Hindawi Publishing Corporation
Scientific Programming
Volume 2015, Article ID 904983, 12 pages
http://dx.doi.org/10.1155/2015/904983

Research Article

Using Coarrays to Parallelize Legacy Fortran Applications: Strategy and Case Study

Hari Radhakrishnan,[1] Damian W. I. Rouson,[2] Karla Morris,[3] Sameer Shende,[4] and Stavros C. Kassinos[5]

[1] *EXA High Performance Computing, 1087 Nicosia, Cyprus*
[2] *Stanford University, Stanford, CA 94305, USA*
[3] *Sandia National Laboratories, Livermore, CA 94550, USA*
[4] *University of Oregon, Eugene, OR 97403, USA*
[5] *Computational Sciences Laboratory (UCY-CompSci), University of Cyprus, 1678 Nicosia, Cyprus*

Correspondence should be addressed to Damian W. I. Rouson; damian@rouson.net

Received 8 April 2014; Accepted 5 August 2014

Academic Editor: Jeffrey C. Carver

This paper summarizes a strategy for parallelizing a legacy Fortran 77 program using the object-oriented (OO) and coarray features that entered Fortran in the 2003 and 2008 standards, respectively. OO programming (OOP) facilitates the construction of an extensible suite of model-verification and performance tests that drive the development. Coarray parallel programming facilitates a rapid evolution from a serial application to a parallel application capable of running on multicore processors and many-core accelerators in shared and distributed memory. We delineate 17 code modernization steps used to refactor and parallelize the program and study the resulting performance. Our initial studies were done using the Intel Fortran compiler on a 32-core shared memory server. Scaling behavior was very poor, and profile analysis using TAU showed that the bottleneck in the performance was due to our implementation of a collective, sequential summation procedure. We were able to improve the scalability and achieve nearly linear speedup by replacing the sequential summation with a parallel, binary tree algorithm. We also tested the Cray compiler, which provides its own collective summation procedure. Intel provides no collective reductions. With Cray, the program shows linear speedup even in distributed-memory execution. We anticipate similar results with other compilers once they support the new collective procedures proposed for Fortran 2015.

1. Introduction

Background. Legacy software is old software that serves a useful purpose. In high-performance computing (HPC), a code becomes "old" when it no longer effectively exploits current hardware. With the proliferation of multicore processors and many-core accelerators, one might reasonably label any serial code as "legacy software." The software that has proved its utility over many years, however, typically has earned the trust of its user community.

Any successful strategy for modernizing legacy codes must honor that trust. This paper presents two strategies for parallelizing a legacy Fortran code while bolstering trust in the result: (1) a test-driven approach that verifies the numerical results and the performance relative to the original code and (2) an evolutionary approach that leaves much of the original code intact while offering a clear path to execution on multicore and many-core architectures in shared and distributed memory.

The literature on modernizing legacy Fortran codes focuses on programmability issues such as increasing type safety and modularization while reducing data dependancies via encapsulation and information hiding. Achee and Carver [1] examined object extraction, which involves identifying candidate objects by analyzing the data flow in Fortran 77 code. They define a cohesion metric that they use to group global variables and parameters. They then extracted methods from the source code. In a 1500-line code, for example, they extract 26 candidate objects.

Norton and Decyk [2], on the other hand, focused on wrapping legacy Fortran with more modern interfaces. They then wrap the modernized interfaces inside an object/abstraction layer. They outline a step-by-step process that ensures standards compliance, eliminates undesirable features, creates interfaces, adds new capabilities, and then groups related abstractions into classes and components. Examples of undesirable features include **common** blocks, which potentially facilitate global data-sharing and aliasing of variable names and types. In Fortran, giving procedures explicit interfaces facilitates compiler checks on argument type, kind, and rank. New capabilities they introduced included dynamic memory allocation.

Greenough and Worth [3] surveyed tools that enhance software quality by helping to detect errors and to highlight poor practices. The appendices of their report provide extensive summaries of the tools available from eight vendors with a very wide range of capabilities. A sample of these capabilities includes memory leak detection, automatic vectorization and parallelization, dependency analysis, call-graph generation, and static (compile-time) as well as dynamic (run-time) correctness checking.

Each of the aforementioned studies explored how to update codes to the Fortran 90/95 standards. None of the studies explored subsequent standards and most did not emphasize performance improvement as a main goal. One recent study, however, applied automated code transformations in preparation for possible shared-memory, loop-level parallelization with OpenMP [4]. We are aware of no published studies on employing the Fortran 2008 coarray parallel programming to refactor a serial Fortran 77 application. Such a refactoring for parallelization purposes is the central aim of the current paper.

Case Study: PRM. Most commercial software models for turbulent flow in engineering devices solve the Reynolds-averaged Navier-Stokes (RANS) partial differential equations. Deriving these equations involves decomposing the fluid velocity field, $\mathbf{u}$, into a mean part, $\overline{\mathbf{u}}$, and a fluctuating part, $\mathbf{u}'$:

$$\mathbf{u} \equiv \overline{\mathbf{u}} + \mathbf{u}'. \tag{1}$$

Substituting (1) into a momentum balance and then averaging over an ensemble of turbulent flows yield the following RANS equation:

$$\rho \overline{u_j} \frac{\partial \overline{u_i}}{\partial x_j} = \rho \overline{f_i} + \frac{\partial}{\partial x_j} \left[-\overline{p}\delta_{ij} + \mu \left(\frac{\partial \overline{u_i}}{\partial x_j} + \frac{\partial \overline{u_j}}{\partial x_i} \right) - \rho \overline{u_i' u_j'} \right], \tag{2}$$

where μ is the fluid's dynamic viscosity; ρ is the fluid's density; t is the time coordinate; u_i and u_j are the ith and jth cartesian components of $\mathbf{u}$; and x_i and x_j are the ith and jth cartesian components of the spatial coordinate $\mathbf{x}$.

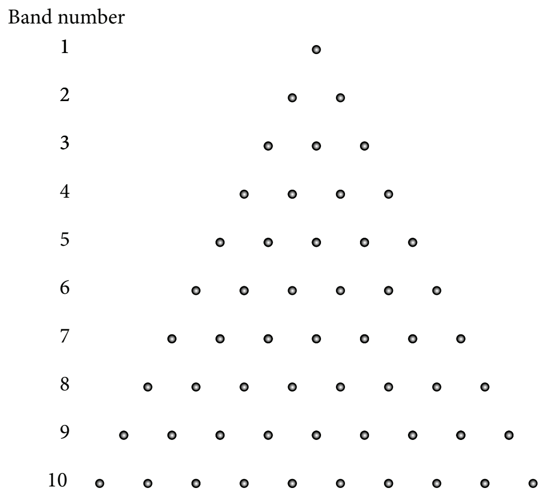


FIGURE 1: Distribution of particles in bands in one octant.

The term $-\rho\overline{u_i' u_j'}$ in (2) is called the Reynolds stress tensor. Its presence poses the chief difficulty at the heart of Reynolds-averaged turbulence modeling; closing the RANS equations requires postulating relations between the Reynolds stress and other terms appearing in the RANS equations, typically the velocity gradient $\partial\overline{u_j}/\partial x_i$ and scalars representing the turbulence scale. Doing so in the most common ways works well for predicting turbulent flows in which the statistics of $\mathbf{u}'$ stay in near-equilibrium with the flow deformations applied via gradients in $\overline{\mathbf{u}}$. Traditional RANS models work less well for flows undergoing deformations so rapid that the fluctuating field responds solely to the deformation without time for the nonlinear interactions with itself that are the hallmark of fluid turbulence. The Particle Representation Model (PRM) [5, 6] addresses this shortcoming. Given sufficient computing resources, a software implementation of the PRM can exactly predict the response of the fluctuating velocity field to rapid deformations.

A proprietary in-house software implementation of the PRM was developed initially at Stanford University, and development continued at the University of Cyprus. The PRM uses a set of hypothetical particles over a unit hemisphere surface. The particles are distributed on each octant of the hemisphere in bands, as shown in Figure 1 for ten bands. The total number of particles is given by

$$\begin{aligned} N_{\text{particles}} &= \underbrace{4}_{\text{Number of octants in hemisphere}} \\ &\quad \times \underbrace{\frac{N_{\text{bands}} \times (N_{\text{bands}} + 1)}{2}}_{\text{Number of particles in one octant}} \\ &= 2 \times N_{\text{bands}} \times (N_{\text{bands}} + 1). \end{aligned} \tag{3}$$

So, the computational time scales quadratically with the number of bands used.

Each particle has a set of assigned properties that describe the characteristics of an idealized flow. Assigned particle properties include vector quantities such as velocity and

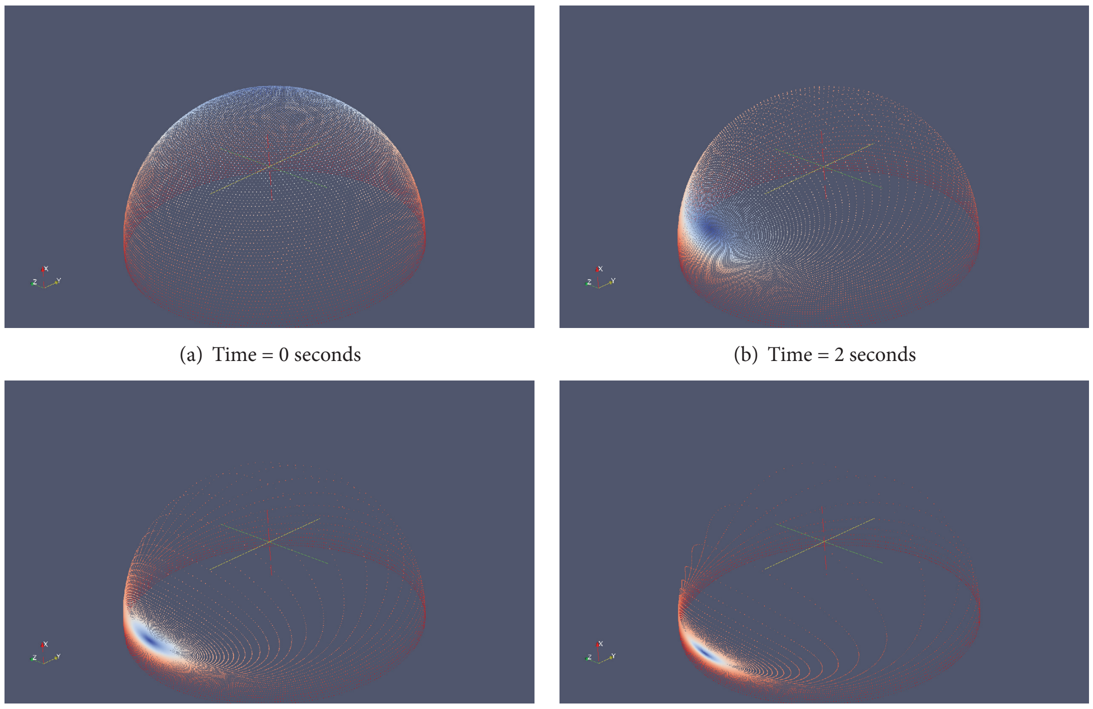

(a) Time = 0 seconds (b) Time = 2 seconds (c) Time = 4 seconds (d) Time = 6 seconds

FIGURE 2: Results of a PRM computation. The particles are colored based on their initial location. The applied flow condition, shear flow along the y-direction, causes the uniformly distributed particles to aggregate along that axis.

orientation as well as scalar quantities such as pressure. Thus, each particle can be thought of as representing the dynamics of a hypothetical one-dimensional (1D), one-component (1C) flow. Tracking a sufficiently large number of particles and then averaging the properties of all the particles (as shown in Figure 2), that is, all the possible flows considered, yield a representation of the 3D behavior in an actual flowing fluid.

Historically, a key disadvantage of the PRM has been costly execution times because a very large number of particles are needed to accurately capture the physics of the flow. Parallelization can reduce this cost significantly. Previous attempts to develop a parallel implementation of the PRM using MPI were abandoned because the development, validation, and verification times did not justify the gains. Coarrays allowed us to parallelize the software with minimal invasiveness and the OO test suite facilitated a continuous build-and-test cycle that reduced the development time.

2. Methodology

2.1. Modernization Strategy. Test-Driven Development (TDD) grew out of the Extreme Programming movement of the 1990s, although the basic concepts date as far back as the NASA space program in the 1960s. TDD iterates quickly toward software solutions by first writing tests that specify what the working software must do and then writing only a sufficient amount of application code in order to pass the test. In the current context, TDD serves the purpose of ensuring that our refactoring exercise preserves the expected results for representative production runs.

Table 1 lists 17 steps employed in refactoring and parallelizing the serial implementation of the PRM. They have been broken down into groups that addressed various facets of the refactoring process. The open-source CTest framework that is part of CMake was used for building the tests. Our first step, therefore, was to construct a CMake infrastructure that we used for automated building and testing and to set up a code repository for version control and coordination.

The next six steps address Fortran 77 features that have been declared obsolete in more recent standards or have been deprecated in the Fortran literature. We did not replace **continue** statements with **end do** statements as these did not affect the functionality of the code.

The next two steps were crucial in setting up the build testing infrastructure. We automated the initialization by replacing the keyboard inputs with default values. The next step was to construct extensible tests based on these default values, which are described in Section 3.

The next three steps expose optimization opportunities to the compiler. One exploits Fortran's array syntax. Two exploit Fortran's facility for explicitly declaring a procedure to be "**pure**," that is, free of side effects, including input/output, modifying arguments, halting execution, or modifying nonlocal state. Other steps address type safety and memory management.

Array syntax gives the compiler a high-level view of operations on arrays in ways the compiler can exploit with various optimizations, including vectorization. The ability to communicate functional purity to compilers also enables numerous compiler optimizations, including parallelism.

Table 1: Modernization steps: horizontal lines indicate partial ordering.

Step	Details
1	Set up automated builds via CMake[1] and version control via Git[2].
2	Convert fixed- to free-source format via "convert.f90" by Metcalf[3].
3	Replace **goto** with **do while** for main loop termination.
4	Enforce type/kind/rank consistency of arguments and return values by wrapping all procedures in a **module**.
5	Eliminate implicit typing.
6	Replace **data** statements with **parameter** statements.
7	Replace write-access to **common** blocks with module variables.
8	Replace keyboard input with default initializations.
9	Set up automated, extensible tests for accuracy and performance via OOP and CTest[1].
10	Make all procedures outside of the main program **pure.**
11	Eliminate actual/dummy array shape inconsistencies by passing array subsections to assumed-shape arrays.
12	Replace static memory allocation with dynamic allocation.
13	Replace loops with array assignments.
14	Expose greater parallelism by unrolling the nested loops in the particle set-up.
15	Balance the work distribution by spreading particles across images during set-up.
16	Exploit a Fortran 2015 collective procedure to gather statistics.
17	Study and tune performance with TAU[4].

[1]http://www.cmake.org/.
[2]http://git-scm.com/.
[3]ftp://ftp.numerical.rl.ac.uk/pub/MandR/convert.f90.
[4]http://tau.uoregon.edu/.

The final steps directly address parallelism and optimization. One unrolls a loop to provide for more fine-grained data distribution. The other exploits the co_sum intrinsic collective procedure that is expected to be part of Fortran 2015 and is already supported by the Cray Fortran compiler. (With the Intel compiler, we write our own co_sum procedure.) The final step involves performance analysis using the Tuning and Analysis Utilities [7].

3. Extensible OO Test Suite

At every step, we ran a suite of accuracy tests to verify that the results of a representative simulation did not deviate from the serial code's results by more than 50 parts per million (ppm). We also ran a performance test to ensure that the single-image runtime of the parallel code did not exceed the serial code's runtime by more than 20%. (We allowed for some increase with the expectation that significant speedup would result from running multiple images.)

Our accuracy tests examine tensor statistics that are calculated using the PRM. In order to establish a uniform protocol for running tests, we defined an abstract base tensor class as shown in Listing 1.

The base class provided the bindings for comparing tensor statistics, displaying test results to the user, and exception handling. Specific tests take the form of three child classes, reynolds_stress, dimensionality, and circulicity, that extend the tensor class and thereby inherit a responsibility to implement the tensor's deferred bindings compute_results and expected_results. The class diagram is shown in Figure 3. The tests then take the form

```
if (.not. stess_tensor%verify_result (when)) &
  error stop 'Test_failed.'
```

where stress_tensor is an instance of one of the three child classes shown in Figure 3 that extend tensor; "when" is an integer time stamp; **error stop** halts all images and prints the shown string to standard error; and verify_result is the **pure** function shown in Listing 1 that invokes the two aforementioned deferred bindings to compare the computed results to the expected results.

4. Coarray Parallelization

Modern HPC software must be executed on multicore processors or many-core accelerators in shared or distributed memory. Fortran provides for such flexibility by defining a partitioned global address space (PGAS) without referencing how to map coarray code onto a particular architecture. Coarray Fortran is based on the Single Program Multiple Data (SPMD) model, and each replication of the program is called an image [8]. Fortran 2008 compilers map these images to an underlying transport network of the compiler's choice. For example, the Intel compiler uses MPI for the transport network whereas the Cray compiler uses a dedicated transport layer.

A coarray declaration of the form

```
real, allocatable :: a (:, :, :) [:]
```

facilitates indexing into the variable "a" along three regular dimensions and one codimension so

```
a (1, 1, 1) = a (1, 1, 1) [2]
```

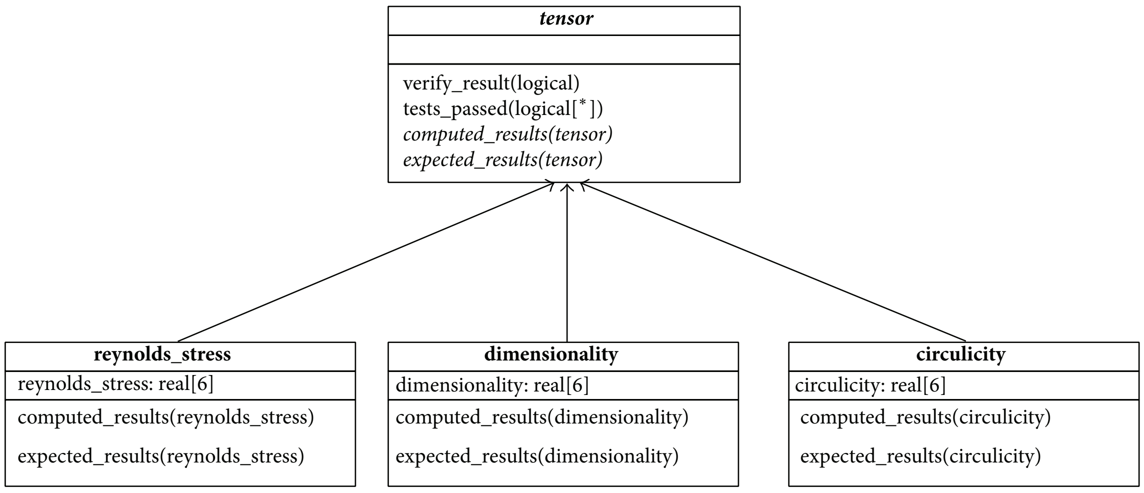


FIGURE 3: Class diagram of the testing framework. The deferred bindings are shown in italics, and the abstract class is shown in bold italics.

```
module abstract_tensor_class
  type, abstract :: tensor
  contains
      procedure(return_computed_results), deferred :: &
          computed_results
      procedure(return_expected_results), deferred :: &
          expected_results
      procedure :: verify_result
  end type
  abstract interface
    pure function return_computed_results(this) &
      result(computed_values)
      import :: tensor
      class(tensor), intent(in) :: this
      real, allocatable :: computed_values(:)
    end function
    ! return_expected_results interface omitted
  end abstract interface
 contains
  pure function verify_result(this) &
    result(all_tests_passed)
    class(tensor), intent(in) :: this
    logical :: all_tests_passed
    all_tests_passed = all(tests_passed( &
    this%computed_results(), this%expected_results()))
  end function
end module
```

LISTING 1: Base tensor class.

```
l = 0 ! Global particle number
do k = 1, nb ! Loop over the bands
    do m = 1, k ! Loop over the particles in band
        ! First octant
        l = l + 1
        ! Do some computations
        ! Second octant
        l = l + 1
        ! Do some computations
        ! Third octant
        l = l + 1
        ! Do some computations
        ! Fourth octant
        l = l + 1
        ! Do some computations
    end do
end do
```

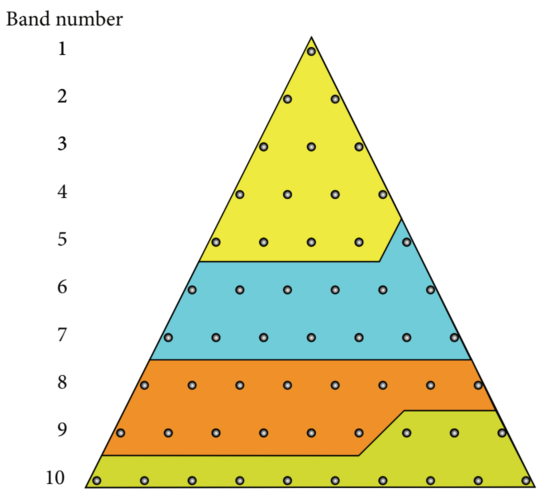
LISTING 2: Legacy particle loop.

copies the first element of image 2 to the first element of whatever image executes this line. The ability to omit the coindex on the left-hand side (LHS) played a pivotal role in refactoring the serial code with minimal work; although we added codimensions to existing variables' declarations, subsequent accesses to those variables remained unmodified except where communication across images is desired. When necessary, adding coindices facilitated the construction of collective procedures to compute statistics.

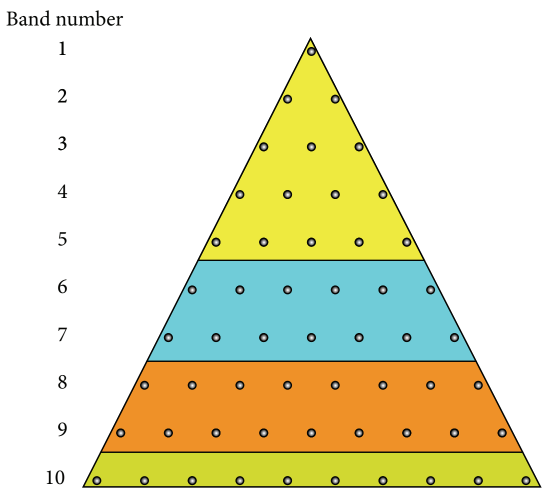
In the legacy version, the computations of the particle properties were done using two nested loops, as shown in Listing 2.

Distributing the particles across the images and executing the computations inside these loops can speed up the execution time. This can be achieved in two ways.

Method 1 works with the particles directly, splitting them as evenly as possible across all the images, allowing image boundaries to occur in the middle of a band. This distribution is shown in Figure 4(a). To achieve this distribution, the two nested do loops are replaced by one loop over the particles, and the indices for the two original loops are computed from the global particle number, as shown in Listing 3. However in this case, the code becomes complex and sensitive to precision.



(a) Partitioning of the particles to achieve even distribution of particles



(b) Partitioning of the bands to achieve nearly even distribution of particles

FIGURE 4: Two different partitioning schemes were tried for load balancing.

```
! Loop over the particles
do l = my_first_particle, my_last_particle, 4
    k = nint(sqrt(real(l) * 0.5))
    m = (l - (1 + 2 * k * (k - 1) - 4))/4
        ! First octant
        ! Do some computations
        ! Second octant
        ! Do some computations
        ! Third octant
        ! Do some computations
        ! Fourth octant
        ! Do some computations
end do
```

LISTING 3: Parallel loop by splitting particles.

```
! Loop over the bands
do k = my_first_band, my_last_band
    ! Global number
    ! of last particle in (k - 1) band
    l = k ** 2 + (k - 1) ** 2 - 1
    ! Loop over the particles in band
    do m = 1, k
        ! First octant
        l = l + 1
        ! Do some computations
        ! Second octant
        l = l + 1
        ! Do some computations
        ! Third octant
        l = l + 1
        ! Do some computations
        ! Fourth octant
        l = l + 1
        ! Do some computations
    end do
end do
```

LISTING 4: Parallel loop by splitting bands.

Method 2 works with the bands, splitting them across the images to make the particle distribution as even as possible. This partitioning is shown in Figure 4(b). Method 2, as shown in Listing 4, requires fewer changes to the original code shown in Listing 2 but is suboptimal in load balancing.

5. Results

5.1. Source Code Impact. We applied our strategy to two serial software implementations of the PRM. For one version, the resulting code was 10% longer than the original: 639 lines versus 580 lines with no test suite. In the second version, the code expanded 40% from 903 lines to 1260 lines, not including new input/output (I/O) code and the test code described in Section 3. The test and I/O code occupied additional 569 lines.

5.2. Ease of Use: Coarrays versus MPI. The ability to drop the coindex from the notation, as explained in Section 4, was a big help in parallelizing the program without making significant changes to the source code. A lot of the bookkeeping is handled behind the scenes by the compiler making it possible to make the parallelization more abstract but also easier to follow. For example, Listing 5 shows the MPI calls necessary to gather the local arrays into a global array on all the processors.

The equivalent calls using the coarray syntax is the listing shown in Listing 6.

Reducing the complexity of the code also reduces the chances of bugs in the code. In the legacy code, the arrays

```
integer :: my_rank, num_procs
integer, allocatable, dimension(:) :: &
    my_first, my_last, counts, displs
call mpi_comm_size(MPI_COMM_WORLD, num_procs, ierr)
call mpi_comm_rank(MPI_COMM_WORLD, my_rank, ierr)
allocate(my_first(num_procs), my_last(num_procs), &
    counts(num_procs), displs(num_procs))
my_first(my_rank + 1) = lbound(sn, 2)
my_last(my_rank + 1) = ubound(sn, 2)
call mpi_allgather(MPI_IN_PLACE, 1, MPI_INTEGER, &
  my_first, 1, MPI_INTEGER, MPI_COMM_WORLD, ierr)
call mpi_allgather(MPI_IN_PLACE, 1, MPI_INTEGER, &
  my_last, 1, MPI_INTEGER, MPI_COMM_WORLD, ierr)
do i = 1, num_procs
    displs(i) = my_first(i) − 1
    counts(i) = my_last(i) − my_first(i) + 1
end do
call mpi_allgatherv(sn, 5 ∗ counts(my_rank + 1), &
  MPI_DOUBLE_PRECISION, sn_global, 5 ∗ counts, &
  5 ∗ displs, MPI_DOUBLE_PRECISION, MPI_COMM_WORLD, ierr)
call mpi_allgatherv(cr, 5 ∗ counts(my_rank + 1), &
  MPI_DOUBLE_PRECISION, cr_global, 5 ∗ counts, &
  5 ∗ displs, MPI_DOUBLE_PRECISION, MPI_COMM_WORLD, ierr)
```

Listing 5: Using MPI_ALLGATHER to collect local arrays into a global array.

```
integer :: my_first[∗], my_last[∗]
my_first = lbound(sn, 2)
my_last = ubound(sn, 2)
do l = 1, num_images()
    cr_global(:, my_first[l]:my_last[l]) = cr(:,:)[l]
    sn_global(:, my_first[l]:my_last[l]) = sn(:,:)[l]
end do
```

Listing 6: Coarray method of gathering arrays.

sn and *cr* carried the information about the state of the particles. By using the coarray syntax and dropping the coindex, we were able to reuse all the original algorithms that implemented the core logic of the PRM. This made it significantly easier to ensure that the refactoring did not alter the results of the model. The main changes were to add codimensions to the *sn* and *cr* declarations and update them when needed, as shown in Listing 6.

5.3. Scalability. We intend for PRM to serve as an alternative to turbulence models used in routine engineering design of fluid devices. There is no significant difference in the PRM results when more than 1024 bands (approximately 2.1 million particles) are used to represent the flow state so this was chosen as the upper limit of the size of our data set. Most engineers and designers run simulations on desktop computers. As such, the upper bound on what is commonly available is roughly 32 to 48 cores on two or four central processing units (CPUs) plus additional cores on one or more accelerators. We also looked at the scaling performance of parallel implementation of the PRM using Cray hardware and Fortran compiler which has excellent support for distributed-memory execution of coarray programs.

Figure 5 shows the speedup obtained for 200 and 400 bands with the Intel Fortran compiler using the two particle-distribution schemes described in the Coarray Parallelization section. The runs were done using up to 32 cores on the “fat” nodes of ACISS (http://aciss-computing.uoregon.edu/). Each node has four Intel X7560 2.27 GHz 8-core CPUs and 384 GB of DDR3 memory. We see that the speedup was very poor when the number of processors was increased.

We used TAU [7] to profile the parallel runs to understand the bottlenecks during execution. Figure 6 shows the TAU plot for the runtime share for the dominant procedures using different number of images. Figure 7 shows the runtimes for the different functions on the different images. The heights of the columns show the runtime for different functions on the individual cores. There is no significant difference in the heights of the columns proving that the load balancing is very good across the images. We achieved this by mainly using the one-sided communication protocols of CAF as shown in Listing 6 and restricting the sync statements to the collective procedures as shown in Listings 7 and 8. Looking at the runtimes in Figure 6, we identified the chief bottlenecks to be the two collective co_sum procedures which sum values across a coarray by sequentially polling each image for its portion of the coarray. The time required for this procedure

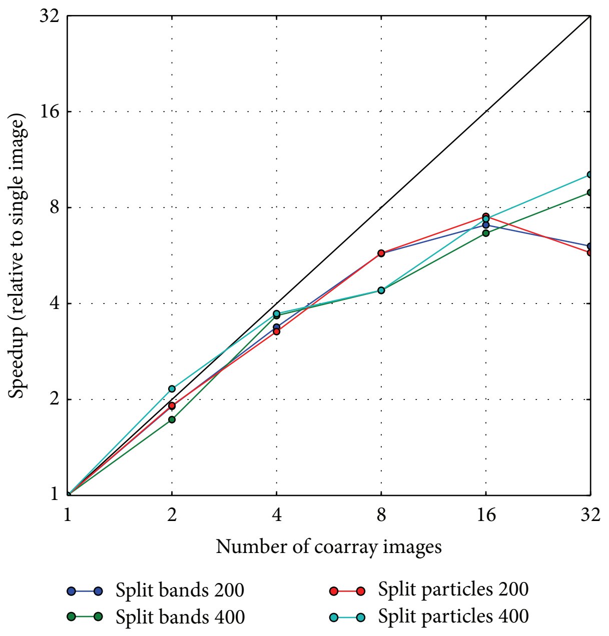


Figure 5: Speedup obtained with sequential co_sum implementation using multiple images on a single server.

```
subroutine vector_co_sum_parallel(vector)
  real(rkind), intent(inout) :: vector(:)[*]
  real(rkind), allocatable :: temp(:)
  integer image, step
  allocate (temp, mold = vector)
  step = 2
  do while (step/2 <= num_images())
    sync all
    if (this_image() + step/2 <= num_images()) then
      temp = vector + vector[this_image() + step/2]
    else
      temp = vector
    end if
    sync all
    vector = temp
    step = step * 2
  end do
  sync all
  if (this_image()/ = 1) vector = vector[1]
  sync all
end subroutine
```

Listing 8: Optimized collective sum routine.

```
subroutine vector_co_sum_serial(vector)
  real(rkind), intent(inout) :: vector(:)[*]
  integer image
  sync all
  if (this_image() == 1) then
    do image = 2, num_images()
      vector(:)[1] = vector(:)[1] + vector(:)[image]
    end do
  end if
  sync all
  if (this_image()/ = 1) vector(:) = vector(:)[1]
  sync all
end subroutine
```

Listing 7: Unoptimized collective sum routine.

is $O(N_{\text{images}})$. The unoptimized co_sum routine for adding a vector across all images is shown in Listing 7. There is an equivalent subroutine for summing a matrix also.

Designing an optimal co_sum algorithm is a platform-dependent exercise best left to compilers. The Fortran standards committee is working on a co_sum intrinsic procedure that will likely become part of Fortran 2015. But to improve the parallel performance of the program, we rewrote the collective co_sum procedures using a binomial tree algorithm that is $O(\log N_{\text{images}})$ in time. The optimized version of the co_sum version is shown in Listing 8.

The speedup obtained with the optimized co_sum routine is shown in Figure 8. We see that the scaling performance of the program becomes nearly linear with the implementation of the optimized co_sum routine. We also see that the scaling efficiency increases when the problem size is increased. This indicates that the poor scaling at smaller problem sizes is due to communication and synchronization [9].

The TAU profile analysis of the runs using different number of images is shown in Figure 9. While there is a small increase in the co_sum computation time when increasing the number of images, it is significantly lower than increase in time for the unoptimized version.

To fully understand the impact of the co_sum routines, we also benchmarked the program using the Cray compiler and hardware. Cray has native support for the co_sum directive in the compiler. Cray also uses its own communication library on Cray hardware instead of building on top of MPI as is done by the Intel compiler. As we can see in Figure 10, the parallel code showed very good strong scaling on the Cray hardware up to 128 images for the problem sizes that we tested.

We also looked at the TAU profiles of the parallel code on the Cray hardware, shown in Figure 11. The profile analysis shows that the time is spent mainly in the time advancement loop when the native co_sum implementation is used.

We hope that, with the development and implementation of intrinsic co_sum routines as part of the 2015 Fortran standard, the Intel compiler will also improve its strong scaling performance with larger number of images. Table 2 shows the raw runtimes for the different runs using 128 bands whose TAU profiles have been shown in Figures 6, 9, and 11. The runtimes for one to four images are very close but they quickly diverge as we increase the number of images due to the impact of the collective procedures.

Table 3 shows the weak scaling performance of the program using the optimized co_sum procedures using the Intel compiler. The number of particles as shown in Figure 1 scales as the square of the number of bands. Therefore, when

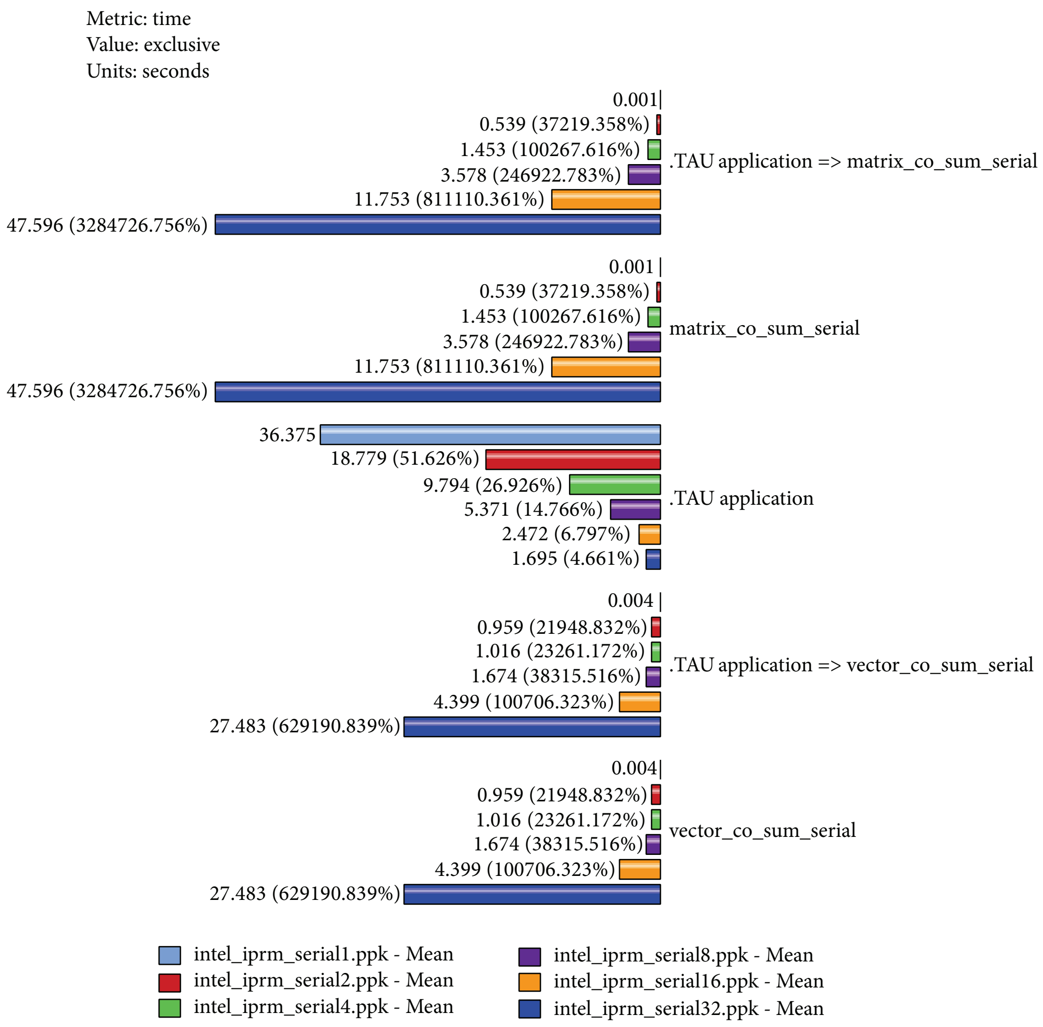


FIGURE 6: TAU profiling analysis of function runtimes when using the unoptimized co_sum routines with 1, 2, 4, 8, 16, and 32 images. The *.TAU application* is the main program wrapped by TAU for profiling, and *.TAU application* => refers to functions wrapped by TAU. This notation is also seen in Figures 7 and 9.

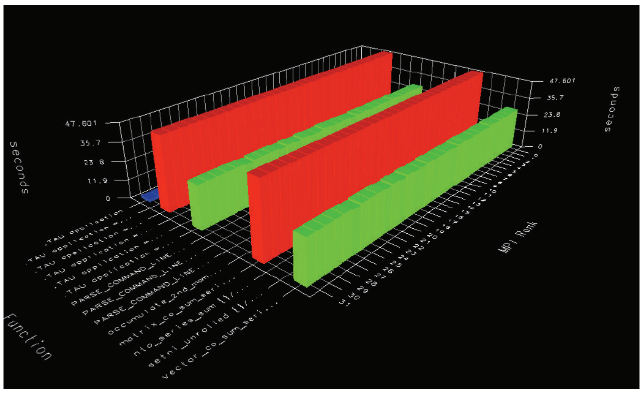


FIGURE 7: TAU analysis of load balancing and bottlenecks for the parallel code using 32 images.

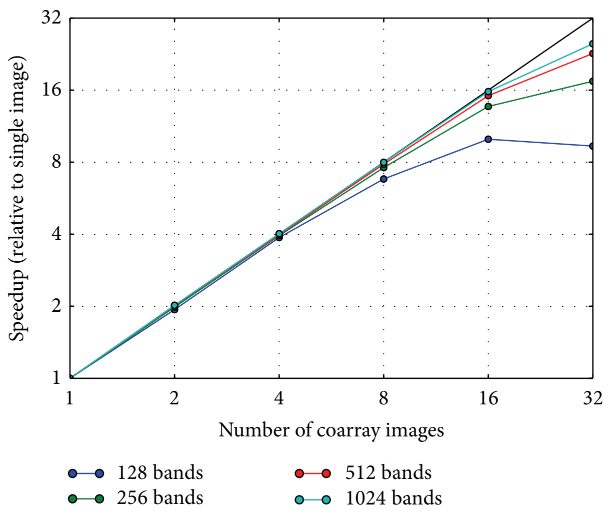


FIGURE 8: Speedup obtained with parallel co_sum implementation using multiple images on a single server.

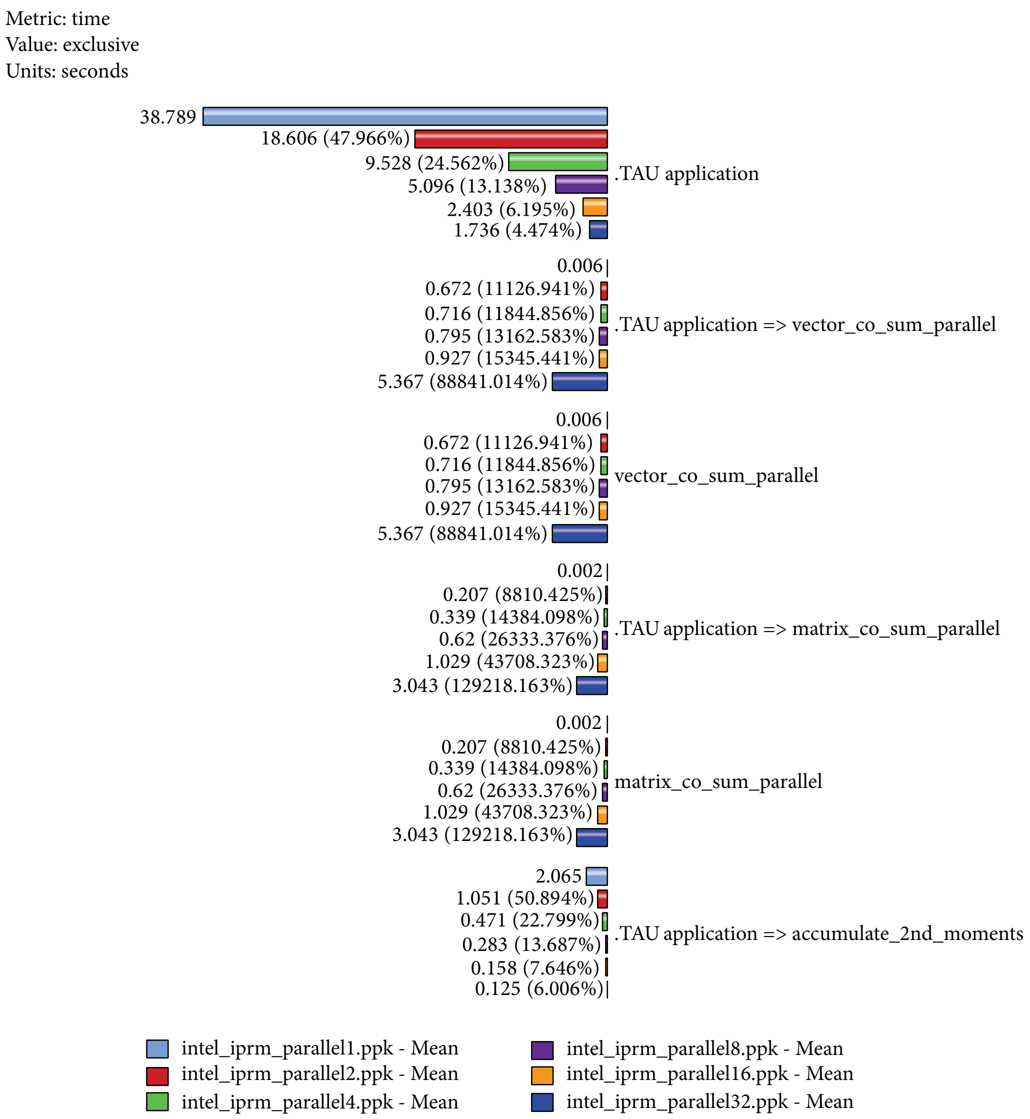


FIGURE 9: TAU profiling analysis of function runtimes when using the optimized co_sum routines with 1, 2, 4, 8, 16, and 32 images.

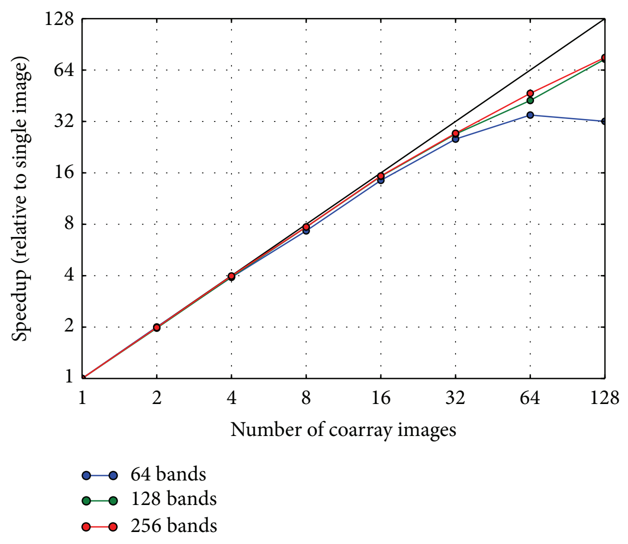


Figure 10: Speedup obtained with parallel co_sum implementation using multiple images on a distributed-memory Cray cluster.

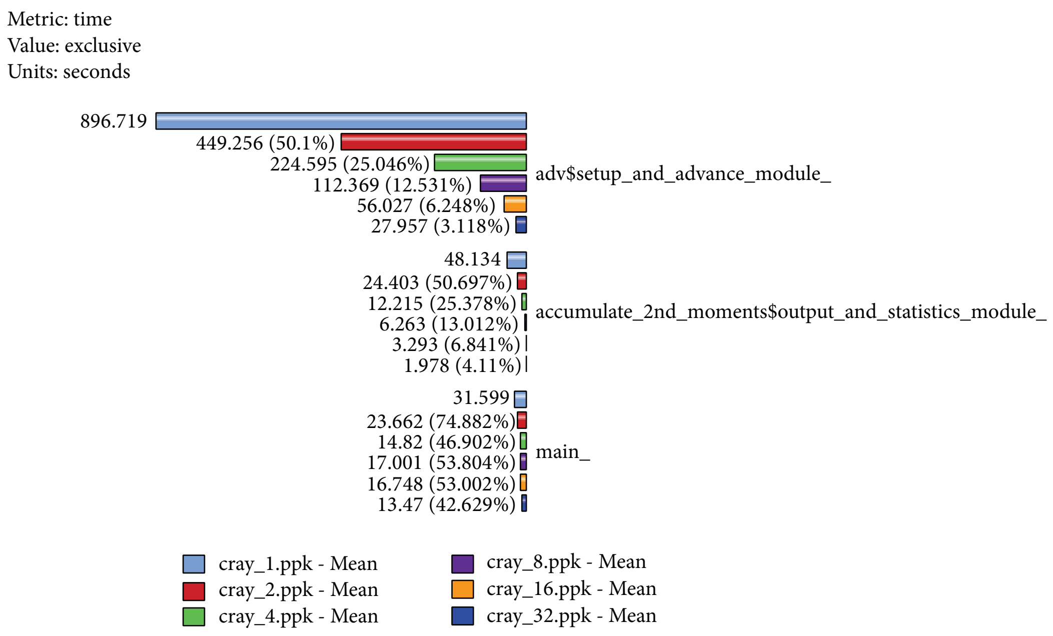


Figure 11: TAU profiling analysis of function runtimes when using the Cray native co_sum routines with 1, 2, 4, 8, 16, and 32 images.

doubling the number of bands, the number of processors must be quadrupled to have the same execution time. The scaling efficiency for the larger problem drops because of memory requirements; the objects fit in the heap and must be swapped out as needed, increasing the execution time.

6. Conclusions and Future Work

We demonstrated a strategy for parallelizing legacy Fortran 77 codes using Fortran 2008 coarrays. The strategy starts with constructing extensible tests using Fortran's OOP features. The tests check for regressions in accuracy and performance. In the PRM case study, our strategy expanded two Fortran 77 codes by 10% and 40%, exclusive of the test and I/O infrastructure. The most significant code revision involved unrolling two nested loops that distribute particles across images. The resulting parallel code achieves even load balancing but poor scaling. TAU identified the chief bottleneck as a sequential summation scheme.

Based on these preliminary results, we rewrote our co_sum procedure, and the speedup showed marked improvement. We also benchmarked the native co_sum implementation available in the Cray compiler. Our results show that the natively supported collective procedures show the best scaling performance even when using distributed memory. We hope that future native support for collective

Table 2: Runtime in seconds for parallel using 128 bands, and different collective sum routines.

	Number of Images					
	1	2	4	8	16	32
Intel Serial co_sum	35.55	19.80	11.69	9.73	18.71	66.82
Intel Parallel co_sum	37.30	19.33	10.00	6.17	4.62	5.41
Cray Native co_sum	46.71	23.68	11.88	6.06	3.06	1.73

Table 3: Weak scaling performance of coarray version.

Number of images	Number of bands	Number of particles	Particles per image	Time in seconds	Runtime per particle	Efficiency
1	128	33024	33024	44.279	1.34×10^{-3}	1.000
4	256	131584	32896	44.953	1.37×10^{-3}	0.978
16	512	525312	32832	49.400	1.50×10^{-3}	0.893
2	256	131584	65792	101.03	1.54×10^{-3}	1.000
8	512	525312	65664	102.11	1.56×10^{-3}	0.987
32	1024	2099200	65600	129.75	1.98×10^{-3}	0.777

procedures in Fortran 2015 by all the compilers will bring such performance to all platforms.

Conflict of Interests

The authors declare that there is no conflict of interests regarding the publication of this paper.

Acknowledgments

The initial code refactoring was performed at the University of Cyprus with funding from the European Commission Marie Curie ToK-DEV grant (Contract MTKD-CT-2004-014199). Part of this work was also supported by the Cyprus Research Promotion Foundation's Framework Programme for Research, Technological Development and Innovation 2009-2010 (ΔΕΣΜΗ 2009-2010) under Grant ΤΠΕ/ΠΛΗΡΟ/0609(ΒΕ)/11. This work used resources of the National Energy Research Scientific Computing Center, which is supported by the Office of Science of the U.S. Department of Energy under Contract no. DE-AC02-05CH11231. This work also used hardware resources from the ACISS cluster at the University of Oregon acquired by a Major Research Instrumentation grant from the National Science Foundation, Office of Cyber Infrastructure, "MRI-R2: Acquisition of an Applied Computational Instrument for Scientific Synthesis (ACISS)," Grant no. OCI-0960354. This research was also supported by Sandia National Laboratories a multiprogram laboratory operated by Sandia Corporation, a Lockheed Martin Company, for the National Nuclear Security Administration under Contract DE-AC04-94-AL85000. Portions of the Sandia contribution to this work were funded by the New Mexico Small Business Administration and the Office of Naval Research.

References

[1] B. L. Achee and D. L. Carver, "Creating object-oriented designs from legacy FORTRAN code," *Journal of Systems and Software*, vol. 39, no. 2, pp. 179–194, 1997.

[2] C. D. Norton and V. K. Decyk, "Modernizing Fortran 77 legacy codes," *NASA Tech Briefs*, vol. 27, no. 9, p. 72, 2003.

[3] C. Greenough and D. J. Worth, "The transformation of legacy software: some tools and processes," Tech. Rep. TR-2004-012, Council for the Central Laboratory of the Research Councils, Rutherford Appleton Laboratories, Oxfordshire, UK, 2004.

[4] F. G. Tinetti and M. Méndez, "Fortran Legacy software: source code update and possible parallelisation issues," *ACM SIGPLAN Fortran Forum*, vol. 31, no. 1, pp. 5–22, 2012.

[5] S. C. Kassinos and W. C. Reynolds, "A particle representation model for the deformation of homogeneous turbulence," in *Annual Research Briefs*, pp. 31–61, Center for Turbulence Research, Stanford University, Stanford, Calif, USA, 1996.

[6] S. C. Kassinos and E. Akylas, "Advances in particle representation modeling of homogeneous turbulence. from the linear PRM version to the interacting viscoelastic IPRM," in *New Approaches in Modeling Multiphase Flows and Dispersion in Turbulence, Fractal Methods and Synthetic Turbulence*, F. Nicolleau, C. Cambon, J.-M. Redondo, J. Vassilicos, M. Reeks, and A. Nowakowski, Eds., vol. 18 of *ERCOFTAC Series*, pp. 81–101, Springer, Dordrecht, The Netherlands, 2012.

[7] S. S. Shende and A. D. Malony, "The TAU parallel performance system," *International Journal of High Performance Computing Applications*, vol. 20, no. 2, pp. 287–311, 2006.

[8] M. Metcalf, J. K. Reid, and M. Cohen, *Modern Fortran Explained*, Oxford University Press, 2011.

[9] H. Radhakrishnan, D. W. I. Rouson, K. Morris, S. Shende, and S. C. Kassinos, "Test-driven coarray parallelization of a legacy Fortran application," in *Proceedings of the 1st International Workshop on Software Engineering for High Performance Computing in Computational Science and Engineering*, pp. 33–40, ACM, November 2013.