allocating shared memory with `shmalloc`, one could use

shmem_ptr to get a Cray pointer to the remote data. This pointer could then be used to access the remote data. Figure 4(a) presents a translation of the code in figure 2 using shmem_ptr. This code generation choice incurs a performance penalty similar to the one incurred by using Fortran 90 pointers to remote data without hoisting their initialization. The obvious answer is to also hoist the initialization of Cray pointers using shmem_ptr outside the loop; an example is given in figure 4(b).

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

(a) Remote element access

```
DO J=1,N
  call CafGetScalar(A_handle, A(J), &
    p, CafTemp_A)
  C(J)=CafTemp_A
END DO
```

(b) General communication code

Fig. 2. General communication code generation.

```
DO J=1,N
  ptrA=>A(J)
  call CafSetPointerAddress(ptrA,&
    p, A_handle)
  C(J)=ptrA
END DO
```

(a) Fortran 90 pointer to remote data

```
ptrA=>A(1:N)
call CafSetPointerAddress(ptrA,&
  p,A_handle)
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.

```
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.

Currently our CAF compiler automatically generates Fortran 90 pointers for local co-array accesses. For remote accesses, the CAF compiler can generate either communication library calls or Fortran 90 pointers without hoisting their initialization. To evaluate the rest of the representation strategies we hand-modified the code generated by the CAF compiler or hand-coded them.

4 Experiments and Discussion

We used two NUMA platforms for our experiments. The first platform is an SGI Altix 3000, with 128 Itanium2 1.5GHz processors with 6MB L3 cache, and 128 GB RAM, running the Linux64 OS with the 2.4.21 kernel and the 8.0 Intel compilers. The second platform is an SGI Origin 2000, with 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. 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 have studied the Random Access benchmark. 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 [7] 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. We present the vector kernels in figure 5 for the Fortran 90 version of the benchmark. The memory bandwidth measured by the STREAM benchmark is reported in MB/sec. The size of each array should exceed the capacity of the last level of cache. The results of the STREAM benchmark depend also on the quality of the back-end compiler optimizations.

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

(a) Copy

```
DO J=1,N
  B(J)=scalar*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)+scalar*C(J)
END DO
```

(d) Triad

Fig. 5. The STREAM benchmark kernels.

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. The resulting bandwidth for the experiments performed on the SGI Altix 3000 and the SGI Origin 2000 platforms is presented in table 1.

Evaluation of local co-array accesses 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 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 We compare the performance of this strategy with the performance using each of the representations for co-array local accesses described in section 3.1 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. Both compilers used for experiments, Intel and MIPSpro, have flags enabling the user to specify that there is no aliasing between pointers; we used those flags in the experiments. On both architectures, the results show that the most efficient representation for co-array local accesses is as common block variables. This representation enables the most effective optimizations by the back-end Fortran 90 compiler. This representation, however, can be used only for common and save co-arrays; a different representation is necessary for allocatable co-arrays. Alternative representations that give good performance are the Fortran 90 pointer to structure and the Cray pointer. The vector of Fortran 90 point-

ers representation, while useful for remote references, performs slightly worse than the common block representation on the Origin.

Evaluation of remote co-array accesses 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 figure 6. 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 general strategy with temporary buffers using ARMCI calls, Fortran 90 pointers, Fortran 90 pointers with the initialization hoisted out of the kernel loops, general communication strategy using SHMEM primitives, Cray pointers, Cray pointers with hoisted initialization, vector of Fortran 90 pointers to remote data. The last result corresponds to a hybrid representation: using the common block representation for co-array local accesses and Cray pointers for remote accesses.

Several conclusions can be reached from these results. 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 of remote pointer initialization, for both Fortran 90 pointers and Cray pointers, is imperative. The communication code that uses a vector of Fortran 90 pointers achieves lower performance than the Cray pointers on the SGI Origin 2000, while performing similarly on the SGI Altix 3000; however, it doesn't require compiler techniques to hoist the pointer.

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, which uses buffers to store all the remote data. This is due to the fact that using local buffers increases the contention on the local memory system, while using the Fortran 90 pointer or Cray pointer representation the load is distributed between the local memory system and the interconnect. Note that in the case of the Cray pointer representation, the compiler can effectively scalarize the vectorized statements and achieve the best performance. For Copy, the general version does not use an intermediate buffer, instead it calls a communication function that performs a `memcpy` data transfer, thus enabling it to achieve high performance.

The results for local and remote accesses lead us to conclude that we will have to use distinct representations for co-array local and remote accesses to achieve the best performance on shared memory platforms. For common and save variables, local co-array parts should reside in common blocks, while for remote accesses the CAF compiler should generate code that uses Cray pointers with hoisted initialization. Until the compiler technology to perform hoisting of pointer initializations is in place, the code generation strategy with vectors of pointers can be used.

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 [8] that measures the rate of random updates of memory and adopted it for our needs.

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

```
DO J=1,N
  B(J)=scalar*C(J)[p]
END DO
```
(b) Scale

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

```
DO J=1, N
   A(J)=B(J)[p]+scalar*C(J)[p]
END DO
```
(d) Triad

Fig. 6. CAF STREAM benchmark kernels accessing remote data.

	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	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	2267	2495	2075	2101	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

Table 1. Bandwidth for STREAM in millions of bytes per second on the SGI Altix 3000 and the SGI Origin 2000 platforms

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`, a number of random updates into it are executed. The kernel of the serial benchmark follows:

```
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
```

We developed a CAF version of the Random Access benchmark. In the CAF implementation, the global table is a co-array. Thus, `TableSize` words reside on every image, so that the global table aggregate size is `GlobalTableSize = TableSize * NumberOfImages`. Each image has a private copy of the substitution table. The current implementation has the limitation that the number of images must be a power of

two. All images concurrently generate random global indices and perform the update of the corresponding locations on the corresponding images. No synchronization is used for concurrent updates (A few errors due to race conditions among processors updating the same memory location are acceptable.) Below is the kernel for all CAF variants of the benchmark:

```
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
```

A parallel version of the original benchmark may use bucket sort to cache a number of updates locally and use a message passing protocol between processors for handling remote updates. A parallel MPI version [8] is available that uses the bucket sort algorithm. The bucket sorting version effectively changes the benchmark properties compared to the version that uses a shared memory implementation with fine-grain communication. It increases the locality, hence, decreasing TLB misses for remote accesses; it also increases the granularity of communication.

Our goal is to evaluate the quality of source-to-source translation for applications where fine-grain accesses are unavoidable either due to the nature of the application or, perhaps, due to algorithmic or programming difficulties required to increase the granularity of communication. Previous studies show the difficulty of improving the granularity of fine-grain shared memory applications [21, 22]. In this study, we use the Random Access benchmark as an analog of such a complex 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, effectively 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; which enabled the expensive `div` and `mod` division operations to be replaced with fast integer 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
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 10^6 updates per second (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
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 10^6 updates per second (MUPs) per processor.

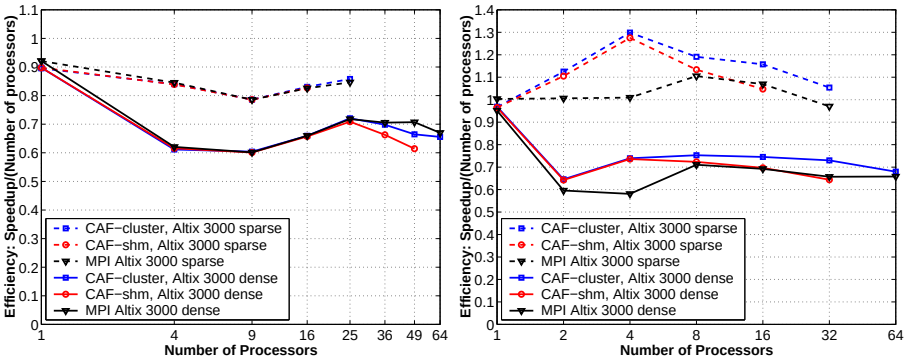
The table presents results in MUPs per processor for six variants of Random Access. (1) **CAF vect. of F90 ptrs.** is a CAF version with a vector of Fortran 90 pointers to represent co-array data. Remote co-array references to image `p` are done via the element `p` of the vector. (2) **CAF F90 pointer** uses Fortran 90 pointers to directly access co-array data. (3) **CAF Cray pointer** uses a vector of 64-bit 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. (4) **CAF shmem** uses `shmem_put` and `shmem_get` functions called directly from Fortran. (5) **CAF general comm.** uses the ARMCI functions to access co-array data. (6) **MPI bucket 2048** implements an MPI version of a bucket sorted algorithm with a bucket size of 2048 words.

The first five elements of each column show the results for the different co-array representations and resulting code shapes. The best co-array representations for fine-grain co-array accesses are the Cray pointer and the vector of Fortran 90 pointers throughout the entire spectrum of image numbers for both table sizes. Other code shapes, 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 for this particular benchmark. It is worth mentioning that the MPI implementation is much harder to code compared to a CAF version.

4.3 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 2.3 Parallel Benchmark suite. The NPB codes are widely regarded as useful for evaluating the performance of compilers on parallel systems.

We compare three versions of the benchmarks: the standard MPI implementation and two CAF versions with different strategies for the source-to-source translation supported by our CAF compiler. The first CAF version, called **CAF-cluster**, 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. The second, called **CAF-shm**, also uses the Fortran 90 pointer co-array representation, but uses Fortran 90 pointers to access remote data. Since all data transfers in these benchmarks are coarse-grain, expressed as co-array section assignments, we



(a) NAS SP (b) NAS MG

Fig. 7. Comparison of parallel efficiencies for NAS SP and NAS MG for MPI, CAF with general communication and CAF with shared memory communication versions on an SGI Altix 3000.

rely on the back-end Fortran 90 compiler to generate efficient copy code, by scalarazing the transformed copy statements. Sequential performance measurements used as a baseline were performed using the NPB 2.3-serial release.

We performed the experiments on the Altix 3000 architecture for two different placements of a parallel job to the processors. The **sparse** placement corresponds to running one process per dual-processor node; in the **dense** placement two images are run on both CPUs of a node, sharing the local memory bandwidth.

NAS SP As described in a NASA Ames technical report [9], the SP NAS benchmark is a simulated CFD applications that solves systems of equations resulting from an approximately factored implicit finite-difference discretization of three-dimensional Navier-Stokes equations, SP is a regular application and uses a skewed block distribution called multipartitioning [9, 23].

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 [9]. The communication is highly structured and goes through a fixed sequence of regular patterns.

For each benchmark, we present the parallel efficiency of the MPI and CAF implementations [1]. Using this metric, perfect speedup would yield efficiency 1.0 for each processor configuration. The experimental results are shown on the figure 7.

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-smp 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. The CAF cluster version outperforms the MPI version for both the sparse and dense configurations.

[1] We compute parallel efficiency as follows. For each parallel version ρ, the efficiency metric is computed as $\frac{\mathtt{t_s}}{\mathtt{P} \times \mathtt{t_p}(\mathtt{P}, \rho)}$. In this equation, $\mathtt{t_s}$ is the execution time of the original sequential version; $\mathtt{P}$ is the number of processors; $\mathtt{t_p}(\mathtt{P}, \rho)$ is the time for the parallel execution on $\mathtt{P}$ processors using parallelization ρ.

5 Conclusions

We investigated several implementation strategies for effi ciently representing distributed data in source code generated by a SPMD compiler for scalable shared memory systems. Our study addresses the effi cient source-level representation of accesses to local portions of distributed data, as well as accesses to portions of distributed data residing on other processors.

We experimented with different representations for local and remote accesses in the context of code generated by the Co-Array Fortran compiler under development at Rice. Generating fi ne-grain communication that uses direct loads and stores for the STREAM benchmark can improve the performance by a factor of 24 on the Altix and a factor of fi ve on the Origin. We found that for benchmarks requiring fi ne-grain communication, such as RandomAccess, a tailored code generation strategy, that takes into account architecture and back-end compiler characteristics, provides better performance. In contrast, benchmarks requiring coarse-grain communication only, can deliver better performance by using architecture-tuned data movement routines. Our current library-based code generation strategy already enables us to achieve performance comparable to or better than that of hand-tuned MPI parallel benchmarks with coarse-grain communication characteristics such as SP and MG.

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, 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. As this study indicates, this will enable `cafc` to generate code with high performance for both local and remote accesses.

Acknowledgments

We thank J. Nieplocha and V. Tipparaju for collaborating on the refi nement and tuning of ARMCI on shared memory platforms. We thank F. Zhao for her work on the Open64/SL Fortran front-end. We thank R. Numrich and A. Wallcraft for providing us with draft CAF versions of the MG and SP NAS parallel benchmarks. We thank Karl Feind for his insights into the performance issues of communication on the SGI Altix.

References

1. Neuner, S.: Scaling linux to new heights: the sgi altix 3000 system. Linux J. **2003** (2003) 3
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 annual international symposium on Computer architecture, ACM Press (1997) 241–251
4. Cray, Inc.: Cray X1 Server. `http://www.cray.com` (2004)

5. Numrich, R.W., Reid, J.K.: Co-Array Fortran for parallel programming. Technical Report RAL-TR-1998-060, Rutheford Appleton Laboratory (1998)
6. Numrich, R.W., Reid, J.K.: Co-Array Fortran for parallel programming. ACM Fortran Forum **17** (1998) 1–31
7. McCalpin, J.D.: Sustainable Memory Bandwidth in Current High Performance Computers. Silicon Graphics, Inc., MountainView, CA. (1995)
8. HPC Challenge Developers: HPC Challenge Benchmark. `http://icl.cs.utk.edu/projectsdev/hpcc` (2003)
9. Bailey, D., Harris, T., Saphir, W., van der Wijngaart, R., Woo, A., Yarrow, M.: The NAS parallel benchmarks 2.0. Technical Report NAS-95-020, NASA Ames Research Center (1995)
10. Snir, M., Otto, S.W., Huss-Lederman, S., Walker, D.W., Dongarra, J.: MPI: The Complete Reference. MIT Press (1995)
11. Nieplocha, J., Carpenter, B.: ARMCI: A portable remote memory copy library for distributed array libraries and compiler run-time systems. Lecture Notes in Computer Science **1586** (1999) 533–??
12. Sun Microsystems, Inc.: High-end Servers. `http://www.sun.com` (2004)
13. Matsubara, K., Teng, A.H.J., Blanchard, B.: IBM pseries 670 and pseries 690 system handbook. Technical Report SG24-7040-02, IBM (2003)
14. Silicon Graphics, Inc.: MPT Programmer's Guide, mpi man pages, intro_shmem man pages. `http://techpubs.sgi.com` (2002)
15. Nieplocha, J., Tipparaju, V., Saify, A., Panda, D.: Protocols and strategies for optimizing performance of remote memory operations on clusters. In: Proc. Workshop Communication Architecture for Clusters (CAC02) of IPDPS'02, Ft. Lauderdale, Florida (2002)
16. Open64/SL Developers: Open64/SL compiler and tools. `http://hipersoft.cs.rice.edu/open64` (2002)
17. Open64 Developers: Open64 compiler and tools. `http://sourceforge.net/projects/open64` (2001)
18. Coarfa, C., Dotsenko, Y., Eckhardt, J., Mellor-Crummey, J.: 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, Intel Corp. and the Portland Group, Inc., Springer-Verlag (2003) Published in 2004.
19. Rasmussen, C., Sottile, M., Bulatewicz, T.: CHASM language interoperability tools. `http://sourceforge.net/projects/chasm-interop` (2003)
20. 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)
21. Hu, Y.C., Cox, A., Zwaenepoel, W.: Improving fine-grained irregular shared-memory benchmarks by data reordering. In: Proceedings of the 2000 ACM/IEEE conference on Supercomputing (CDROM), IEEE Computer Society (2000) 33
22. 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
23. Naik, V.: A scalable implementation of the NAS parallel benchmark BT on distributed memory systems. IBM Systems Journal **34** (1995)