Test-driven parallelization of a legacy Fortran program

Damian Rouson
CEES, Stanford University,
Stanford, California, USA
http://cees.stanford.edu

Hari Radhakrishnan and
Stavros Kassinos
UCy-CompSci, University of
Cyprus, Nicosia, Cyprus
http://ucy-compsci.org/

Karla Morris
Combustion Research Facility,
Sandia National Laboratories,
Livermore, California, USA
http://crf.sandia.gov

Sameer Shende
University of Oregon, Eugene,
Oregon, USA
http:
//ix.cs.uoregon.edu/~sameer/

ABSTRACT

This poster describes a strategy for using the object-oriented (OO) and coarray features of Fortran 2003 and 2008, respectively, to parallelize a legacy Fortran 77 program. 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 manycore accelerators in shared and distributed memory. We delineate 17 code modernization steps, study the resulting performance, and identify one bottleneck that should be resolved 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 "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.

We present 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. One recent study applied automated code transformations in preparation for possible shared-memory, loop-level parallelization with OpenMP. We are aware of no published studies on using Fortran 2008 coarrays to refactor and parallelize a serial Fortran 77 application.

1.2 Case Study: PRM

Most turbulence theories estimate the statistics of a fluid velocity vector field after decomposing it into its mean and fluctuating constituents:

$$\vec{u} = \langle \vec{u} \rangle + \vec{u}' \tag{1}$$

The Particle Representation Model (PRM) describes flows in which these statistics are spatially homogeneous [1]. The PRM tracks particles on a unit hemisphere as proxies for hypothesized one-dimensional, one-component flows. The PRM initially distributes particles in bands on each octant of the hemisphere. Each particle has properties that describe the dynamics of one hypothetical flow. 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. Previous attempts to reduce execution time by parallelizing 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. The OO test suite facilitated a continuous build-and-test cycle that reduced the development time.

2. METHODOLOGY

2.1 Modernization Strategy

Our poster tabulates 17 steps involved in refactoring the PRM. The steps are grouped based on their objectives. While each group of steps must be completed before proceeding

SC'13 November 17-22, 2013, Denver, CO, USA

to the next group, no ordering of steps is implied within a group. The objectives of the different groups of steps were:

Step 1: Construct infrastructure for automated building and testing, and version control.

Steps 2-7: Refactor Fortran 77 code to Fortran 90 standard and remove obsolete and deprecated features.

Steps 8,9: Setup automated testing using Fortran 77 legacy code as benchmark.

Steps 10-12: Expose optimization opportunities to compiler.

Steps 13-17: Parallelize using Fortran 2008 coarrays; analyze and tune code performance.

2.2 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 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% expecting that significant speedup when running multiple images will compensate this overhead.

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. 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 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 `verify_result` invokes two aforementioned deferred bindings to compare the computed results to the expected results.

2.3 Coarray Parallelization

Coarray Fortran uses a partitioned global address space (PGAS) with the single program, multiple data (SPMD) parallelization scheme.

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 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 communication across images was needed. When necessary, adding coindices facilitated the construction of collective procedures to compute statistics.

The legacy PRM initially distributes particles in two nested loops: an outer loop over bands of particles and an inner loop within each band. Unrolling these into one loop over all the particles increases the code's scalability. We provided two new ways to achieve this distribution. Method 1 works with the bands, splitting between bands to divide the particles across images as evenly as possible. Method 2 works with the particles directly, splitting them within bands to achieve a more even particle distribution across images.

Method 1 requires fewer changes to the original code but is sub-optimal in load balancing. Method 2 requires replacing two nested loops with one loop over the number of particles and splitting the particles evenly across the images. This involves a reverse computation of the band number and the particle number within a band from the global particle number. In this case, the code in the above example becomes complex and sensitive to precision.

3. RESULTS

Source code impact: We applied our strategy to two PRM versions. For one, 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 section 2.2 of this paper. The test and I/O code occupied an additional 569 lines.

Scalability: The poster shows the speedup obtained with the Intel Fortran compiler using the two particle-distribution schemes on a single server with four Intel 8-core CPUs and 384GB RAM. Using the TAU performance tools[1] to analyze each image's runtime share exposed a bottleneck in a sequential `co_sum` collective reduction procedure. 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. To test the impact of a parallel `co_sum` routine, we implemented one using a binomial tree algorithm. The poster shows the resulting improvement.

4. 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 identifies the chief bottleneck as a sequential summation scheme. Future work will involve evaluating alternative summation algorithms, including a `co_sum` implementation available in the Cray compiler.

References

[1] 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.

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