Test-driven coarray parallelization of a legacy Fortran application

Hari Radhakrishnan
EXA High Performance Computing, Nicosia, Cyprus
hari@exahpc.com

Damian W. I. Rouson
Center for Computational Earth and Environmental Sciences
Stanford University, Stanford, California, USA
rouson@stanford.edu

Karla Morris
Combustion Research Facility
Sandia National Laboratories, Livermore, California, USA
knmorris@sandia.gov

Sameer Shende
Performance Research Laboratory
University of Oregon, Eugene, Oregon, USA
sameer@cs.uoregon.edu

Stavros C. Kassinos
UCY-CompSci
University of Cyprus, Nicosia, Cyprus
kassinos@ucy.ac.cy

ABSTRACT

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 multi-core processors and many-core accelerators in shared and distributed memory. We delineate 17 code modernization steps used to refactor and parallize the program, and study the resulting performance. Our scaling studies show that the bottleneck in the performance was due to the implementation of the collective sum procedure. Replacing the sequential procedure with a binary tree procedure improved the scaling performance of the program. This bottleneck will be resolved in the future by new collective procedures in Fortran 2015.

1. INTRODUCTION

1.1 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 manycore accelerators, one might reasonably label any serial code as "legacy software." 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 manycore architectures in shared and distributed memory.

The published 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 [6], for example, 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.

Grenough and Worth [2] surveyed tools that enhance software quality by helping to detect errors and to highlight poor practices. The appendices of their report paper provide extensive summaries of the tools available from eight vendors with a very wide range of capabilities. A sample of these capabilities include memory leak detection; auto-

SEHPCCSE'13 November 22, 2013, Denver, CO, USA

matic 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 [8]. 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.

1.2 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 Equation 1 into a momentum balance and then averaging over an ensemble of turbulent flows yields the 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 i^{th} and j^{th} cartesian components of $\mathbf{u}$; and x_i and x_j are the i^{th} and j^{th} cartesian components of the spatial coordinate $\mathbf{x}$.

The term $-\rho\overline{u_i'u_j'}$ in Equation 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 $\partial\overline{u_j}/\partial x_i$. 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) [3, 4] addresses this shortcoming. Given sufficient computing resources, PRM exactly predicts the response of the fluctuating velocity field to rapid deformations.

The PRM is implemented by distributing 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. 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 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, i.e., all the possible flows considered, yields a representation of the 3D behavior in an actual flowing fluid.

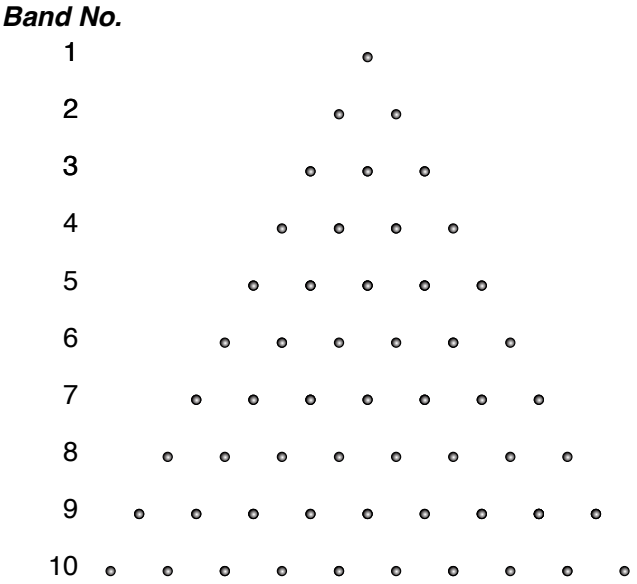


Figure 1: Distribution of particles in bands in one octant. Total number of particles is $NP = 2 \times NB \times (NB + 1)$.

Historically, a key disadvantage of the PRM has been costly execution times. Parallelization reduces this cost. Previous attempts to parallelize the PRM using MPI were abandoned because the development, validation and verification times did not justify the gains. Co-arrays 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.

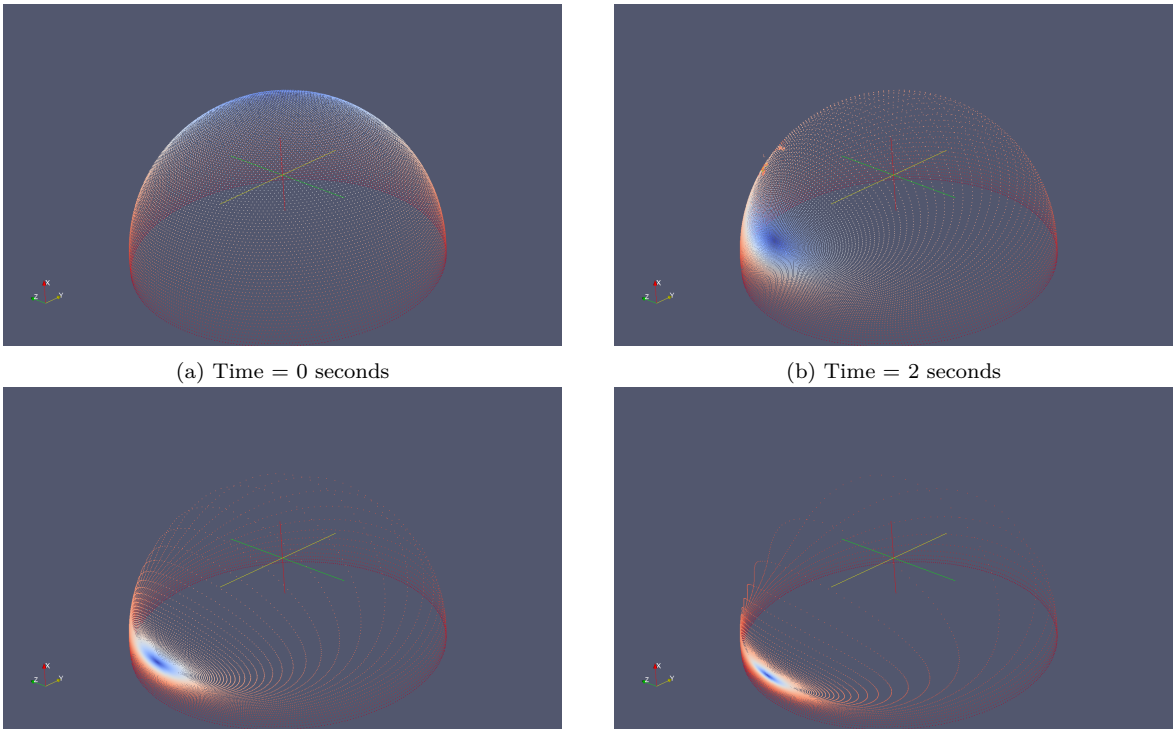
2. METHODOLOGY

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 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 setup 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.



(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.

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, and 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`," i.e., free of side effects, including input/output, modifying arguments, halting execution, or modifying non-local 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.

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 PRM 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.)

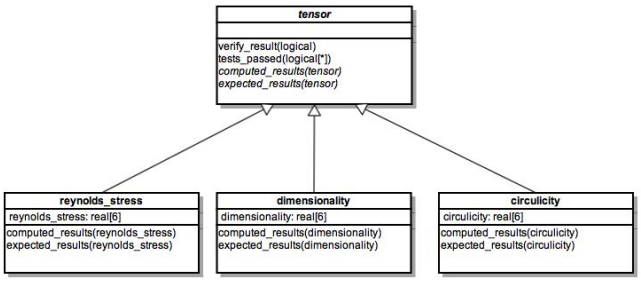


Figure 3: Class diagram of the testing framework. The deferred bindings are shown in italics, and the abstract class is shown in bold italics.

Our accuracy tests examine tensor statistics that PRM calculates. In order to establish a uniform protocol for running tests, we defined an abstract base `tensor` class. 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 child classes 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 in shown in Figure 3. The tests then take the form

```
if (.not. stess_tensor%verify_result(when)) &
```

Step	Details
1	Set up automated builds via CMake[a] and version control via Git[b].
2	Convert fixed- to free-source format via "convert.f90"by Metcalf[c].
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[d].
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 201x collective procedure to gather statistics.
17	Study and tune performance with TAU[e].

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

[a]http://www.cmake.org
[b]http:/git-scm.com
[c]ftp://ftp.numerical.rl.ac.uk/pub/MandR/convert.f90
[d]http://www.cmake.org
[e]http://tau.uoregon.edu

```
    error stop 'Test failed.'
```

where `&` is Fortran's line continuation character; `stress_tensor` is an instance of a class that extends `tensor`; "`error stop`" halts all images and prints the shown string to standard error; and `verify_result` invokes two aforementioned deferred bindings to compare the computed results to the expected results.

4. COARRAY PARALLELIZATION

Modern HPC software must execute on multicore processors or manycore 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 [5]. The Fortran 2008 compilers then 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]
```

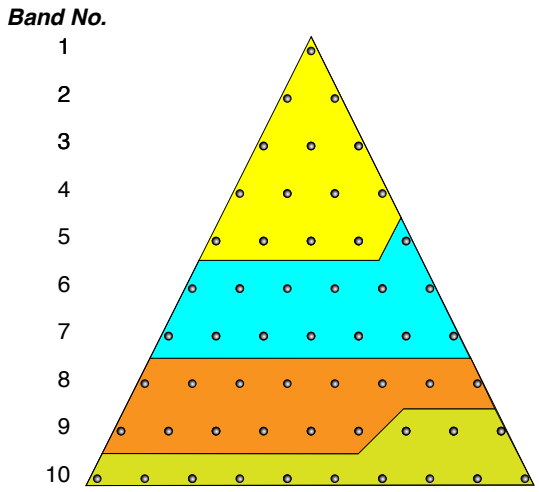
copies the first element of image 2 to the first element of whatever image executes this line. The ability to omit the coindex on the left-hand side (LHS) played a pivotal role in refactoring the PRM with minimal work: although we added codimensions to existing variables' declarations, subsequent accesses to those variables remained unmodified except where it is desired communicate across images. 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 below.

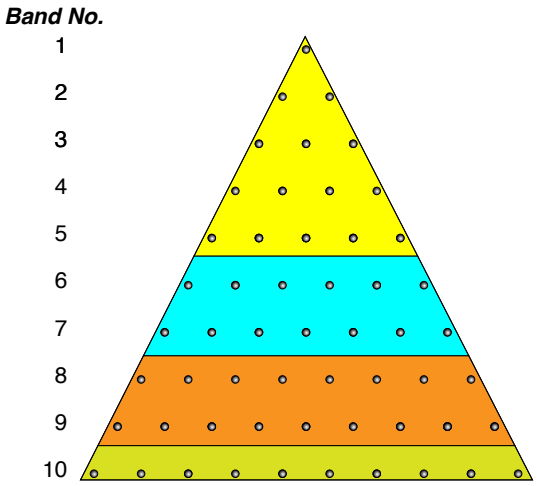
```
l = 0 ! Global particle number
do k = 1, nb ! Loop over the bands
    do m = 1,k ! Loop over the particles in band

        ! 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
```

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 distribu-



(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.

tion is shown in Figure 4a. 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 below. However in this case, the code becomes complex and sensitive to precision.

```
! 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
```

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 4b. Method 2 requires fewer changes to the original code but is sub-optimal in load balancing.

```
! 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
```

5. RESULTS

Source code impact: We applied our strategy to two versions of 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, PRM expanded 40% from 903 lines to 1260 lines, not including new input/output (I/O) code and the code described in the Object-Oriented Test Suite section of this paper. The test and I/O code occupied an additional 569 lines.

Compare and Contrast with MPI: The ability to drop the coindex from the notation was a big help in parallelizing the program without making significant changes to the source code. A lot of the book-keeping is handled behind the scenes by the compiler making it possible to make the parallelization more abstract but also more easier to follow. For example, these are the code fragments necessary to collect the local arrays into a single global array using the coarray syntax:

```
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
```

The equivalent code fragment in MPI:

```
integer :: my_rank, num_procs
integer, allocatable :: 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)
```

Reducing the complexity of the code also reduces the chances of bugs in the code. In the legacy code, the 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 PRM logic. This made it significantly easier to ensure that the refactoring did not alter the results of the model. The main changes that were implemented were to define global arrays for *sn* and *cr*, and update them when needed, as shown above.

Scalability: We intend for PRM to serve as an alternative to turbulence models used in routine engineering design of fluid devices. Most 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.

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[1]. Each node has four Intel X7560 2.27GHz 8-core CPUs and 384GB of DDR3 memory. We

[1]http://aciss.uoregon.edu

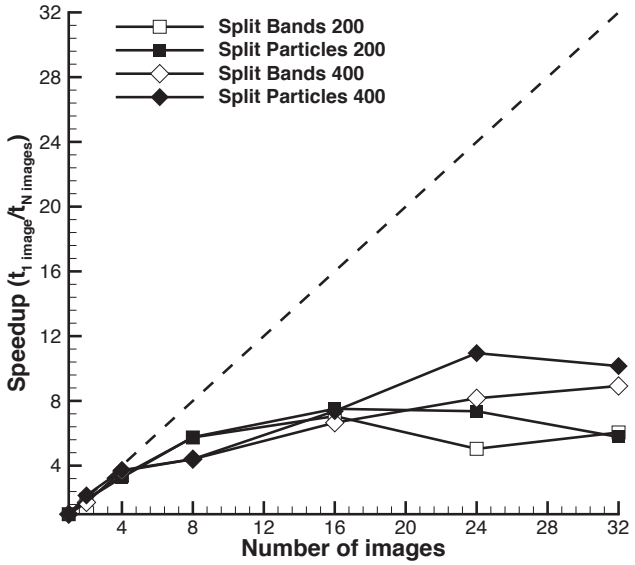


Figure 5: Speedup obtained with sequential co_sum implementation using multiple images on a single server with 384 GB of RAM and Intel X7560 2.27GHz 8-core CPUs.

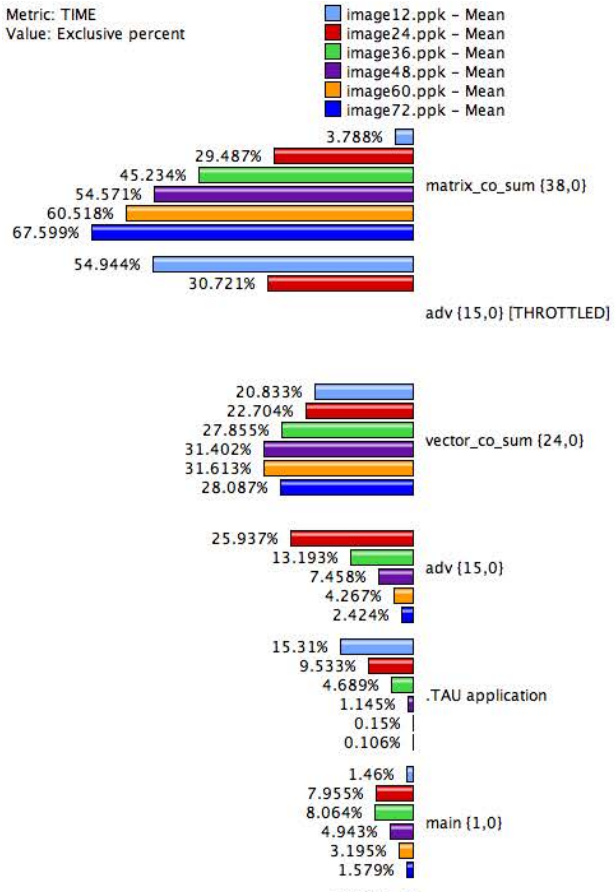


Figure 6: TAU analysis of load balancing and bottlenecks.

N Images	N Particles	N Particles Per Image	Time (secs)
1	8320	8320	44.279
4	33024	8256	44.953
16	131584	8224	49.400
2	131584	65792	355.835
8	525312	65664	364.091
32	2099200	65600	370.000

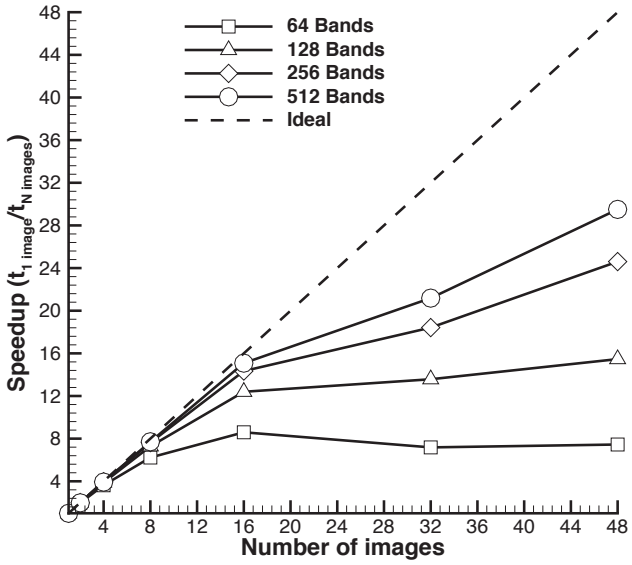


Figure 7: Speedup obtained with parallel co_sum implementation using multiple images on a single server with 64 GB of RAM and four AMD Opteron 6238 Processors, each with 12 cores.

see that the speedup was very poor when the number of processors were increased.

We used TAU [7] to profile the parallel runs on the "basic" nodes of the ACISS[2] to understand the bottlenecks during execution. Each node has two X5650 6-core CPUs at 2.67GHz, and 72GB DDR3 memory. Figure 6 shows the TAU plot for the runtime share for the dominant procedures using different number of images. We identified the chief bottlenecks to be the 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 is $O(N_{images})$. 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 $O(logN_{images})$ in time.

Table 2 shows the weak scaling performance of the program using the binomial tree collective sum procedures. The number of particles as shown in Figure 1 scales as the square of the number of bands. Therefore, when doubling the number of bands, the number of processors must be quadrupled to have the same execution time.

Figure 7 shows the speedup obtained for different number of bands using the parallel `co_sum` algorithms with the Intel Fortran compiler using the first scheme (uniform distribution of the particles) described in the Coarray Parallelization section. These tests were run on a single server that had four AMD Opteron 6238 12-core CPUs and 64 GB of RAM. The results show that eliminating the bottleneck has improved the scaling performance of the program. Also the scaling efficiency increases when the problem size is increased which shows that the poor scaling at smaller problem sizes is due to communication and synchronization.

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. 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 improved. We are currently benchmarking alternative summation algorithms, including a `co_sum` implementation available in the Cray compiler. However our work has been stalled due to some compiler/runtime bugs when using distributed memory coarrays. We expect to complete this work soon, and present the full set of benchmarking and optimization results.

Acknowledgements

This work 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 #: OCI-0960354.

Sandia is a multi-program 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.

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(BE)/11.

References

[1] B. L. Achee and D. L. Carver. Creating object-oriented designs from legacy fortran. *Journal of Systems and Software*, 37:179–194, 1997.

[2] http://aciss.uoregon.edu

[2] C. Greenough and D. J. Worth. The transformation of legacy software: Some tools and processes. Technical Report TR-2004-012, Council for the Central Laboratory of the Research Councils, Rutherford Appleton Laboratories, Chilton, Didcot, Oxfordshire, UK, 2004.

[3] S. Kassinos and E. Akylas. Advances in particle representation modeling of homogeneous turbulence. from the linear prm version to the interacting viscoelastic iprm. In F. Nicolleau, C. Cambon, J.-M. Redondo, J. Vassilicos, M. Reeks, and A. Nowakowski, editors, *New Approaches in Modeling Multiphase Flows and Dispersion in Turbulence, Fractal Methods and Synthetic Turbulence*, volume 18 of *ERCOFTAC Series*, pages 81–101. Springer Netherlands, 2012.

[4] S. C. Kassinos and W. C. Reynolds. A particle representation model for the deformation of homogeneous turbulence. In *Annual Research Briefs*, pages 31–61, Center for Turbulence Research, Stanford University, Stanford, CA, 1996.

[5] M. Metcalf, J. K. Reid, and M. Cohen. *modern fortran explained*. Oxford University Press, 2011.

[6] C. D. Norton and V. K. Decyk. Modernizing fortran 77 legacy codes. *NASA Tech Briefs*, 27(9):72, 2003.

[7] S. Shende and A. D. Malony. Tau parallel performance system. *International Journal of High Performance Computing Applications*, 20(2):287–311, Summer 2006.

[8] F. G. Tinetti and M. Méndez. Fortran legacy software: Source code update and possible parallelisation issues. *ACM Fortran Forum*, 31(1), April 2012.