Test-driven parallelization of a legacy Fortran program

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

1. Center for Computational Earth and Environmental Sciences, Stanford University, Stanford, California, USA
2. UCY-COMPSCI, Dept. of Mechanical and Manufacturing Engineering, University of Cyprus, Nicosia, Cyprus
3. Combustion Research Facility, Sandia National Laboratories, Livermore, California, USA
4. Performance Research Laboratory, University of Oregon, Eugene, Oregon, USA

Abstract

This poster describes the steps involved in modernizing a Fortran 77 turbulent flow model using the object-oriented (OO) and coarray parallel programming features of Fortran 2003 and 2008, respectively. OO programming (OOP) facilitates the construction of an extensible suite of model-verification and performance tests that drive the development. Fortran's new coarray parallel programming feature set facilitates 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.

The modernization strategy involves 17 steps from automating the building and testing process to adding version control, transforming the source code, and analyzing the performance with the Tuning and Analysis Utilities (TAU) [1]. The verification tests ensure program correctness after each source-transformation step. The performance tests guard against execution-time regressions. The ultimate steps of parallelization via the coarray syntax demonstrates the approach's minimal impact on the source code.

Background

Legacy software is commonly defined as 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 "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 poster presents two strategies for parallelizing a legacy Fortran code by offering a clear path to execution on multicore and many-core architectures in shared and distributed memory while bolstering trust in the result.

The published literature on modernizing legacy Fortran codes focuses primarily on programmability issues such as increasing type safety and modularization while reducing data dependancies via encapsulation and information hiding [2,3,4]. One recent study applied automated code transformations in preparation for possible shared-memory, loop-level parallelization with OpenMP [5]. We are aware of no published studies on employing the Fortran 2008 coarray parallel programming features to refactor a serial Fortran 77 application.

Case Study: PRM

Turbulent flow theories typically attempt to predict the statistics of a fluid velocity vector field after decomposing it into its mean and fluctuating constituents:

$$\vec{u} \equiv \langle \vec{u} \rangle + \vec{u}'$$

The Particle Representation Model (PRM) describes flows in which these statistics are spatially homogeneous [6]. PRM provides exact predictions of turbulent flow regimes for which a mean velocity gradient of sufficient magnitude causes fluctuating velocity statistics to evolve purely under the action of the gradient without time for the fluctuating field to interact with itself.

As shown in Figure 1, the PRM tracks particles on a unit hemisphere as proxies for one-dimensional, one-component flows. The PRM initially distributes particles in bands on each octant of the hemisphere. Figure 2 shows ten flattened bands. Each particle has properties such as velocity, vorticity, stream function, vector orientation, and pressure. Averaging the properties of all the particles predicts the statistical behavior of an actual 3D flow.

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

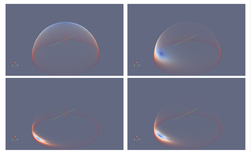

Figure 1. PRM particles for statistically homogeneous turbulence undergoing uniform shear. The images are organized chronologically clockwise with the upper left image being the initial condition. Color indicates initial position: blue (pole) and red (equator).

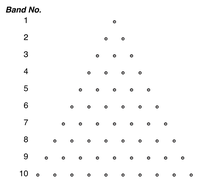

Figure 2. Flattened band structure.

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 shows 17 steps we employed in refactoring the PRM. The objectives of the different groups of tests were

- Step 1: Construct an infrastructure for automated building, testing, and version control.
- Steps 2 – 7: Refactor the Fortran 77 code to leverage standard Fortran 90 capabilities and to remove obsolete and deprecated features.
- Steps 8,9: Setup automated testing using the Fortran 77 code as a benchmark.
- Steps 10 – 12: Expose optimization opportunities to the compiler.
- Steps 13 – 17: Parallelize, analyze, and tune the code's performance.

Step 16 involves writing a collective reduction procedure "co_sum" to circumvent the Intel compiler's lack of support for this likely feature of the upcoming Fortran 2015 standard.

Table 1. Modernization steps employed in refactoring the PRM model. Horizontal lines are used to group the tests that must all be completed before proceeding further. Within a group of tests, no specific ordering is implied.

Step	Details
1	Set up automated builds via CMake and version control via Git.
2	Convert fixed- to free-source format using 'convert.f90' by Metcalf.
3	Replace `GOTO` with `do while` for main loop control.
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.
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.

Extensible OO Test Suite

After Step 9, we ran a suite of accuracy tests at every step to verify that the results of a representative simulation did not deviate from the legacy version'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 version did not increase more than 20% over that of the legacy version.

Our accuracy tests examine tensor statistics that are computed by PRM. In order to establish a uniform protocol for running tests, we defined the abstract base class tensor shown below.

```
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
```

With this infrastructure in place, specific tests take the form of child classes that extend the tensor class and thereby inherit a responsibility to implement the tensor class's two deferred bindings named `computed_results` and `expected_results`. The tests then take the form

```
if (.not. stress_tensor%verify_result(when)) error stop 'Test failed.'
```

where `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 `when` is a time stamp.

Coarray Parallelization

Fortran 2008 is a partitioned global address space (PGAS) language that supports the single-program, multiple-data (SPMD) parallelization scheme [7].

A coarray declaration of the form

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

facilitates indexing into the variable "a" along three 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 whichever image executes this line. The ability to omit the coindex on the left-hand side 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 to communicate across images. When necessary, adding coindices facilitated the construction of collective procedures to compute statistics.

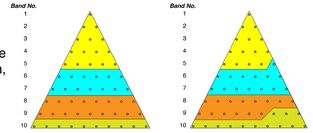

Figure 3. Distribution of particles across four images using two different partitioning methods: 1) Partitioning at the band boundaries to minimize code changes, and 2) Partitioning across bands to improve load balancing.

```
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
```

Legacy nested loop used to compute particle properties

```
do k = my_first_band, my_last_band ! Loop over the bands
   l = k**2 + (k-1)**2 ! Global number of first particle in kth band
   do m = 1,k ! Loop over the particles in band

      ... Computations ...

   end do
end do
```

Parallel nested loop that corresponds to the legacy nested loop shown on left when the bands are distributed across images (Method 1).

```
do l = my_first_particle, my_last_particle ! Loop over the particles
   k = nint(sqrt(real(l)*0.5))
   m = (l - (1 + 2*k*(k-1) -k))/4

      ... Computations ...

end do
```

Single loop that corresponds to the legacy nested loop shown on left when the particles are distributed across images (Method 2).

In the legacy version, the computations of the particle properties were done using two nested loops. As shown on the left, an outer loop ranges across bands of particles. An inner loop ranges within bands. We tried two ways of distributing the particles across the images, as shown in Figure 3. Method 1 places image boundaries only between bands. Method 2 works with the particles directly, splitting them evenly across all the images, allowing image boundaries to occur in the middle of a band.

Method 1 requires fewer changes to the original code. However, this distribution is sub-optimal. Method 2 facilitates optimal work distribution at the cost of increased sensitivity to precision and more work in replacing two nested loops with one loop over the number of particles and a reverse computation of the band number and the particle number within a band from the global particle number. The speedup obtained using the two different parallelization algorithms is shown in the Results section.

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 poster. The test and I/O code occupied an additional 569 lines.

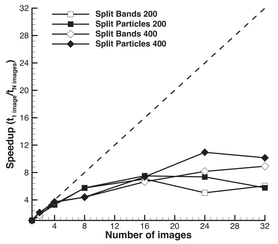

Figure 4: Scaling using the two different partitioning schemes and the sequential co_sum routine. The runs were done on a single server with four Intel 8-core CPUs and 384 GB RAM.

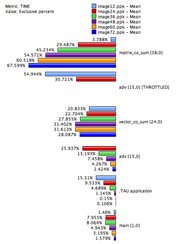

Figure 5: Instrumentation using TAU revealed that the co_sum routine was the bottleneck in achieving good speedup.

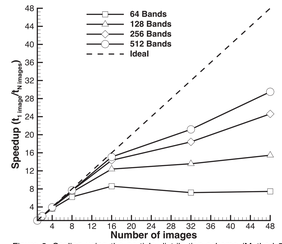

Figure 6: Scaling using the particle distribution scheme (Method 2) and the parallel co_sum routine. The runs were done on a single server with four AMD 12-core processors and 64 GB RAM.

Scalability: Figure 4 shows the speedup we obtained with the Intel Fortran compiler using the two particle-distribution schemes described in the Coarray Parallelization section of this poster. The plot includes data from runs with 200 and 400 bands of particles on the "fatnodes" of ACISS (http://aciss.uoregon.edu). Each node has four Intel X7560 2.27GHz 8-core CPUs and 384GB of DDR3 memory.

Figure 5 shows a TAU plot of each image's runtime share for the dominant procedures which shows that the chief bottlenecks are collective co_sum procedures. The Fortran standards committee is working on a co_sum intrinsic procedure that will likely become part of Fortran 2015.

We implemented a parallel version of the co_sum procedure using a binomial tree algorithm to see its impact on scaling. Figure 6 shows that we approach ideal, linear speedup as we increase the number of bands.

Conclusions and Future Work

We have demonstrated a strategy for parallelizing legacy Fortran 77 codes using Fortran 2008 coarrays. The strategy starts with building extensible OO tests that check for regressions in accuracy or 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 identifies the chief bottleneck: a sequential summation. A simple parallel implementation of the co_sum routine significantly improved the speedup of the program. We believe that native implementations of collective algorithms such as sum, min, max, etc. will significantly boost the adoption of Fortran 2008 coarrays. Future work will involve evaluating alternative summation algorithms, including a co_sum implementation natively available in the Cray compiler.

Acknowledgements

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). 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.

References

[1] S. Shende and A. D. Malony, TAU Parallel Performance System, I*nternational Journal of High Performance Computing Applications*, Vol. 20(2), pp. 287-311, Summer 2006. http://tau.uoregon.edu
[2] C. D. Norton and V. K. Decyk. Modernizing Fortran 77 Legacy Codes. *NASA Tech Briefs 27* (9), 72, 2003.
[3] B. L. Achee and D. L. Carver. Creating object-oriented designs from legacy Fortran. *Journal of Systems and Software*, 37:179–194, 1997.
[4] C. Greenough and D. J. Worth. *The transformation of legacy software: Some tools and processes.* Council for the Central Laboratory of the Research Councils, Rutherford Appleton Laboratories, Chilton, Didcot, Oxfordshire, UK, TR-2004-012.
[5] F. G. Tinetti and M. Méndez. Fortran Legacy Software: Source Code Update and Possible Parallelization Issues. *ACM Fortran Forum*, April, 31(1), pp. 5-22, 2012.
[6] S. C. Kassinos and W. C. Reynolds. A particle representation model for the deformation of homogeneous turbulence. *Annual Research Briefs*, Center for Turbulence Research, Stanford University, Stanford, California, USA, 31-61, 1996.
[7] M. Metcalf, J. Reid, and M. Cohen. *Modern Fortran Explained.* Oxford University Press, 2011.