A Coarray Fortran implementation to support data-intensive application development

Citation: Eachempati, Deepak, Alan Richardson, Siddhartha Jana, Terrence Liao, Henri Calandra, and Barbara Chapman. "A Coarray Fortran Implementation to Support Data-Intensive Application Development." Cluster Comput 17, no. 2 (October 12, 2013): 569–583.

As Published: http://dx.doi.org/10.1007/s10586-013-0302-7

Publisher: Springer US

Persistent URL: http://hdl.handle.net/1721.1/104091

Version: Author's final manuscript: final author's manuscript post peer review, without publisher's formatting or copy editing

A Coarray Fortran Implementation to Support Data-Intensive Application Development

Deepak Eachempati*, Alan Richardson†, Siddhartha Jana*,
Terrence Liao‡, Henri Calandra§ and Barbara Chapman*
*Department of Computer Science, University of Houston, Houston, TX
Email: {dreachem,sidjana,chapman}@cs.uh.edu
†Department of Earth, Atmospheric, and Planetary Sciences (EAPS), MIT, Cambridge, MA
Email: alan_r@mit.edu
‡TOTAL E&P R&T USA, LLC, Houston, TX
Email: terrence.liao@total.com
§TOTAL E&P, Pau, France
Email: henri.calandra@total.com

***Abstract*—In this paper, we describe our experiences in implementing and applying Coarray Fortran (CAF) for the development of data-intensive applications in the domain of Oil and Gas exploration. The successful porting of reverse time migration (RTM), a data-intensive algorithm and one of the largest uses of computational resources in seismic exploration, is described, and results are presented demonstrating that the CAF implementation provides comparable performance to an equivalent MPI version. We then discuss further language extensions for supporting scalable parallel I/O operating on the massive data sets that are typical of applications used in seismic exploration.**

***Index Terms*—Parallel I/O; PGAS; Compilers; Language Design**

I. Introduction

New programming models are needed to meet the challenges posed by ever-increasing data processing requirements. They must provide mechanisms for accessing and processing large-scale data sets. Moreover, we believe they must be simple and intuitive to use, particularly for non-expert programmers. A large fraction of the codes that are used in the Oil and Gas industry are written in Fortran, and it is important that new programming models can work interoperably with these existing codes.

In the latest Fortran 2008 standard, new features were added which can potentially transform Fortran into an effective programming language for developing data-intensive, parallel applications [1]. These features, collectively, are based on Co-Array Fortran (CAF) [2], a Fortran extension proposed in the 1990s that incorporated a Partitioned Global Address Space (PGAS) programming model into the language. However, there has been an unfortunate lack of available implementations for these new language features, which therefore prompted us to develop our own implementation in the open-source OpenUH compiler.

Our initial evaluations revealed that additional support for handling large data sets is essential if coarrays are to be used on very large scale comptuations. As the size of such data sets, and the computational capacity of platforms, continues to grow there is an increasing gap between this and the rate at which data is input and output from and to disk. In order to accommodate it, we believe that there is a need for parallel I/O features in Fortran which complement the existing parallel-processing features provided by the coarray model. In addition to our work to identify suitable features, we are also exploring approaches that would allow Fortran applications to share data in memory in order to avoid the slowdown implied by disk storage.

This paper is organized as follows. In section II we provide an overview of the CAF features that were adopted in Fortran 2008. We then describe our implementation of these features in section III. We present our experience in porting an existing reverse time migration (RTM) application, widely used by Oil and Gas exploration teams for seismic images processing, to CAF in section IV. In section V, we discuss our thoughts on extending the language for enabling scalable, parallel I/O. We present related work in section VI, and close with our conclusions in section VII.

II. Coarray Fortran

The CAF extension in Fortran 2008 adds new features to the language for parallel programming. It follows the Single-Program, Multiple-Data (SPMD) programming model, where multiple copies of the program, termed *images* in CAF, execute asynchronously and have local data objects that are globally accessible. CAF may be implemented on top of shared memory systems and distributed memory systems. The programmer may access memory of any remote image without the explicit involvement of that image, as if the data is "distributed" across all the images and there is a single, global address space (hence, it is often described as a language implementation of the PGAS model). The array syntax of Fortran is extended with a trailing subscript notation in square brackets, called cosubscripts. A list of cosubscripts is used when the programmer intends to access the address space of another image, where an image index may range from 1 to the total number of images that are executing.

Figure 1 shows the logically shared but partitioned memory view that characterizes PGAS programming models. Specifying *A_Coarray(n)* without cosubscripts (square brackets) accesses only the local coarray. Since the remote memory access is explicit, it provides a clearly visible marker for potentially expensive communication operations in the code. This differentiates PGAS implementations like CAF from shared memory models which do not distinguish between local and remote data.

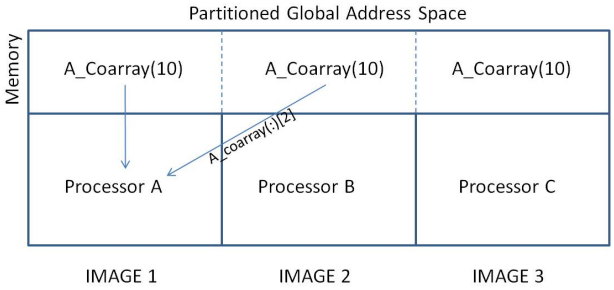


Fig. 1: Logical view of memory in CAF

Only objects declared as coarrays (i.e. one or more codimensions) can be accessed from a different image. Coarrays can be global/static or dynamically allocated, but in any case they must exist on all images. This has many consequences described in the standard, among them being that the allocation of an allocatable coarray is a collective operation with an implicit barrier. Coarrays may be declared with multiple codimensions, in which case the number of images are logically organized into a multi-dimensional grid. For example, the declaration *real :: c(2,3)[2,3:4,*]* logically arranges the images into a $2 \times 2 \times n$ grid for all cosubscripted references to the coarray *c*. A cosubscripted coarray reference generally indicates a remote memory access. For example, the statement *b(5:6)[2] = a(3:4)* writes to the 5th and 6th element of coarray *b* on image 2. Similarly, the statement *a(1:2) = b(5:6)[2]* reads from the 5th and 6th element of coarray *b* on image 2.

CAF provides both a global barrier synchronization statement (*sync all*) and a partial barrier synchronization statement (*sync images*) which may be used to to synchronize with a specified list of images. Critical sections, locks, and atomic operations are also part of the language. Additionally, CAF includes several intrinsic functions for image inquiry such as returning the image index of the executing process (*this_image*), the total number of running images (*num_images*), and the image index holding a coarray with specified cosubscripts (*image_index*).

III. CAF Implementation

We have implemented support for coarrays in OpenUH [3][4], an open-source research compiler based on the Open64 compiler suite. CAF support in OpenUH comprises three areas: (1) an extended front-end that accepts the coarray syntax and related intrinsic functions/subroutines, (2) back-end translation, optimization, and code generation, and (3) a portable runtime library that can be deployed on a variety of HPC platforms. Additionally, we have implemented a number of (currently) non-standard features that would be beneficial in developing more scalable parallel codes.

A. OpenUH Compiler

We have used OpenUH to support a range of research activities in the area of programming model research. OpenUH, depicted in Fig. 2, includes support for C/C++ and Fortran, inter-procedural analysis, a comprehensive set of state-of-the-art loop nest optimizations, an SSA-based global scalar optimizer, and code generators for a variety of target platforms. OpenUH can emit source code from an optimized intermediate representation and also selectively instrument the lowered intermediate code for low-overhead performance profiling.

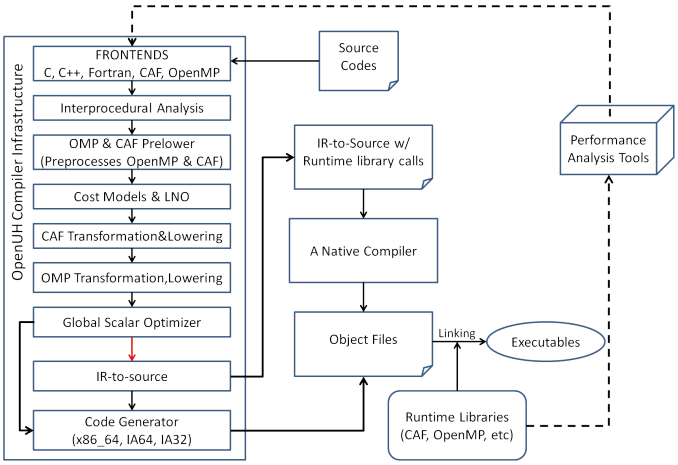


Fig. 2: The OpenUH Compiler Infrastructure

B. CAF Implementation Method in OpenUH

Front-end: Historically, the Fortran front-end of the OpenUH compiler derives from an older version of the Cray Fortran compiler. We modified this front-end to support our coarray implementation. Cray had provided some support for the extended CAF syntax. It could parse the `[]` syntax for recognizing coarray declarations and coindexed variables, it handled certain CAF intrinsics, and it translated the CAF language features to SHMEM-based runtime calls for targeting distributed systems with global address spaces. In order to take advantage of the analysis and optimizing capabilities in the OpenUH back-end, however, we needed to preserve the coarray representation into the back-end. To accomplish this, we adopted a similar approach to that used in Open64/SL-based CAF compiler [5]: the front-end emits an intermediate representation which retains the cosubscripts as extra array subscripts. Furthermore, we extended our front-end to support the set of new intrinsics subroutines/functions defined in Fortran 2008 for the CAF model, added support for dynamically allocating coarrays (where the lower and upper bounds for all dimensions and codimensions are specified at runtime), and we extended our compiler's internal representation for allocatable pointers/arrays to support coarrays.

Back-end: In the back-end, we first added a prelowering phase which normalizes the intermediate code (represented as an *abstract syntax tree (AST)* and emitted from the front-end) to facilitate dependence analysis. Next, we added a coarray lowering phase in which the compiler will generate communication based on remote coarray references. This step currently occurs prior to the Fortran 90 (F90) lowering phase, during which elemental array operations and array section references are translated into loops. We make use of the higher-level F90 array operations, supported by our compiler's very high-level internal representation, for generating bulk communication. AST nodes for representing single-element array references and array section references are processed to determine if they include cosubscripts, in which case it is a coindexed reference. A coindexed coarray variable signifies a remote access. For certain remote accesses that occur in the code, a compiler-generated temporary *local communication buffer* (LCB) is allocated for writing to or reading from the remotely accessed elements, functioning as a local source or destination buffer, respectively. In cases where a program variable can be used directly as the source or destination buffer, the compiler may not use an LCB to avoid the additional buffer space and memory copy that it would entail. After coarrays are lowered in this fashion, their corresponding *type* in the compiler's symbol tables are adjusted so that they only contain the local array dimensions.

Suppose the Coarray Lowering phase encounters the following statement:

$$\begin{aligned} A(i,j,1:n)[q] =& B(1,j,1:n)[p] + C(1,j,1:n)[p] \\ &+ D[p] \end{aligned} \quad (1)$$

This means that array sections from coarrays B and C and the coarray scalar D are brought in from image p. They are added together, following the normal rules for array addition under Fortran 90. Then, the resulting array is written to an array section of coarray A on image q. To save all the intermediate values used for communication, temporary buffers are made available. Our translation creates 4 buffers – t1, t2, t3, and t4 – for the above statement. We can represent transformation carried out by the compiler as:

$$\begin{aligned} A(i,j,1:n)[q] \leftarrow t1 =& t2 \leftarrow B(1,j,1:n)[p] \\ &+ t3 \leftarrow C(1,j,1:n)[p] \\ &+ t4 \leftarrow D[p] \end{aligned} \quad (2)$$

For each expression of the form $t \leftarrow R(...)[...]$, the compiler generates an allocation for a *local communication buffer* (LCB) t of the same size as the array section $R(...)$. The compiler then generates a `get` runtime call. This call will retrieve the data into the buffer t using an underlying communication subsystem. The final step is for the compiler to generate a deallocation for buffer t. An expression of the form $L(...)[...] \leftarrow t$ follows a similar pattern, except the compiler generates a `put` runtime call.

```
pseudocode:
    get( t2, B(1, j, 1:n), p )
    get( t3, C(i, j, 1:n), p )
    get( t4, D, p )
    t1 = t2 + t3 + t4
    put( t1, A(i, j, 1:n), q )
```

The above pseudo-code depicts the result of the initial lowering phase for the statement representation given in (2).

One of the key benefits of the CAF programming model is that, being language-based and syntactically explicit in terms of communication and synchronization, programs are amenable to aggressive compiler optimizations. This includes hoisting potentially expensive coarray accesses out of loops, message vectorization where the Fortran 90 array section syntax is not specified by the programmer, and generating non-blocking communication calls where it is feasible and profitable. For instance, a subsequent compiler phases may then convert `get` and `put` calls to non-blocking communication calls and use data flow analysis to overlap communication with computation and potentially aggregate messages, similar to work described in [6].

Runtime Support: The implementation of our supporting runtime system relies on an underlying communication subsystem provided by GASNet [7] or ARMCI [8]. This work entails memory management for coarray data, communication facilities provided by the runtime, and support for synchronizations specified in the CAF language. We have also added support for reduction operations in the runtime, as well as point-to-point synchronization using event variables. It is expected that both these features will be included in the next revision of the Fortran standard.

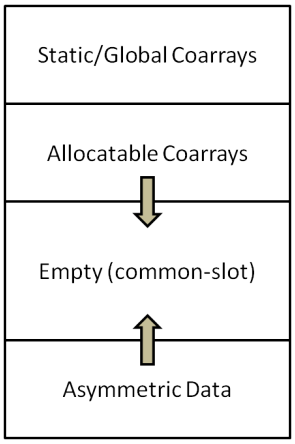


Fig. 3: Remotely-Accessible Memory Segment

The communication layer reserves a segment of remotely-accessible memory – registered and pinned-down for direct memory access by the network – for storing coarrays and other data that should be accessible across images. The structure of this memory segment is shown in Figure 3. Coarrays declared with static array bounds are reserved space at the top of the segment. The rest of the segment serves as a

remote memory access heap, reserved for dynamic memory allocations. Allocatable coarrays, which must exist with the same bounds on every image, are dynamically allocated and deallocated during the run of the program in a symmetric fashion across all images. Data objects which need not exist symmetrically across all images, but nevertheless should be remotely accessible, are allocated from the bottom of the heap.

C. Additional Supported Features For Scalability

In addition to the work we have done to comply with the current Fortran 2008 CAF model, we have added support for some currently non-standard language features. These comprise support for a set of collective operations, events, and compiler directives for controlling overlap between communication and computation. These features enable the programmer to more easily write scalable CAF codes, while providing facilities to avoid synchronization and communication latency costs. The Fortran standard committee is expected to formally accept language support for an expanded set of collectives and events in a soon to be released *Technical Specification* (TS) detailing new parallel processing extensions.

a) Collectives: Support for collective operations that can operate on distributed data in a scalable fashion is essential. The current Fortran 2008 standard is missing a specification of collective subroutines. We decided to adopt a set of coarray collectives that were available in the Cray Fortran compiler. Currently, we support a number of reductions: `co_sum`, `co_product`, `co_maxval`, and `co_minval`. The reductions take two arguments: the first is a source coarray, and the second is a destination array with the same rank and extents. The compiler will check that the source and destination arrays match. We also support broadcasts using the `co_bcast` intrinsic subroutine. The broadcast takes two arguments: the first is a coarray, and the second is the source image. It will broadcast the values in the coarray at the source image to the corresponding coarray in the rest of the images.

b) Event Variables: Events provide a much more flexible means for synchronizing two images compared to the `sync images` statement. An event variable is a coarray of type `event_type`, which is an intrinsic type defined in `iso_fortran_env`. An event may be posted, queried, and waited on using the `event post`, `event query`, and `event wait` statements, respectively. For example, to post an event `ev` on image `i`, one can use the statement `event post(ev[i])`. This is a non-blocking action, meaning it will return without waiting for the event to be "consumed" by image `i`. Image `i` can then block until the event `ev` has been posted with `event wait(ev)`, or it can query if the event has been posted with `event query(ev, state)` where state is a `logical` variable with value `.true.` or `.false.`.

c) Compiler Directives for Communication-Computation Overlap: CAF does not provide a language mechanism to explicitly control the use of non-blocking communication, as for instance is available with MPI using `mpi_isend` and `mpi_irecv`. This puts responsibility on the compiler to automatically make use of non-blocking `get` and `put` communication while preserving correctness using data flow analysis. While these optimizations are still in development, we have in place a directive-based mechanism to give the programmer some measure of control in affecting communication-computation overlap. Here is a short example illustrating its usage:

```
!dir$ caf_nowait(h1)
y(:) = x(:)[p]
...
!dir$ caf_wait(h1)
z(:) = y(:) * j
```

In the above example, *h1* is a declared variable of an intrinsic type, implemented internally as a pointer, that acts as a handle for representing a single outstanding non-blocking `get` operation. The `caf_nowait` directive says that the following statement should initiate a non-blocking read but the compiler should not generate a wait instruction for guaranteeing completion. Completion of the operation will only be ensured if a subsequent synchronization statement is encountered or a `caf_wait` directive with the specified handle is encountered. We only support the use of `caf_nowait` directive for assignment statements with a co-indexed variable as the only term on the right hand side. This is sufficient to allow the user to specify and control the completion of non-blocking read operations.

IV. Application

In this section, the application of Coarray Fortran to implement an algorithm of relevance to the Oil and Gas industry will be described. The code that was developed performs Reverse Time Migration (RTM) [9]. This method is widely used in subsurface exploration via seismic imaging. A source emits a wave pulse, which reflects off of subsurface structures and is recorded by an array of receivers. RTM transforms the recorded data into an image of the reflector locations. It is suitable for parallelization by domain decomposition, and is often executed on distributed memory systems due to the large volumes of data involved.

RTM uses the finite difference method to numerically solve a wave equation, propagating waves on a discretized domain. It consists of two stages. The first is referred to as the forward stage as it propagates an approximation of the source wave forward in time. In the second, backward stage, the recorded data is propagated backward in time. The RTM algorithm assumes that the forward propagated source wavefield and backward propagated data wavefield overlap in space and time at the location of reflectors. This imaging condition is evaluated during the backward pass. As it involves the wavefield from the forward pass, a technique must be employed to allow this wavefield to be recovered at each step during the backward stage. This could be achieved by saving the entire forward wavefield at each time step, and then reloading the appropriate data when it is needed. More sophisticated methods, such as only saving the boundaries of the forward wavefield [10],

reduce the memory requirement at the expense of additional computations by recomputing the forward wavefield during the backward pass.

Typical seismic experiments involve several thousand shots. Each shot consists of a source wave pulse, usually at a distinct location, and the associated receiver data. A single shot in a production survey typically contains several gigabytes of data. The total acquisition data size is usually in the terabyte range. For example, one dataset acquired from 30 blocks (3×3 square miles per block) consists of 82K shots, 460M traces with 3600 samples per trace with total data size of about 6.6 TB. Shots are processed individually, but potentially in parallel, each requiring tens of thousands of time steps of a 3D eighth-order finite difference propagator on a 3D domain containing billions of cells. Depending on the storage strategy used, significant volumes of data may need to be saved during each time step of the forward pass and recovered during the backward pass to enable the application of the imaging condition. RTM processing is therefore among the most data intensive applications used in industry.

The simplest implementations of the RTM algorithm make the assumption that the Earth is isotropic, this means that wave propagation speed is not dependent on angle. In reality this is often not a valid assumption as rocks such as shale, a common material at the depths of interest to hydrocarbon exploration, can exhibit strong anisotropy. Since hydrocarbon exploration frequently occurs in regions covered by sedimentary rocks, which were laid down in layers, a more plausible assumption is that of transverse isotropy, where waves can propagate at different speeds parallel and perpendicular to bedding planes. To account for buckling, uplift, and other processes which can change the orientation of the rock layers, tilted transverse isotropy (TTI) [11] may be assumed. The TTI implementation of RTM is considerably more compute and communication intensive than the isotropic version.

A. Implementation

A full RTM code was developed using Coarray Fortran, capable of processing production data. Both isotropic and TTI propagators were included. A section of the image produced by the code of the 2004 BP model [12], is displayed in Fig. 4. This is a synthetic model designed to be representative of challenging sub-salt environments currently being encountered in exploration of the Gulf of Mexico and off-shore West Africa. It consists of more than 1000 source locations spread over 67 km, on a grid discretized into 12.5m $\times$ 6.25m cells.

It was possible to design the code in such a way that only the communication modules were specific to a coarray implementation of the algorithm. This permitted an equivalent MPI version to also be created for performance comparison by merely writing MPI communication modules.

At each time step in the code, wavefield propagation is carried out in parallel by all images, utilizing halo exchanges to communicate face (and, in the case of TTI model, edge) data amongst neighbors. For the forward stage of the isotropic model, this entails a total of six halo exchanges for one wavefield, while for the TTI model there are eighteen halo exchanges of two wavefields. In the backward stage, if boundary swapping is being employed, then the total number of halo exchanges for each processor is doubled for the isotropic and TTI models. In both the CAF and MPI implementations, the halos from all of the wavefields to be transmitted to a particular processor were packed into a single buffer and then unpacked at the destination. This reduces the number of messages ultimately sent over the network, potentially improving communication performance. A further advantage of this approach is that the wavefields themselves do not need to be declared as coarrays in the CAF implementation, only the buffers, simplifying the creation of an MPI version free of coarrays.

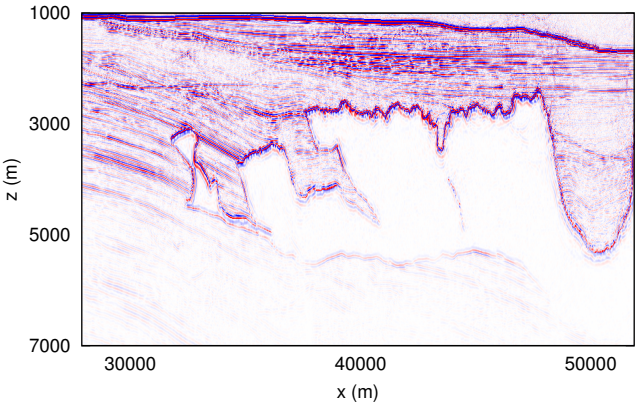


Fig. 4: A section of the 2004 BP model image generated by RTM using CAF.

As CAF currently lacks parallel I/O, for this RTM implementation we utilized conventional sequential I/O and performed a post-processing step to merge the output. Two different inputs are required for RTM. One is the earth model. This consists of a model of the wave speed in the region of the earth being imaged, and may also contain models of other parameters. Each of these parameter files is a single large file, containing a 3D block of floating point numbers. Each process only needs access to the part of the model that corresponds to the part of the domain that it has been assigned. The models only need to be read once by each process, and are then stored in memory for the duration of the execution. This was implemented as each process calling a function to read the relevant parts of the files by looping over two dimensions, performing contiguous reads in the third, fast, dimension. Another input that is required is the data recorded by receivers. This is the data containing recordings of waves that reflected in the subsurface and returned to the surface. RTM uses this data to try to construct an image of where the reflections took place. Processes only need access to data recorded by receivers located within their local domain. Typically a single file contains the data needed by all of the processes. This was again implemented by having each process repeatedly call a function that performs a contiguous read. This also only needs

to be done once by each process. The time required for reading input data is typically quite small compared to the overall runtime, as once it is loaded from disk into memory it is used in a large number of computations.

The output from RTM is the image of the reflectors in the earth. Each process produces an image for its part of the domain. In this implementation each process writes its image as a separate file, with a header file describing which part of the domain the data corresponds to. The RTM code is executed many times, once for each shot. A large number of ouput files are therefore created. Post-processing is performed to load all of these, and combine them into a single image file. In a production environment this may require loading hundreds of thousands of files, potentially straining the filesystem. Parallel I/O would have been advantageous for this situation, as it would enable the files from all of the processes working on a single shot to be joined before being written to disk, so the total number of files would be divided by the number of processes working on each shot.

B. Performance

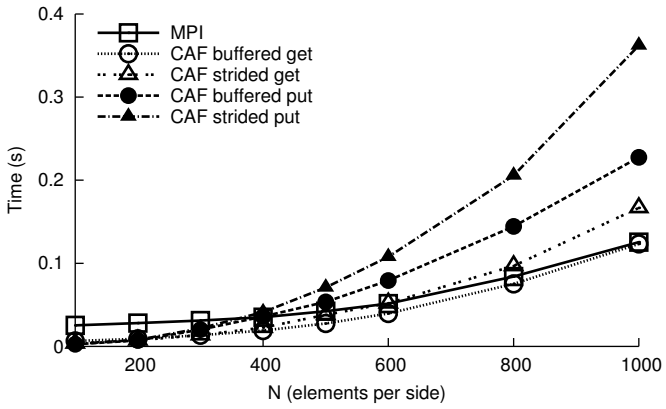
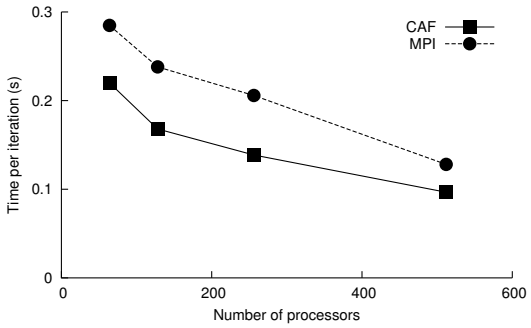
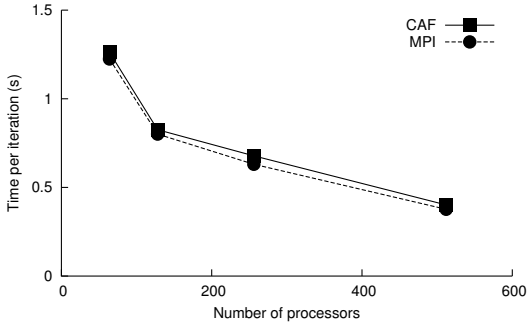


Fig. 5: Neighbor Communication Time for MPI, CAF GET, and CAF PUT (strided and buffered)

In Fig. 5, we depict communication time between neighbors. We compare communication using MPI send and receive (where non-contiguous data is packed and unpacked as necessary into contiguous buffers), and using 1-sided `get` and `put` via co-indexed accesses. For the latter, we considered (1) the "buffered" approach where the program packs all the data to be communicated into a contiguous buffer, and (2) the "strided" approach where the non-contiguous data to be communicated is represented directly using Fortran array-section notation and no additional buffer copy is used by the programmer (or generated by the compiler). All the CAF variants for communication worked well for smaller message sizes (less than 400 elements per side). For larger message sizes, we see the CAF `get` variants, and particularly the buffered version, delivering performance similar to or exceeding the MPI communication. Similar results reflecting the advantage CAF has for smaller message sizes over MPI have been reported for other applications [13][14][15]. The



(a) Isotropic



(b) TTI

Fig. 6: Runtime (communication and computation) of the forward stage of CAF and MPI versions of the code, for (a) isotropic and (b) TTI, as the number of processors was increased with a fixed total problem size.

performance benefit shown here in using contiguous buffers for CAF communication motivated our decision to employ such buffers for implementing halo exchanges in the RTM code.

The plots in Fig. 6 show the runtime per time step of the CAF and MPI versions during the forward stage of RTM. For the MPI version we used the Intel Fortran compiler, and for the CAF version we used the OpenUH compiler with comparable optimization options. A fixed domain size consisting of $1024 \times 768 \times 512$ cells was used, on a cluster with Intel Nehalem CPUs and QDR Infiniband interconnect. These measurements appear to show good performance of the CAF code: it was up to 32% faster for the isotropic case, while approximately equal to the MPI code for TTI. Recall that for each time step, each image performs halo exchanges of its wavefield data with its neighbors. For the forward stage, the TTI model requires eighteen halo exchanges for two wavefields, while the isotropic model requires only six halo exchanges for one wavefield. As these performance results was obtained prior to the availability of directives for hiding communication latency (described in Section III-C), the benefits observed for CAF in the isotropic case are offset by the significantly higher communication latency costs of the halo exchanges in the TTI case. Nevertheless, we believe this demonstrates that a production-level industrial code can be developed using CAF, with performance similar to that of

an MPI implementation being achievable while retaining the benefits of a compact, clean syntax that is more accessible to the non-expert programmer.

V. Parallel I/O in CAF

Although CAF currently lacks support for parallel I/O, for the RTM implementation described in the preceding section we were able to utilize traditional sequential I/O. Each processor wrote its output to a unique file and post-processing was performed to obtain the final resulting image by combining these outputs. To load the relevant portion of the input data on each processor, strided reading was employed. For the number of processors tested, combined with the use of a Lustre parallel file system, this was not found to create a bottleneck. Thus, it was possible to develop a parallel RTM code using CAF, despite the latter's lack of support for parallel I/O. It was suggested, however, that the file system may be overloaded by this approach on larger systems used in production environments. More generally, we believe that in the petascale/exascale era, I/O will increasingly become the primary performance bottleneck for data-intensive applications. Therefore, a sound parallel I/O mechanism is a crucial aspect for future parallel programming models and implementations.

A. Design Issues for Parallel I/O

We describe several design points for parallel I/O that we consider especially relevant.

1) **File Handlers: Individual versus Shared** While working on a file, multiple images may use seperate file handlers, one per image, to keep track of individual progress of read/write operations within a file. In this case, each image remains unaware of the progress of other images. However, the drawback of this approach is that additional communication needs to be performed to obtain the position of the shared file handler within the file. An alternative approach can be to read/write the same file using a handler which is shared by all the images. In this case, since the progress of the read/write operations is indicated by a shared resource, each image has to communicate with other images to determine the position of the handler. In a two-way communication model, such small communications can prove to be extremely inefficient. A more favorable approach would be to use *direct-access* file access operations which would enable every image to explicitly specify an offset within the file by specifying a record number. As a result, there does not arise a need to include shared file pointer operations for parallel I/O for CAF.
2) **Concurrent I/O: Independent versus Collective** One of the primary reasons for performance degradation of file I/O operations is the high overhead caused due to the seek latency involved with multiple reads/writes. This worsens when the read/write requests arrive at the file system nodes in a discrete unordered sequence. Since such requests come from images that are unaware of the progress of other images, the disk header might have to repeatedly move back and forth through the same sectors of the disk to service the requests. With increase in the number of images, the performance takes a clear toll. An alternative approach is for all the images to aggregate their requests such that the range of the new request spans over contiguous sections on the disk, thereby dramatically increasing the performance. This can be achieved by collective operations and has been explored heavily over the past few years. A consideration while using such a collective operation is that faster images would wait to synchronize with the slower images so that the request for file operations can be collectively put forth to the file system. However, in general the benefits of aggregated requests have been shown to supercede the costs of this synchronization for such collective operations by MPI-I/O[16], OpenMP-I/O[17], and UPC-I/O[18].
3) **Synchronous versus Asynchronous I/O** To avoid idle CPU cycles during the servicing of data transfer requests, images could benefit if such requests could proceed in an asynchronous manner. In a multi-processing environment, this leads to an opportunity of overlapping the I/O operations with useful computation and communication among the images.
4) **File Consistency Among Processors** In data-intensive HPC applications, performing repeated disk accesses to access consecutive sections of the disk will lead to a severe degradation of performance. There have been efforts to exploit the principle of locality by caching sections of the file which are frequently accessed by different images and to service the read/write operations more efficiently at the risk of relaxed consistency between the cache buffers and the physical non-volatile disks. This means that images might benefit from having a local copy of a section of the file and translate the read/write operations in the form of quick memory transfers. This involves additional pre- and post-processing at regular intervals to provide a consistent view of the shared file.

WG5, the working group responsible for the development of Fortran standards, has not yet settled upon a solution for parallel I/O to be part of the Fortran standard. One approach would be to follow a somewhat modest extension for parallel I/O that was originally part of Co-Array Fortran [2]. Here, the `OPEN` statement is extended to allow multiple images to collectively open a single file for shared access. Files are represented as a sequence of records of some fixed length. Once the file is opened, each image may independently read or write to it, one record at a time. It is the programmer's responsibility to coordinate these accesses to the files with the use of appropriate synchronization statements. Some details on how precisely the extension should be incorporated into the standard are still unresolved and under discussion [19], and hence it has not been slated for inclusion in the Technical Specification for extended features [20]. We point out some potential drawbacks from our perspective:

- Files are organized as a 1-dimensional sequence of records. Higher-level representations of data (e.g. multi-dimensional view of records) would be easier to deal with for programmers and potentially enable more optimized I/O accesses.
- Records may be read or written by an image only one at a time. This hinders the ability of the programmer (and, to a degree, the implementation) to aggregate reads and writes for amortizing the costs of I/O accesses.
- There is no mechanism for collective accesses to shared files. Collective I/O operations are extremely important, particularly as we scale up the number of images that may be doing I/O. New collective variants for the `READ` and `WRITE` statements will allow the implementation to funnel I/O accesses through dedicated nodes to reduce excessive traffic to secondary storage.

B. Incorporating Parallel I/O in Coarray Fortran

Before a detailed description of our proposed extensions, let us consider the file access pattern illustrated in Fig. 7, in which multiple images simultaneously access the non-contiguous, shaded portions of the file.

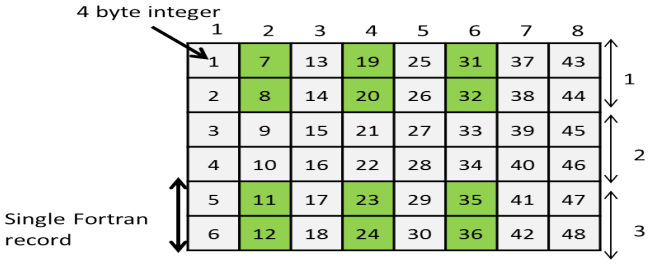


Fig. 7: Example of a file access pattern by multiple images

The following code snippet uses MPI I/O to enable each image to independently access this section of the input file.

```
1
2    INCLUDE .mpif.h.
3
4    MPI_INT blklens(3)
5    MPI_TYPE old_types(3)
6    MPI_INT indices(3)
7    MPI_INT ierr
8    .....
9
10 ! MPI_TYPE_VECTOR(count, blklen, str, old_type,
       new_type, err )
11   call MPI_TYPE_VECTOR(2, 2, 4, MPI_REAL8 ,type1
         , ... )
12
13 ! MPI_TYPE_VECTOR(count, blklen, str, old_type,
       new_type, err )
14   call MPI_TYPE_VECTOR(3, 1, 2, type1    ,
         type2, ... )
15
16 ! MPI_TYPE_VECTOR(count, blklen, str, old_type,
       new_type, err )
17   call MPI_TYPE_VECTOR( 3, 1, 2, type1, type2,
         ... )
18
19   count = 3
20   blklens = (/1, 1, 1/)
21   indices = (/0, 48, 384/) ! in bytes
22   old_types = (/MPI_LB, type2, MPI_UB)
23
24   call MPI_TYPE_STRUCT ( count , blklens,
         indices, old_types, newtype, ...)
25
26   call MPI_TYPE_COMMIT(newtype)
27
28 ! setting the view of the file for the image
29   call MPI_FILE_OPEN(MPI_COMM_WORLD,
         "/file_path/...", MPI_MODE_RD_ONLY, ...,
         fhdl, ...)
30
31   call MPI_FILE_SET_VIEW(fhdl, 0, MPI_REAL8,
         newtype, 'native', ... );
32
33 ! reading begins
34   call MPI_FILE_READ(fh, buff, 1 ,newtype,...)
35 ....
36   call MPI_FILE_FREE(newtype)
37   call MPI_FILE_CLOSE(fhdl)
```

The following is a code snippet to read the same file without changing the logical data layout in Fig. 7, using our CAF extension for parallel I/O.

```
1  OPEN(UNIT=10, TEAM='yes', ACCESS='DIRECT',
       ACTION='READ', FILE='/file_path/...',
       RECL=16, NDIMS=2 DIMS=(\3\))
2
3  READ(UNIT=10, REC_LB=(/1,2/), REC_UB=(/3,6/),
       REC_STR=(/2,2/)) buff(:)
4
5  CLOSE(10)
```

It can be observed that the use of the extensions improves usability by dramatically reducing the size of the code.

1) OPEN statement: The OPEN statement is used to open an external file by multiple images. Fortran files which are used for parallel I/O should contain only unformatted binary data. Fortran 2008 forbids the simultaneous opening of files by more than one image. However, to add support for parallel I/O, this restriction can be lifted with the `TEAM='yes'` specifier, which indicates that all images in the currently executing team[1] will collectively open the file for access.

Table I enlists the different types of control specifiers within the OPEN statement which are relevant to Parallel I/O support.

Some of the key specifiers which have been extended or added have been discussed below:

- `UNIT`: While `OPEN`ing a file collectively, all images must use the same unit number in the statement.
- `FILE`: This includes the full path of the file in the file system. While using a remote file system in a clustered environment, the programmer must ensure that the path-name provided should be the same and should evaluate to the same physical file on all the nodes on which the images execute.

[1]Teams are a feature that are likely to be added in the next revision of the Fortran standard and will be described in the forthcoming Technical Specification for additional parallel processing features

TABLE I: List of the primary specifiers related to the *modified* OPEN statement. The list includes existing specifiers as well as the newly proposed ones.

Specifier	Same value accross all images (Yes/No)
UNIT[1]	YES
FILE[1]	YES
IOSTAT[1]	NO
TEAM[2]	YES
NDIMS[2]	YES
DIMS[2]	YES
OFFSET[2]	YES
ACCESS[1]	YES
ACTION[1]	YES

[1] This specifier already exists in the current Fortran 2008 standard. However, the restrictions applicable to this specifier has been modified as part of this proposal.
[2] This specifier does not exist in the Fortran 2008 standard and has been proposed as an extension.

- `IOSTAT`: If an `OPEN` statement (with a valid `UNIT` number) is executed and if the associated file has already been opened by the same executing image, the *stat-variable* becomes defined with the processor-dependent positive integer value of the constant `STAT_FILE_OPENED_BEFORE` from the intrinsic module `ISO_FORTRAN_ENV`. Also, in accordance with Section 13.8.2.24 of the Fortran 2008 standard [21], if an OPEN statement is executed by an image when one or more images in the same `TEAM` (described below) has already terminated, the *stat-variable* becomes defined with the processor-dependent positive integer value of the constant `STAT_STOPPED_IMAGE`.
- `TEAM`: The `TEAM='yes'` specifier is used to indicate that a file is to be opened in a parallel I/O context (i.e. shared access between multiple images in the current team). The Fortran standard working group has proposed a mechanism for team formation by images [20]. Using team objects for parallel I/O allows the user to limit the I/O operations to a subset of images. In the absence of this specifier, the image is connected to the file with exclusive access.
- `NDIMS`: This specifier accepts a positive integer which specifies the number of dimensions of the representation of the data in the file.
- `DIMS`: This provides the programmer with the flexibility of representing sequential records in the file in the form of multi-dimensional arrays. The *array-expr* passed to the specifier contains a list of extents along each dimension except the last (which can be calculated internally using the record length and the total size of the file).
- `OFFSET`: This specifier accepts a positive integer which defines the offset within the file (in file storage units, typically bytes) at which the multi-dimensional file data starts. It may be used to reserve space in the file for header information.
- `ACCESS`: Out of the two access modes in Fortran - direct and sequential, performing a shared file access in parallel should be limited to direct-access mode. The WG5 standards committee suggests that sequential access mode should not be supported for parallel I/O.
- `ACTION`: All images must pass the same value (`read`, `write`, or `readwrite`) for the specifier.

Data-intensive scientific applications which require representations with multiple dimentional-views of the same dataset can represent the layout while using the `OPEN` statement. For example, Fig. 8 represents how a series of flat 24 records may be viewed as 8x3, or a 4x2x3 array of records.

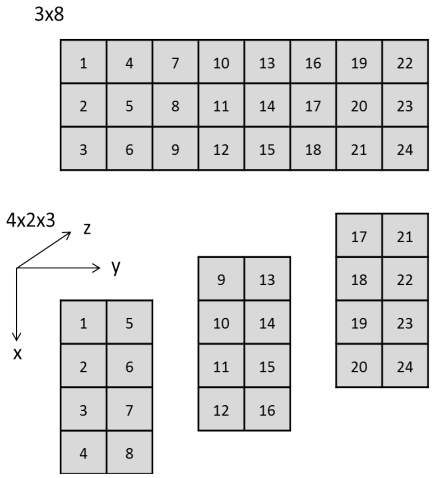


Fig. 8: The flexibility of the same dataset being represented using different dimension layout allows the adaptability of algorithms semantics to file contents

2) READ/WRITE statements: One of the main concerns for programming models that support parallel I/O for data intensive applications is the scalability of I/O requests and the actual transfer of the data among the multiple nodes and the I/O servers. To reduce excess network traffic in large scale systems, the runtime could take advantage of I/O requests from different images targetting adjacent sections in the shared file. This allows opportunities for combining the large number of requests to a few. This helps to avoid bottlenecks at the I/O servers, reducing average latency among the I/O servers and the nodes. To meet this end, we extend the `READ` and `WRITE` statements to enable *collective* accesses to different (though not necessarily *disjoint*) sections of the multi-dimensional data, using the optional `COLLECTIVE='yes'` specifier.

3) Asynchronous operations: I/O intensive applications alternate between computation and I/O phases. As a result, it is not essential that all the images have the same view of the file during the complete execution of the application. To support this, the `ASYNCHRONOUS` specifier as included by the Fortran 2003 standard can be extended for parallel I/O. The value *YES* needs to be passed to this specifier while connecting to a file (using `OPEN`). Additionaly, the programmer has a flexibility to choose which I/O operation on an already connected file, needs to be made asynchronous. This is achieved by explicitly specifying `ASYNCHRONOUS='yes'` for every I/O statement

on that file. Images may write to disjoint sections of the file, without its changes being immediately visible to other images sharing the same file. Such semantics enables the runtime to delay the completion of the effects of read/write operations until a definitive point is hit. This point in the program can be the existing `FLUSH` statement.

Table II enlists the different types of control specifiers within the READ / WRITE statements which are relevant to parallel I/O support.

C. Access patterns using the triplet representation

As described before, the DIMS specifier enables the programmer to set the layouts of the records of a file as a multi-dimensional array. Once a file is connected, the access pattern is represented by specifying a triplet *<lower bound, upper bounds, and strides>* along each dimension. These can be defined within the I/O READ and WRITE statements using the REC_LB, REC_UB, and REC_STR specifiers.

Large scale scientific applications deal with complex data patterns based on a multi-dimensional view of the algorithmic data structures. Consider Fig. 10a, where the highlighted sections of the arrays are accessed by an image. In this, the data layouts represent the usefulness of the triplet representation when compared to the flat sequential records view of the current Fortran standard or the streaming-bytes view of POSIX compliant C programs.

While most access patterns can be easily represented using a single triplet notation across each dimension, there exist corner cases, where multiple disjoint accesses become necessary to completely represent the complex patterns using a standard triplet notation. For example, consider Fig. 9. Here, the elements accessed (3, 4, 7, and 8) do not have a uniform stride between them when the file is represented as a 1x8 array. Using this layout, an application programmer might have to resort to performing multiple I/O operations (two steps in this case), to completely read the data. Such multiple I/O requests for small chunks of data clearly would take a toll on the overall performance. Instead, if the programmer were to choose to represent the layout of all the records as a 2x4 array (Fig. 10b), the access pattern can be easily represented using a single triplet representation. Fig. 11 illustrates this point further.

TABLE II: List of the primary specifiers related to the *modified* READ/WRITE statement. The list includes existing specifiers as well as the newly proposed ones.

I/O statement specifier	(if COLLECTIVE=YES) Same value across all images (Yes/No)
UNIT[1]	YES
ASYNCHRONOUS[1]	YES
IOSTAT[1]	NO
REC_LB[2]	NO
REC_UB[2]	NO
REC_STR[2]	NO
REC_STR[2]	NO
COLLECTIVE[2]	YES

[1] This specifier already exists in the current Fortran 2008 standard. However, the restrictions applicable to this specifier has been modified as part of this project.

[2] This specifier does not exist in the Fortran 2008 standard and has been proposed as an extension.

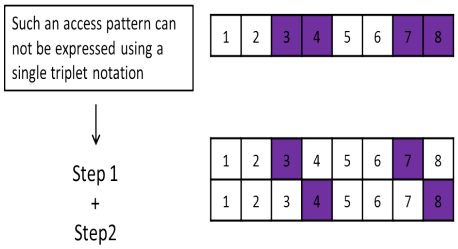


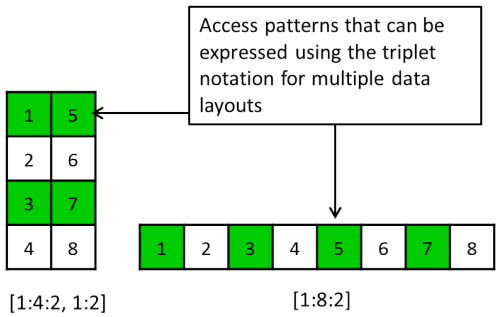
Fig. 9: Multiple steps used to access a pattern from a file represented as a one-dimensional array.

D. Parallel I/O Implementation

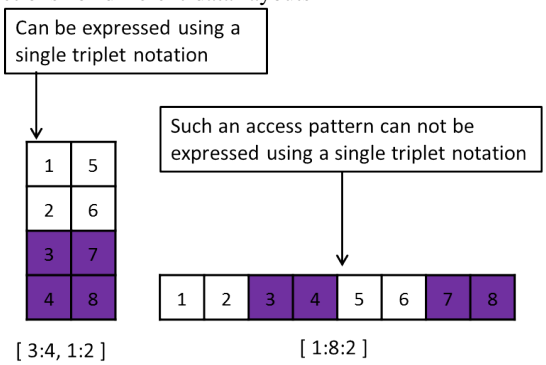
We are currently in the process of implementing the proposed parallel I/O language extensions. For our implementation, we incorporate the use of "global arrays", provided for example in the Global Arrays (GA) Toolkit [22] [23], into our CAF runtime. Global arrays are physically distributed in memory, but are accessible as a single entity in a logical global address space through the APIs provided in the GA toolkit. We make use of global arrays as I/O buffers in memory, taking advantage of the large physical memory available on the nodes and the fast interconnect in our HPC system. `READ` and `WRITE` statements are implemented using `get` or `put` to a segment of this global array buffer in memory. At the same time, an independent group of processes, managed by the runtime, will handle the data movement between the global array buffer and the actual storage devices asynchronously. The mapping of global arrays to disk could be implemented using disk resident arrays, as described by Nieplocha et al. in [24], but we are also exploring other options. We believe that the domain decomposition approach to parallelism used in RTM, which matches CAF's communication model, naturally lends itself to this array-based approach to parallel I/O.

VI. Related Work

Having an open-source compiler is important for an emerging language as it promotes sharing of ideas and encourages people to freely experiment with it. There have been few public Coarray Fortran implementations to date. Dotsenko et al. developed CAFC [5], a source-to-source implementation based on Open64 with runtime support based on ARMCI [8] and GASNet [7]. They used Open64 as a front-end and implemented enhancements in the IR-to-source translator to generate Fortran 90 source code that could then be compiled using a vendor compiler. G95 [25] provides a coarray implementation which adheres to the Fortran 2008 standard. G95 allows coarray programs to run on a single machine with multiple cores, or on multiple images across (potentially heterogeneous) networks (a license is required in this case for more than 4 images). There has been an on-going effort to implement coarrays in gfortran [26], and an updated design



(a) Different triplet notations can target the same file sections for different data layouts



(b) File sections that cannot be accessed using a single triplet notation can be rearranged using a different layout to allow ease of access

Fig. 10: Use of triplet notations to access different data patterns in a file

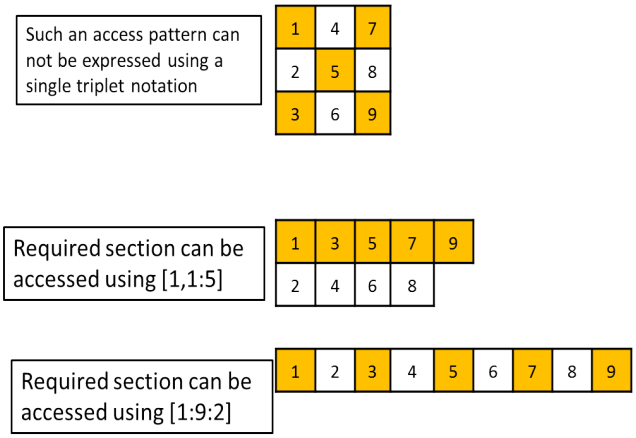


Fig. 11: The triplet notation can easily be used to represent the data access pattern if the file is represented logically as a one dimensional array instead of higher dimensions.

document for this implementation is maintained online. The gfortran implementation provides comprehensive support for single-image execution, but as of this writing the runtime support (which may be MPI- or ARMCI-based) for multi-image execution is mostly incomplete. Rice University, more recently, has developed an open source compiler for CAF 2.0 [27] which uses the ROSE compiler infrastructure. While CAF 2.0 provides support for "coarrays" that are functionally similar to coarrays as defined in Fortran 2008, their language features diverge significantly from the Fortran 2008 specification.

To utilize parallel I/O in CAF applications, one may combine the use of MPI library calls within CAF programs. However, this negates the goal of enhancing the programmability of data-intensive scientific applications. As illustrated in past examples, one of the strong features of parallel I/O in MPI is the availability of library interfaces to set *file-views* and read sections of the file. However, as illustrated in the prior code example, the use of MPI requires the programmer to explicitly calculate offsets for performing file I/O and then use different library functions to initialize the view. Since the proposed parallel I/O extension for Fortran allows for direct mapping of the data from the algorithm-domain to the actual application, a compiler's knowledge of the data types of the buffer involved in I/O and the actual control flow of the program itself can help determine the low-level details. Parallel I/O for Unified Parallel C (UPC), an extension to C which provides a PGAS programming mode similar to CAF, has been described [18]. However, the majority of the file operations are provided by collective operations and replicates MPI in terms of the file model. Parallel I/O support for OpenMP, a prevalently used API for shared memory programming, has also been proposed [17].

There exist supporting parallel I/O libraries which are implemented to serve a certain class of applications. For example, ChemIO [28] is a set of tuned parallel I/O libraries (like the Disk Resident Arrays [29]) with an execution model suited for data models common in chemistry applications. Efforts have also been directed towards designing a more abstract set of libraries which are closer to the data models used by scientific applications. Examples include Hierarchical Data Format (HDF5) [30] and Parallel Network Common Data Form (PnetCDF) [31][32]. The abstractions provided by these libraries can be implemented using middleware libraries like MPI. Such libraries store the data layout and dimensionality information in the form of file headers. However, scientific applications which dump raw data into storage disks can not always be relied upon to use such libraries to prepend the files with such headers. Doing so also restricts programmers from using different I/O interfaces, thereby affecting portability of the data files. Since we provide the option of specifying offsets in the OPEN statement (using the `OFFSET` specifier), in our proposed scheme CAF programs can be used with a wider variety of files.

VII. Conclusion

In this paper, we described the CAF programming model which has been standardized in Fortran 2008, and its implementation in an open-source compiler. Our implementation of the Fortran coarray extension is available at the OpenUH

website [33]. We have demonstrated that a widely used, data-intensive seismic processing application, reverse time migration (RTM), could be implemented using coarrays, and share much of the same code base as an MPI version. Performance was found to be comparable to the MPI implementation, outperforming the MPI version by up to 32% in one setting. The lack of a scalable parallel I/O feature in the programming model still needs to be addressed. We have defined an extension to Fortran for enabling parallel I/O. We are exploring a new approach for implementing these extensions which utilizes global arrays in memory as I/O buffers and asynchronously manages the data transfer between the buffer and secondary storage.

ACKNOWLEDGMENT

The authors would like to thank TOTAL management for supporting this project and granting permission to submit this paper. Thanks are also due to the Parallel Software Technologies Laboratory (PSTL) at the University of Houston for providing valuable feedback on the proposed semantics.

REFERENCES

[1] J. Reid, "Coarrays in the next fortran standard," *SIGPLAN Fortran Forum*, vol. 29, no. 2, pp. 10–27, Jul. 2010. [Online]. Available: http://doi.acm.org/10.1145/1837137.1837138

[2] R. W. Numrich and J. Reid, "Co-array Fortran for parallel programming," *SIGPLAN Fortran Forum*, vol. 17, no. 2, pp. 1–31, 1998.

[3] B. Chapman, D. Eachempati, and O. Hernandez, "Experiences Developing the OpenUH Compiler and Runtime Infrastructure," *International Journal of Parallel Programming*, pp. 1–32, Nov. 2012. [Online]. Available: http://www.springerlink.com/index/10.1007/s10766-012-0230-9

[4] D. Eachempati, H. J. Jun, and B. Chapman, "An Open-Source Compiler and Runtime Implementation for Coarray Fortran," in *PGAS'10*. New York, NY, USA: ACM Press, Oct 12-15 2010.

[5] Y. Dotsenko, C. Coarfa, and J. Mellor-Crummey, "A multi-platform co-array fortran compiler," in *PACT '04: Proceedings of the 13th International Conference on Parallel Architectures and Compilation Techniques*. Washington, DC, USA: IEEE Computer Society, 2004, pp. 29–40.

[6] W.-Y. Chen, C. Iancu, and K. Yelick, "Communication optimizations for fine-grained upc applications," in *Proceedings of the 14th International Conference on Parallel Architectures and Compilation Techniques*, ser. PACT '05. Washington, DC, USA: IEEE Computer Society, 2005, pp. 267–278. [Online]. Available: http://dx.doi.org/10.1109/PACT.2005.13

[7] D. Bonachea, "Gasnet specification, v1.1," Berkeley, CA, USA, Tech. Rep., 2002.

[8] J. Nieplocha and B. Carpenter, "ARMCI: A portable remote memory copy library for distributed array libraries and compiler run-time systems," in *Proceedings of the 11 IPPS/SPDP'99 Workshops Held in Conjunction with the 13th International Parallel Processing Symposium and 10th Symposium on Parallel and Distributed Processing*. Springer-Verlag, 1999, pp. 533–546.

[9] E. Baysal, D. Kosloff, and J. Sherwood, "Reverse time migration," *Geophysics*, vol. 48, no. 11, pp. 1514–1524, 1983.

[10] E. Dussaud, W. Symes, P. Williamson, L. Lemaistre, P. Singer, B. Denel, and A. Cherrett, "Computational strategies for reverse-time migration," 2008.

[11] R. Fletcher, X. Du, and P. Fowler, "Reverse time migration in tilted transversely isotropic (TTI) media." *Geophysics*, vol. 74, no. 6, pp. WCA179–WCA187, 2009.

[12] F. Billette and S. Brandsberg-Dahl, "The 2004 BP velocity benchmark," in *67th Annual Internat. Mtg., EAGE, Expanded Abstracts*. EAGE, 2005, p. B035.

[13] A. I. Stone, J. M. Dennis, and M. M. Strout, "Evaluating coarray fortran with the cgpop miniapp," in *Fifth Conference on Partitioned Global Address Space Programming Model (PGAS11), Texas, USA*, 2011.

[14] M. Hasert, H. Klimach, and S. Roller, "Caf versus mpi - applicability of coarray fortran to a flow solver," in *Recent Advances in the Message Passing Interface*, ser. Lecture Notes in Computer Science, Y. Cotronis, A. Danalis, D. Nikolopoulos, and J. Dongarra, Eds. Springer Berlin Heidelberg, 2011, vol. 6960, pp. 228–236. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-24449-0_26

[15] R. Barrett, "Co-array fortran experiences with finite differencing methods," in *The 48th Cray User Group meeting, Lugano, Italy*, 2006.

[16] R. Thakur, W. Gropp, and E. Lusk, "Data sieving and collective I/O in ROMIO," in *Proceedings of the 7th Symposium on the Frontiers of Massively Parallel Computation*, ser. FRONTIERS '99. Washington, DC, USA: IEEE Computer Society, 1999, pp. 182–. [Online]. Available: http://dl.acm.org/citation.cfm?id=795668.796733

[17] K. Mehta, E. Gabriel, and B. Chapman, "Specification and performance evaluation of parallel i/o interfaces for openmp," in *Proceedings of the 8th international conference on OpenMP in a Heterogeneous World*, ser. IWOMP'12. Berlin, Heidelberg: Springer-Verlag, 2012, pp. 1–14.

[18] T. El-ghazawi, F. Cantonnet, P. Saha, R. Thakur, R. Ross, and D. Bonachea, "UPC-IO: A Parallel I/O API for UPC v1.0pre10," 2003.

[19] J. Reid, "Requirements for further coarray features - N1924," May 2012.

[20] WG5, "Draft TS 18508 Additional Parallel Features in Fortran - N1983," July 2013.

[21] ANSI and ISO, "ISO/IEC 1539-1:2010 Information technology – Programming languages – Fortran – Part 1: Base language," 2010.

[22] "Global Arrays Webpage," http://www.emsl.pnl.gov/docs/global/.

[23] J. Nieplocha, B. Palmer, V. Tipparaju, M. Krishnan, H. Trease, and E. Aprà, "Advances, applications and performance of the global arrays shared memory programming toolkit," *Int. J. High Perform. Comput. Appl.*, vol. 20, no. 2, pp. 203–231, May 2006. [Online]. Available: http://dx.doi.org/10.1177/1094342006064503

[24] J. Nieplocha and I. Foster, "Disk resident arrays: An array-oriented i/o library for out-of-core computations," in *Proceedings of the 6th Symposium on the Frontiers of Massively Parallel Computation*, ser. FRONTIERS '96. Washington, DC, USA: IEEE Computer Society, 1996, pp. 196–. [Online]. Available: http://dl.acm.org/citation.cfm?id=795667.796727

[25] A. Beddall, "The g95 project," url:http://www.g95.org/coarray.shtml. [Online]. Available: http://www.g95.org/coarray.shtml

[26] T. Moene, "Towards an implementation of Coarrays in GNU Fortran," http://ols.fedoraproject.org/GCC/Reprints-2008/moene.reprint.pdf.

[27] J. Mellor-Crummey, L. Adhianto, and W. Scherer, "A new vision for Coarray Fortran," in *PGAS '09*. Rice University, 2009.

[28] J. Nieplocha, I. Foster, and R. A. Kendall, "ChemIO: High Performance Parallel I/O for Computational Chemistry Applications," 1998. [Online]. Available: http://www.mcs.anl.gov/chemio/

[29] J. Nieplocha and I. Foster, "Disk resident arrays: An array-oriented i/o library for out-of-core computations."

[30] T. H. Group, *HDF5 User's Guide, Release 1.6.6*, august 2007. [Online]. Available: http://hdfgroup.org/

[31] trac - Integerated SCM & Project Management. Parallel nerCDF : A High Performance API for NetCDF File Access. [Online]. Available: http://trac.mcs.anl.gov/projects/parallel-netcdf

[32] R. Pincus and R. Rew, *The NetCDF Fortran 90 Interface Guide*, June 2011.

[33] "The OpenUH compiler project," http://www.cs.uh.edu/~openuh, 2005.