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

1EXA High Performance Computing, 1087 Nicosia, Cyprus
2Stanford University, Stanford, CA 94305, USA
3Sandia National Laboratories, Livermore, CA 94550, USA
4University of Oregon, Eugene, OR 97403, USA
5Computational 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

Copyright © 2015 Hari Radhakrishnan et al. This is an open access article distributed under the Creative Commons Attribution License, which permits unrestricted use, distribution, and reproduction in any medium, provided the original work is properly cited.

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, \( \bar{\mathbf{u}} \), and a fluctuating part, \( \mathbf{u}' \):

\[
\mathbf{u} = \bar{\mathbf{u}} + \mathbf{u}'.
\]  

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

\[
\rho \frac{\partial \bar{\mathbf{u}}}{\partial t} + \rho \frac{\partial \mathbf{u}'}{\partial t} \left[ -\bar{\rho} \delta_{ij} + \mu \left( \frac{\partial \bar{\mathbf{u}}_i}{\partial x_j} + \frac{\partial \bar{\mathbf{u}}_j}{\partial x_i} \right) - \rho u'_i u'_j \right],
\]

where \( \mu \) is the fluid's dynamic viscosity; \( \rho \) 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 \( -\bar{\rho} \delta_{ij} \) 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 \( \frac{\partial \mathbf{u}}{\partial x} \) 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 \( \bar{\mathbf{u}} \). 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 I for ten bands. The total number of particles is given by

\[
N_{\text{particles}} = \frac{4 \times \text{Number of octants in hemisphere}}{\text{Number of particles in one octant}} \times \frac{\text{Number of octants in hemisphere}}{2} \times (N_{\text{bands}} + 1).
\]

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

\textsuperscript{1}http://www.cmake.org/.
\textsuperscript{2}http://git-scm.com/.
\textsuperscript{3}ftp://ftp.numerical.rl.ac.uk/pub/MandR/convert.f90.
\textsuperscript{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 \texttt{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 \texttt{co_sum} procedure.) The final step involves performance analysis using the Tuning and Analysis Utilities \cite{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, \texttt{reynolds\_stress}, dimensionality, and circulicity, that extend the tensor class and thereby inherit a responsibility to implement the tensor's deferred bindings \texttt{compute\_results} and \texttt{expected\_results}. The class diagram is shown in Figure 3. The tests then take the form

\begin{verbatim}
if (.not. stress\_tensor\%verify\_result (when)) &
    error stop 'Test failed.'
\end{verbatim}

where \texttt{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; \texttt{error stop} halts all images and prints the shown string to standard error; and \texttt{verify\_result} is the \texttt{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 \cite{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

\begin{verbatim}
real, allocatable :: a (:, :, :) [:]
\end{verbatim}

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 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.

```fortran
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
  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
```

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

**Listing 1:** Base tensor class.

```fortran
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.
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

Listing 3: Parallel loop by splitting particles.

Listing 4: Parallel loop by splitting bands.
null
### Speedup (relative to single image)

<table>
<thead>
<tr>
<th>Number of coarray images</th>
<th>32</th>
<th>16</th>
<th>8</th>
<th>4</th>
<th>2</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Split bands 200</td>
<td>248</td>
<td>16</td>
<td>8</td>
<td>4</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>Split particles 200</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Split bands 400</td>
<td>16</td>
<td>8</td>
<td>4</td>
<td>2</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>Split particles 400</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

![Speedup Graph](image)

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

### subroutine `vector_co_sum_serial(vector)`

```fortran
real(kind), 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
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.

Metric: time
Value: exclusive
Units: seconds

Figure 9: TAU profiling analysis of function runtimes when using the optimized 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
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 ТΠΕ/ΠΑΗΡΟ/0609(ΒΕ)/II. 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, “MIR- 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
Submit your manuscripts at http://www.hindawi.com