Scientific Programming 15 (2007) 27–44
IOS Press

The transition and adoption to modern programming concepts for scientific computing in Fortran

Charles D. Norton[a], Viktor K. Decyk[b], Boleslaw K. Szymanski[c] and Henry Gardner[d]
[a]*Jet Propulsion Laboratory, California Institute of Technology, Pasadena, CA 91109-8099, USA*
[b]*Jet Propulsion Laboratory, California Institute of Technology, Pasadena, CA 91109-8099, USA and Department of Physics and Astronomy, University of California, Los Angeles, CA 90095-1547, USA*
[c]*Department of Computer Science, Rensselaer Polytechnic Institute, Troy, NY 12180-3590, USA*
[d]*Computer Science, FEIT, Australian National University, Canberra, ACT 0200, Australia*

Abstract. This paper describes our experiences in the early exploration of modern concepts introduced in Fortran90 for large-scale scientific programming. We review our early work in expressing object-oriented concepts based on the new Fortran90 constructs – foreign to most programmers at the time – our experimental work in applying them to various applications, the impact on the WG5/J3 standards committees to consider formalizing object-oriented constructs for later versions of Fortran, and work in exploring how other modern programming techniques such as Design Patterns can and have impacted our software development. Applications will be drawn from plasma particle simulation and finite element adaptive mesh refinement for solid earth crustal deformation modeling.

1. Introduction

We describe the challenges faced by the scientific community in early-90s for large-scale programming and the issues in deciding if legacy Fortran software should be abandoned and re-written in C++ or if the growing complexity of writing software in Fortran 77 would become unmanageable. A careful analysis of these challenges led us into "discovery" of how Fortran90 concepts shared common properties with object-oriented features. In this paper, we describe how we apply these methods for plasma particle-in-cell experiments to explore issues in data organization and performance. Another example of this methodology given in this paper arose in the area of Adaptive Mesh Refinement (AMR) implementation that many believed to be infeasible using Fortran due to complexities of data structure representation. The resulting implementation is an example of how modern Fortran can support adaptive methods while preserving investment in legacy solvers.

Finally, the paper includes a follow-on discussion describing how even more complex systems can be modeled with Design Patterns using Fortran. As computers become more powerful, there is a growing desire to make scientific codes increasingly complex. In the computer science literature, researchers have discovered that similar solutions to programming complex problems have appeared in different contexts, and they have called such solutions "design patterns." In this section we will discuss one such design pattern that has been useful for us in our goal of creating more ambitious Fortran95-based scientific programs. Some of the techniques used in implementing these design patterns are based on earlier work in emulating object-oriented concepts in Fortran95 [4,5]. In general, the patterns we have found most useful are those which in one place encapsulate the variation and control of some important capability in the code.

ISSN 1058-9244/07/$17.00 © 2007 – IOS Press and the authors. All rights reserved

2. Exploration of object-oriented concepts in Fortran90

The C++ programming language is well known for its support of object-oriented concepts useful in abstraction modeling. Containing many important features, its popularity grew in the early 1990's with a new generation of scientists anxious to bring clarity and flexibility to their programming efforts. Nevertheless, most of the scientific applications in development and in use at that time, and today, are based on Fortran – still the most popular language for scientific programming.

Fortran is not a static language; it has continually evolved, somewhat slowly, to include the most recent proven ideas and concepts garnered from other programming languages. Before Fortran90, modern aspects of software design were not supported which complicated abstraction modeling for large-scale development projects. This made Fortran77 software difficult to comprehend, often unsafe, and potentially problematic. The emergence of Fortran90 [11] dramatically altered traditional Fortran programming. Many modern programming language techniques were included in the standard with new features introduced to influence practical scientific programming [12]. These features extended far beyond the well-publicized array-syntax operations and reached into the object-oriented programming paradigm.

Although Fortran77 programs are completely compatible with the Fortran90 standard, Fortran90 was a very different language. The new constructs encouraged the creation of abstract data types, encapsulation, information hiding, inheritance, and generic programming and provided many features to ensure the safe development of advanced programs. These new features ushered in Fortran90 as a modern language whose benefits could be compared and evaluated on equal footing with other modern languages, including C++. This was one focus of our early work that demonstrated and proved how Fortran90 supports object-oriented programming concepts. We found that, sometimes, Fortran90 terminology matched that of other languages, even though the meaning of the terms differed. Incidentally, this became a challenge when we proposed mechanisms to convey and teach how to best use the Fortran90 features to both the C++ and Fortran77 communities that were actively debating how to develop new software while preserving their legacy codes and expertise.

The features of Fortran90 are quite powerful by themselves, even if the object paradigm is not applied. For experienced Fortran77 users looking to modernize their programs to enhance collaboration, usability, sharing, and software extension, one benefit is that the new aspects of Fortran90 can be incrementally introduced into existing Fortran77 programs. This allows programs to evolve in the context of a well-known environment, permitting scientific productivity to continue while new ideas are acquired. Unfortunately, there is always some risk associated with change – using Fortran90 most effectively requires a paradigm-shift. Some reluctance is natural given the investments in existing software and process of code development. Nevertheless Fortran90 is a standard, which when applied properly, can make scientific programming very clear and productive, particularly when object-oriented methodology is applied. Our initial work in exploring object-oriented concepts for modernizing Fortran77 codes began with plasma simulation and this is where we begin this paper.

3. Modern Fortran for plasma particle simulation

When a material is subjected to conditions under which the electrons are stripped from the atoms, acquiring free motion, the mixture of heavy positively charged ions and fast electrons forms an ionized gas called a plasma. Ionization can be introduced by extreme heat, pressure, or electric discharges. Fusion energy is an important application area of plasma physics research, but more familiar examples of plasmas include the Aurora Borealis, neon signs, the ionosphere, and solar winds. The plasma particle-in-cell simulation model [13] integrates in time the trajectories of millions of charged particles in their self-consistent electromagnetic fields. The method assumes that particles do not interact with each other directly, but through the fields that they produce. Particles can be located anywhere in the spatial domain; however, the field quantities are calculated on a fixed grid. In our example, only the electrostatic (coulomb) interactions are included.

The General Concurrent Particle-in-Cell (GCPIC) Algorithm [14] partitions the particles and grid points among the processors of the MIMD (multiple-instruction, multiple-data) distributed-memory parallel processing machine. The particles are evenly distributed among processors in the primary decomposition, which makes advancing particle

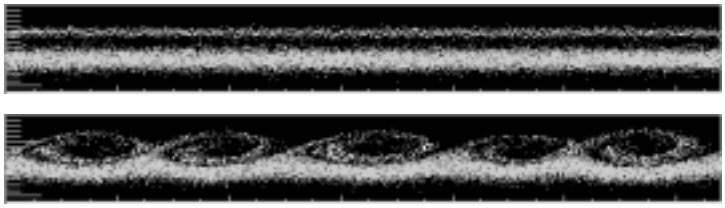


Fig. 1. Electron phase space of one-dimensional beam-plasma instability.

positions and velocities in space efficient. A secondary decomposition partitions the simulation space evenly among processors, which makes solving the field equations on the grid efficient. As particles move among partitioned regions they are passed to the processor responsible for the new region. For computational efficiency, field/grid data on the border of partitions are replicated on the neighboring processor to avoid frequent off-processor references. Particles scatter charge and gather force data to/from their nearest grid points. Electric field components from each dimension are required to advance particles to new positions.

A Beam-Plasma instability experiment models the injection of a low density electron beam into a stationary (yet mobile) background plasma of high density, driving plasma waves to instability. This is a special case of the more general two-stream instability where two streams of electrons flow through each other. Beam-Plasma interactions cause particle bunching, forming potential wells that are self-enhanced. This leads to particle trapping creating vortices in phase space. The ions are modeled as a fixed neutralizing background. Although the number of particles per processor will vary during this simulation, the load remains sufficiently well balanced. This is not the case for all kinds of plasma simulations where dynamic load balancing may be required. An experiment such as this can be used to verify plasma theories and to study the time evolution of macroscopic quantities such as potential and velocity distributions.

The initial state of a one-dimensional experiment is shown in Fig. 1, where the phase space diagrams plot position against velocity. The instability drives plasma waves to saturation. In this experiment, we only measure the energy diagnostic which shows how the kinetic energy is rapidly converted into field energy.

The routines of advancing particles and computing their charge deposition from the original Fortran77 one-dimensional program are:

```
dimension part(idimp,np), q(nx), fx(nx)
data qme,dt /-1.,.2/
call push1 (part,fx,qtme,dt,wke,idimp,np,nx)
call dpost1 (part,q,qme,np,idimp,nx)
```

where an array of particles (part), charge density field (q), and electric force field (fx) are used in the operations of pushing particles (push) and depositing charge (dpost1). Additional parameters include the charge on an electron (qme), the time step (dt), field dimension (nx), particle kinetic energy (wke), and number of particles in the electron species (np). Even in this example, the code would be clearer and easier to manipulate if particle and field information were encapsulated into logically related units:

```
USE plasma_module
TYPE (species) :: electrons, ions
TYPE (fields) :: charge_density, efield
real :: dt = .2
call plasma_push1 (electrons, efield, dt)
call plasma_dpost1 (electrons, charge_density)
```

In this Fortran90 example the collection of electrons, with their properties, are localized to the definition of the species type. Additional species can be created, including ions, due to the encapsulation of species features. Similarly, the properties of fields are encapsulated into a fields type from which the charge density and electric field are created. The details of how these fields are stored and manipulated are hidden in the definition of the type. This notion, related to encapsulation and known as information hiding supports creation of user defined types

called abstract data types. Together, encapsulation, information hiding, inheritance, and abstract data types form the building blocks of object-oriented programs.

Comparing the plasma_push1 and plasma_dpost1 calls to the push1 and dpost1 calls, the parameters have been simplified representing problem-related abstractions. These new data types and associated operations can be combined into a module for coherency. Modules can be used in parts of the program that require access to the routines and data they provide. For added protection, the internal details of the module can be hidden (private). Then, the only way to manipulate module data is with the functions and subroutines that the module makes publicly accessible. The private data will always be directly accessible to the routines defined within the module. This provides a clean interface, adding clarity, protection, and a stable context for information sharing and collaboration that are essential for increasingly complex scientific programs.

The module was a critically important new addition to Fortran90 since modules increase the visibility and accessibility of data and routines throughout the program. Modules are much more powerful than this, however. They can easily be used to support the object-oriented methodology. Modules allow encapsulation of derived types and the routines that operate with them; they can be "used" wherever module variables (objects) are needed. The use statement allows the features of modules to be accessible to any program unit. For example, we can create a module for the particle species that requires features of the distribution function, seen below:

```
MODULE species_module
USE distribution_module        ! access additional features
IMPLICIT NONE
TYPE particle1d
   REAL :: x, vx               ! x position and velocity
END TYPE particle1d
TYPE species
   REAL :: qm, qbm, ek         ! charge, charge/mass, energy
   INTEGER :: nop              ! number of particles
   TYPE (particle1d), DIMENSION (:), POINTER :: p
END TYPE species
CONTAINS
   SUBROUTINE species_distribute (species,distf)
   ! details omitted...
   END SUBROUTINE species_distribute
! list of additional routines...
END MODULE species_module
```

Sketch of a Fortran90 collective species module that uses a distribution function module for spatial and velocity distribution of various particle species is given above.

Through use-association, access to the distribution_module derived types and features is now provided to the species_module. A function to distribute the species, given the distribution properties from the distf object, can be defined in the species_module after the contains statement:

```
SUBROUTINE species_distribute (species,distf)
TYPE (species1d), INTENT (out) :: species
TYPE (distribution_function1d), INTENT (in) :: distf
   ! uniform density profile
   do j = 1, distf%number_of_particles_x
      species%p(j)%x = distf%spatial_density*(float(j)-.5)
   enddo
   ! remainder of routine...
END SUBROUTINE species_distribute
```

The distf object has direct access to its components through use-association of the distribution module. Note that Fortran90 loops do not require labels and that free-format source code programming is supported.

Returning to the species_module, notice that in addition to the distribution features, a particle1d type and a species type have been added. The species includes the charge (qm), charge/mass (qbm), species kinetic energy (ek), and the number of particles in the species (nop) as part of the encapsulated definition. A pointer to a one-dimensional array of particles is also included in the species derived type. Fortran90 supports pointer types and dynamic memory allocation; however, we should explain why this pointer structure was introduced.

Derived types have some restrictions on their content. We require storage for an array of particles as part of the species derived type definition. If the number of particles is known at compile time, the derived type could contain the statement:

```
TYPE (particle1d), DIMENSION (NumberOfParticles) :: p
```

However, the number of particles may be determined dynamically implying that storage must be allocated at runtime. An allocatable array in Fortran90 allows the programmer to create and destroy arrays dynamically, but allocatable arrays are not allowed in derived type definitions in Fortran90. Pointers to arrays are allowed, explaining the use of the pointer attribute:

```
TYPE (particle1d), DIMENSION (:), POINTER :: p
```

where a pointer to a one-dimensional dynamically allocated array is declared. Storage allocation occurs in a separate executable statement. Some recent Fortran compilers allow the allocatable attribute to appear in a derived type, but this is not universally available in current compiler implementations.

Use-association gives the species_distribute routine direct access to the distribution function derived type, since the components of the distribution_module were public. However, we may want to restrict access for protection or information hiding. Any component of a module can be made private to the module, limiting external access to its internal management while creating a standard interface to the features the module provides. Indeed, the designer has complete control over the accessibility of any component of the module. This includes data, derived types, and routines that may be specified as public or private explicitly. When a derived type is in module, its components may be private while the type itself is still publicly available wherever the module is used. If the following derived type were defined in the species_module

```
TYPE particle1d
   PRIVATE
   REAL :: x, vx       ! x position and velocity
END TYPE particle1d
```

where the position and velocity are declared, then objects of type particle1d can be created by use-association of the module, but the components are only accessible to routines defined within the module or by an interface that the module must provide. Alternatively the derived type may be entirely private to the species_module:

```
TYPE, PRIVATE :: hidden_particle1d
         REAL :: x, vx
      END TYPE hidden_particle1d
```

Now, variables of type hidden_particle1d cannot be created outside of the module by use-association; the type is hidden. These examples illustrate two forms of information hiding.

All modules have a default accessibility of public types unless explicitly specified otherwise. The accessibility to each component of a module can be listed explicitly, or overridden, if desired. To encourage information hiding in object-oriented programming the default accessibility of the module can be private, followed by a list of routines and data that are publicly accessible by the public attribute. Modules should be used as much as possible in object-oriented Fortran90 programming since they provide the only mechanism to gain access to the advanced features of the language. They will have an important role in the construction of inheritance relationships in Fortran90. Topics regarding inheritance and polymorphism representation in Fortran90 are discussed in greater depth in [4,5].

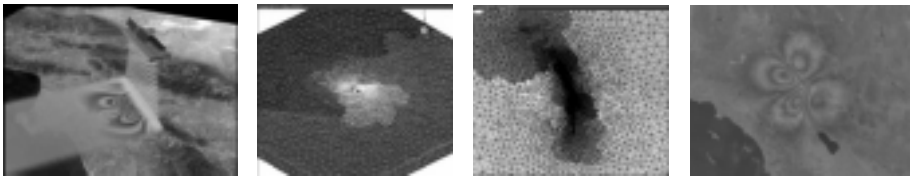


Fig. 2. Quakesim Project simulation of Lander's Earthquake showing finite element mesh domain and InSAR fringe visualization of crustal deformation developed under NASA's ESTO/CT program at JPL. GeoFEST and PYRAMID codes have been coupled and run on Project Columbia and other systems. The InSAR fringe image was visualized using RIVA developed by J. Parker, G. Lyzenga, C. Norton, B. Tisdale, T. Baker, M. Glasscoe, Peggy Li, and A. Donnellan (PI).

3.1. New challenges and the impact of new Fortran90 features

Our work on the PIC algorithm was successful and drove additional application of these techniques to other areas of scientific programming important to our work, and the work of others. Indeed, we participated in the WG5/J3 standards committee meeting to explain and demonstrate the role of object-oriented methods based on Fortran90 that helped to usher formal standardization of object-oriented features as technical reports (TRs) into Fortran2000 (which later became Fortran2003).

Nevertheless, we found other application areas that would benefit from applying these techniques while still having Fortran90 and Fortran95 available, such as adaptive mesh refinement. Most of the codes and libraries for this application were based on C and C++, but many of the legacy solvers were in Fortran77. The new features of Fortran90/95 allowed for infrastructure development in Fortran90 that could be efficient and scalable while also supporting interoperability with C (which is another feature that appears in later Fortran standards).

The next section describes how we transition from simply reorganizing a legacy code to improve development and what the role modern concepts in Fortran played in solving problems of greater complexity where mesh adaptation and integration with legacy solvers (in C) were required.

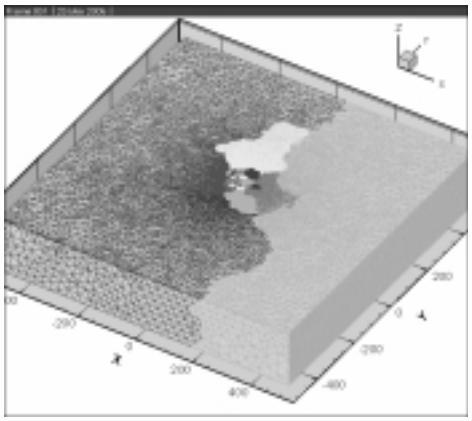
4. Modern Fortran for adaptive mesh refinement in geophysics

Under the NASA High Performance Computing and Communications (HPCC) Earth and Space Sciences (ESS) Project the vision of creating a new software tool supporting unstructured parallel adaptive mesh refinement (AMR) was created. The goal was to develop a library maximizing the use of new and most appropriate features of Fortran90 to provide a modern, simple, efficient, and scalable approach for large-scale parallel finite element analysis.

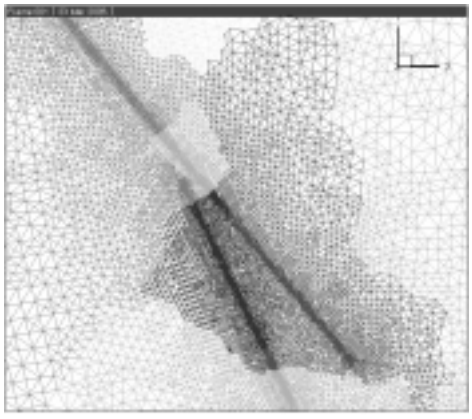
Under the internal development project named "PYRAMID", research and development began in 1998 to realize this vision. The work has since transitioned to the NASA Earth-Sun Systems Technology Office Computational Technologies (ESTO/CT) Program. PYRAMID established the feasibility of the goal and demonstrated that key technological issues could be overcome. It validated the following core principles and ideas:

1. Object-Oriented design methodology for finite element methods (FEM) based on modern software features of Fortran90
2. Automatic mesh quality control to ensure good element aspect-ratios (geometry) under successive adaptive refinement
3. Development of high-level library routines to simplify the use of parallel AMR techniques in application programs

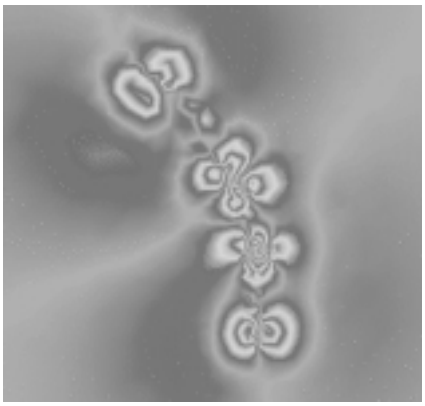
The PYRAMID advanced prototype software has been released to the public via the Open Channel Foundation (http://www.openchannelfoundation.org) and is being used for Earth, space, and engineering applications both within and outside NASA. The most recent use of the software has been for Solid Earth earthquake crustal deformation and active tectonics modeling. The coupling of PYRMAID with the GeoFEST geophysical finite element simulation tool now allows efficient simulations involving millions of elements where only thousands of elements were once possible [6].



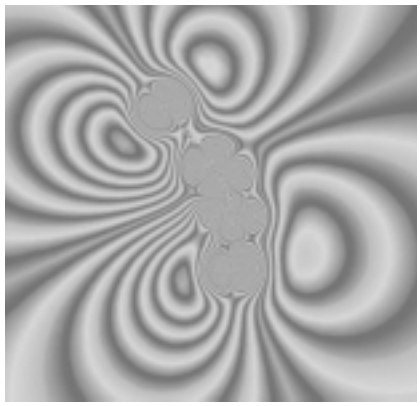
Input Mesh: Finite element mesh over 1000 x 1000 x 60 km domain. Colors indicate partitioning over 490 processors (available) but only 16 colors are used.



AMR Mesh: Zoom view of 2 neighbor faults for Landers simulation where solution-adaptive refinement was applied based on strain energy in the fault system.



Low Resolution: InSAR fringe map of co-seismic Landers earthquake with low-resolution mesh does not resolve surface displacement well.



High Resolution: Use of PYRAMID for AMR dramatically improves InSAR fringe map over low-resolution case.

Fig. 3. Simulation results of 1992 Landers earthquake event showing dramatic improvement in resolution due to unstructured parallel adaptive mesh refinement with PYRAMID.

PYRAMID contains numerous routines that support mesh I/O, repartitioning, load balancing, mesh migration, adaptive refinement, numerical methods, and visualization all capable of running in parallel using the Message Passing Interface (MPI) standard for inter-processor communication. The current version is designed in Fortran 95 with many capabilities that simplify integration with existing Fortran (and C-based) numerical solvers for Earth and Space science applications.

This software has been used on a variety of large computational systems such as Project Columbia at NASA Ames Research Center, Cosmos at the Jet Propulsion Laboratory, System X at Virginia Tech., as well as numerous other parallel computers and clusters. These represent the Intel, SGI, and Apple platforms, all running variants of the Unix Operating System thereby demonstrating the portability of the software.

The images in Fig. 3 show how PYRAMID has been used in the Quakesim project to model crustal deformation. This example shows the outcome of parallelizing the GeoFEST geophysical finite element simulation code with PYRAMID. Visualization of the 1992 Landers Earthquake has been simulated over a 500 year period where surface deformation is represented as InSAR fringes, overlaid on Landsat data of the Earth's surface, corresponding to vertical ground surface uplift. The simulation is fully 3D, but only selected snapshots of the 2D surface are shown. GeoFEST can now perform larger and more detailed simulation than previously possible.

The Quakesim project was motivated by our recognition that tool support for modeling is critical and represents the only mechanism by which phenomena, described by complex interactions that occur both over extremely short and long scales in space and time, can be understood. An essential goal of?? this effort is to combine space-based observational data with simulation models to come to a more complete understanding of global Solid Earth processes. This capability will be an important part of an InSAR analysis system, as such a mission is currently being investigated within the scientific community.

4.1. Modifying the GeoFEST legacy software

The finite element technique offers currently the most general framework for modeling heterogeneous faulted regions of the Earth's crust (e.g., Los Angeles). GeoFEST(P) is an MPI-parallel code which has demonstrated 500 year simulations of postseismic Southern California deformation processes including multiple interacting faults, using 1.4 million finite elements, half-year time steps, and up to 512 processors of various computing systems. Wallclock times are typically a few hours per run.

GeoFEST simulates stress evolution, fault slip and plastic/elastic processes in realistic materials [6–8]. The products of such simulations are synthetic observable time dependent surface deformation on scales from days to decades. Scientific applications of the code include the modeling of static and transient co– and postseismic Earth deformation, Earth response to glacial, atmospheric and hydrological loading, and other scenarios involving the bulk deformation of geologic media. It has also been integrated into the QuakeSim portal, which offers users an integrated web services environment. Problems can be specified and solved by non-experts through the web portal, and the resulting deformation can be displayed in combination with Landsat imagery and a digital elevation model.

GeoFEST is designed to aid in interpretation of GPS, InSAR and other geodetic techniques, some of which are undergoing an exponential increase in volume due to NASA remote sensing and satellite geodesy programs. For these, and other reasons, parallelizing GeoFEST represents an important part of the Quakesim Project. For example, simulations are planned that will use a 16 million element model of the Los Angeles basin to find a physical basis for the observed localized compression of the northern portion of the basin. Adaptive mesh refinement capabilities are also needed, allowing elements to be concentrated in areas where high resolution is required. To achieve these goals GeoFEST has been coupled with the PYRAMID library.

Adaptive Mesh Refinement (AMR) is an advanced computational technology applicable to a wide variety of science and engineering simulation problems [7,8]. The PYRAMID library supports parallel adaptive methods by providing parallel grid-based re-meshing techniques. Our software simplifies the user interactions with the grid for problems that exhibit complex geometry and require the use of parallel computers. PYRAMID contains many library commands, written in Fortran90, generally organized around the concepts of a mesh data structure and operations that allow manipulation of that structure. All of the concepts about parallel computation are hidden behind library interface commands.

4.2. Software development

Working closely with the Quakesim Team, we instrumented the GeoFEST software with PYRAMID library commands to support parallelization of the sequential version of the code. This required building a C-adaptor library that allowed GeoFEST (written in C) to use the PYRAMID library (written in Fortran90). Furthermore, commands from the library for I/O, parallel mesh partitioning, and support for communication of boundary data among processors were added and/or created. During this process, optimizations were added to both codes. The resulting software runs on a variety of platforms including both Intel Pentium/Itanium-based and Apple PowerPC-based cluster computers with high efficiency.

The team used an approach rarely applied in scientific software development called "Extreme Programming". In this method team members collaborate interactively during the design and integration process. This software team development technique consists of one team member editing the software while all members contribute ideas and approaches by viewing the software within a group setting. This method was effective in allowing all team members to bring their specific expertise to the effort that included geophysics, computational science, computer science, and visualization. It also helped to minimize errors in the software development as a team member often

detected problems immediately. Using this method was not an a-priori objective – it simply occurred due to the nature of our collaboration. Four members were involved and this appeared to be an appropriate number in this case. Nevertheless, our team faced numerous challenges during the development process. These challenges, and their solutions, are presented below.

Using techniques developed in the previous projects, like the plasma simulation work, we were able to create an advanced data structure for parallel systems. The AMR library acts on mesh "objects" where the external interface to various libraries, as well as within the code itself, has an object-oriented look and feel. The following example, for instance, allows one to define a hierarchy of meshes, load input data, perform data distribution, define term properties on mesh components and perform the adaptive process on parallel computers. This is a skeleton example as there are well over 100 routines that implement various operations.

```
PROGRAM pyramid_example
USE pyramid_module
   type (mesh), dimension(10) :: meshes
   real, dimension(:), allocatable :: node_terms, element_terms
   integer, dimesion(:), allocatable :: refine_elems
   call PAMR_INIT( meshes )
   call PAMR_DEFINE_MESH_TERMS( meshes, num_node_terms = 10,
        & num_element_terms = 5 )
   call PAMR_LOAD_MESH( meshes(1), "Meshes/input_mesh.dat" )
   call PAMR_REPARTITION( meshes(1) )
   call PAMR_SET_MESH_NODE_TERMS( meshes(1), node_terms )
   do I = 1, refine_level
        call PAMR_MARK_REFINEMENT( meshes(1), meshes(2), refine_elems)
        call PAMR_LOGICAL_AMR( meshes(1) )
        call PAMR_REPARTITION( meshes(1) )
        call PAMR_PHYSICAL_AMR( meshes(1), meshes(2) )
        call PAMR_ELEMENT_COUNT( meshes(2) )
   end do
   call PAMR_FINALIZE( .mpi_active = .true. )
END PROGRAM pyramid_example
```

4.3. C and Fortran90 interoperability

GeoFEST is written in the C programming language while PYRAMID is implemented in Fortran90, so interoperability was an immediate challenge. The approach we used was not based on creating C wrapper software around existing Fortran90 calls. While this approach is completely portable, it would require the use of numerous copies as pointer data structures are passed from Fortran90 to C.

Instead, a C-adapter library was created that represents an innovation for inter-language communication between Fortran90 and C. This is an interface that provides direct access to Fortran90 data structures from C. This is accomplished by creating an exact replica (in C) of the PYRAMID Fortran90 mesh data structure and calculating a few compiler specific entities, such as the size, representation, and layout of Fortran90 pointer types. Although arguments are passed from PYRAMID in Fortran90 to GeoFEST in C, no copies are ever made and there is effectively no cost imposed on the interaction. Note, however, that Fortran 2003 includes C interoperability as part of the standard.

4.4. Parallel programming issues

The parallel version of GeoFEST is designed to be as functionally similar to the original sequential code as possible. From the user perspective, the code is essentially identical, with a few additional steps that convert the sequential input file into one that the parallel code can utilize. The basis for the parallel computation performed by GeoFEST is the concept of domain decomposition. The machine model assumed for this style of parallel

computing consists of some number of independent processors, each with its own addressable core memory space. The processors are each executing identical code, but not synchronously, as each processor acts and branches in distinct ways on its unique data. The processors interact and exchange data with one another by message passing, and this communication is mediated in the GeoFEST code through use of routines from the PYRAMID library, as well as through a small number of direct calls to the MPI protocol.

At the algorithmic level, domain decomposition requires each of the processors to work on a given spatially contiguous piece of the finite element grid. This requirement necessitates communication needed to update and maintain consistency between the subdomains where they join one another. Implementing such communication efficiently is the principal challenge of the parallel programming problem. While the PYRAMID library provides data to GeoFEST describing the current data distribution, routines were needed to handle the inter-partition data communication. An example of the domain partitioning of mesh for 1992 Landers earthquake event is shown in Fig. 3.

In the GeoFEST parallel decomposition scheme, each processor has exclusive ownership of a block of finite elements; from this it follows that there will exist components, such as nodes, that are shared among processors. These are the nodes that are simultaneously members of elements that belong to two or more different processors. From this scheme, it follows that certain tasks (those which are inherently element-based) can be carried out completely in parallel, without need for interprocessor communication. On the other hand, tasks that are inherently node-based will generally require addition of updating steps that communicate shared nodal information between processors.

The calculation and storage of element stiffness matrix contributions is a task of the first kind; once GeoFEST has been given processor assignments for each element from PYRAMID's data partitioning, the formation of each element contribution can proceed independently in each processor. However, operations involving the assembled vector of nodal displacements that comprise the fundamental unknowns of the problem are of the second kind. This fact led us to a decision point in choosing the solver for the global finite element matrix equation that would be an interactive preconditioned conjugate gradient (PCG) solver. This approach has benefits, given our domain decomposition strategy and the need to demonstrate scalability for large three-dimensional problems on parallel computers. (This also gives a hint of the kinds of capabilities modern codes now need for choosing various capabilities that will be addressed by techniques like the strategy pattern.)

The PCG algorithm does not require the stiffness matrix to be assembled in global form; it is sufficient to retain the individual element contributions (and accompanying indexing information) in element-specific storage distributed among processors. As for node-based vectors such as the vectors of displacements and forces, each processor stores that subset of the vectors that correspond to the nodes exclusively within its region, along with redundant storage of all the nodal degrees of freedom that are shared, that is, that are located on a boundary between processor regions.

Note that the two important tasks in the algorithm that require interprocessor communication are the vector dot product and the stiffness matrix-vector product. In the case of the former, each processor calculates its contribution to the scalar product, using the vector entries in its local storage. This is immediately followed by MPI communication calls that combine the pieces into a global result and distribute the product to all processors. At the conclusion of this operation, each processor is then free to carry on with its independent process. The matrix-vector product is carried out similarly, although the communication pattern is somewhat more complex. In this task, each processor carries out the multiplication of locally stored matrix elements with locally stored vector entries. The result is usually a vector entry in the local processor, but some of the results will fall on a boundary node that is shared with another processor. In this case, rather than a global (all processors) MPI communication, a pair-wise communication between the involved processors is used to update and reconcile the vector results at all shared nodes. These communications are synchronized, so that at the conclusion of the communication step, all processors will contain vector values that agree with one another, and with the values that would be obtained in the equivalent single-processor sequential calculation. The PYRAMID library provides routines for GeoFEST to accomplish these operations.

5. The strategy pattern and Fortran

It is often the case that we use different algorithms in scientific computing in different situations, or we want to compare the behavior of different algorithms to help us decide which one is the best. The strategy pattern is

designed to encapsulate the choice of algorithm and make the algorithms interchangeable. It does so by separating the selection of the algorithm from the implementation.

In many scientific programs, it is often the case that one needs to solve a set of differential equations subject to different boundary conditions. For example, in particle simulation of plasmas, one needs to solve for the electrostatic potential Φ, given a charge density ρ provided from the particle data. A widely used boundary condition, especially for basic studies, is periodic. This boundary condition can be implemented by making use of a Fourier series expansion. The charge density, ρ an be represented as follows:

$$\rho(x,y) = \sum_{n,m} \rho_{nm} e^{2\pi i n x/Lx} e^{2\pi i m x/Ly}$$

for the system of size Lx, Ly. The potential, Φ, is then given by:

$$\Phi(x,y) = \sum_{n,m} \Phi_{nm} e^{2\pi i n x/Lx} e^{2\pi i m x/Ly}, \text{ where}$$

$$\Phi_{nm} = \frac{4\pi\rho_{nm}}{(2\pi n/Lx)^2 + (2\pi m/Ly)^2}.$$

We will begin by creating a class for fields. The purpose of a class is to simplify handling variations. Classes are not a native construct in Fortran95, but they can easily be implemented by the use of modules which have the following structure: a type, followed by functions which operate on that type, and optional shared data [4]. Let us start with a base class which contains the common elements needed by all fields.

```
   module base_field_class

   type field
      integer :: psolve
      integer :: nx, ny, nz
   end type

contains

!
   subroutine new_field(this,nx,ny,nz,psolve)
! assign initial values to type
   ...
   subroutine init_field(this,f)
! allocate and initialize field
   ...
   subroutine delete_field(this,f)
! deallocate field
   ...
   end module base_field_class
```

The type describes properties of fields, but does not actually contain field data, which will be stored in normal (and familiar) Fortran95 arrays. In this example, the integers (`nx, ny, nz`) describe the size of the field, and `psolve` will describe the type of solver to be used. The constructor of the class merely assigns initial values to the type, as follows:

```
   subroutine new_field(this,nx,ny,nz,psolve)
! assign initial values to type
   implicit none
   type (field) :: this
   integer :: nx, ny, nz, psolve
```

```
   this%nx = nx; this%ny = ny; this%nz = nz
   this%psolve = psolve
   end subroutine new_field
```

The function `init_field` allocates the actual field array

```
   subroutine init_field(this,f)
! allocate and initialize field
   implicit none
   type (field) :: this
   real, dimension(:,:,:), pointer :: f
   allocate(f(this%nx,this%ny,this%nz))
   end subroutine init_field
```

and the function `delete_field` deallocates it. The class is listed in full in Appendix A.

To implement the periodic Poisson solver, we create a `periodic_field` class, which makes use of, or "inherits" the `base_field` class.

```
   module periodic_field_class
   use base_field_class
   integer, parameter :: PERIODIC = 1

   contains
!
   subroutine solve_poisson(this,rho,phi)
! solve for poisson equation with periodic boundary conditions
   ...
   end module periodic_field_class
```

This class contains one public function, `solve_poisson`, which provides the solution in real space. It also makes the functions of the `base_field` class available.

```
   subroutine solve_poisson(this,rho,phi)
! solve for poisson equation with periodic boundary conditions
   implicit none
   type (field) :: this
   real, dimension(:,:,:), pointer :: rho, phi
   call fft(this,rho,-1)
   call periodic_poisson(this,rho,phi)
   call fft(this,phi,1)
   end subroutine solve_poisson
```

The full listing of this class is in Appendix B. To make use of this solver, the main program may look like:

```
program main
use periodic_field_class
integer :: nx = 32, ny = 32, nz = 32, psolve = 0
real, dimension(:,:,:), pointer :: rho, phi
type (field) :: es
call new_field(es,nx,ny,nz,psolve)
call init_field(es,rho)
call init_field(es,phi)
call solve_poisson(es,rho,phi)
end program main
```

Although the `new_field` constructor asks for the variable `psolve`, this class does not make use of it.

Another boundary conditions which is commonly used is the Dirichlet boundary condition, which physically represents a conductor. This boundary condition can be implemented by making use of a sine expansion. The charge density in this case can be represented as a Fourier sine series, as follows:

$$\rho(x,y) = \sum_{n,m} \rho_{nm} \sin(n\pi x/Lx) \sin(m\pi y/Ly).$$

The potential is then given by:

$$\Phi(x,y) = \sum_{n,m} \Phi_{nm} \sin(n\pi x/Lx) \sin(m\pi y/Ly), \text{ where}$$

$$\Phi_{nm} = \frac{4\pi\rho_{nm}}{(\pi n/Lx)^2 + (\pi m/Ly)^2}.$$

To implement the Dirichlet Poisson solver, we create a `dirichlet_field` class, which also makes use of the `base_field` class.

```
   module dirichlet_field_class
   use base_field_class
   integer, parameter :: DIRICHLET = 2

   contains
!
   subroutine solve_poisson(this,rho,phi)
! solve for poisson equation with dirichlet boundary conditions
   ...
   end module dirichlet_field_class
```

This class contains one public function, `solve_poisson`, which provides the solution `phi` from a given density function `rho`.

```
   subroutine solve_poisson(this,rho,phi)
! solve for poisson equation with dirichlet boundary conditions
   implicit none
   type (field) :: this
   real, dimension(:,:,:), pointer :: rho, phi
   call fst(this,rho)
   call dirichlet_poisson(this,rho,phi)
   call fst(this,phi)
   end subroutine solve_poisson
```

The full listing of this class is in Appendix C. To make use of this solver, the main program looks exactly like the previous one, except that the statement

```
use periodic_field_class
```

is replaced by

```
use dirichlet_field_class
```

Such a simple change was possible because both solvers had the same signature (or interface), that is, they had the same subroutine name and arguments.

To implement the strategy pattern, we create a `fields_strategy` class which will choose the solver for us. To do this, we make use of the variable, `psolve`, in the `fields` type, which has been ignored until now. This class needs to contain a function which will choose the appropriate solver to call. However, we have a name conflict if the both classes use the same name for the solver. Fortunately, Fortran95 has a renaming facility that we can use to locally rename the functions when one module uses another. In the example here, we rename the `solve_poisson` function in the `periodic_field` class to `p_solve_poisson`, and the `solve_poisson` function in the `d_fields` class is renamed `d_solve_poisson`:

```
module fields_strategy_class
use periodic_field_class, p_solve_poisson => solve_poisson
use dirichlet_field_class, d_solve_poisson => solve_poisson
implicit none
```

Then we can refer to the different names when we create a new `solve_poisson` function which implements the strategy pattern:

```
   subroutine solve_poisson(this,rho,phi)
! solve poisson equation
   implicit none
   type (field) :: this
   real, dimension(:,:,:), pointer :: rho, phi
   select case (this%psolve)
   case (PERIODIC)
      call p_solve_poisson(this,rho,phi)
   case (DIRICHLET)
      call d_solve_poisson(this,rho,phi)
   end select
   end subroutine solve_poisson
```

To make use of the strategy pattern, the only change to the main code is needed to replace the statement

```
use periodic_field_class
```

with

```
use fields_strategy_class
```

We also need to set the psolve variable to the desired value, either the constant `PERIODIC`, which is defined in the `periodic_field` class, or the constant `DIRICHLET`, which is defined in the `dirichlet_field` class. The complete listing of the Fields Strategy class is given in Appendix D. Note that we have encapsulated what varies (the choice of solver), and separated the decision making from the implementations in the different classes.

6. Conclusion

In this paper, we have reviewed our early and more recent work in applying modern Fortran to complex scientific applications. We have also tried to show the significance of strategy of software design pattern for scientific programming and the ways to implement it in Fortran95. Although Fortran95 is not an object-oriented language, many design patterns can be implemented quite naturally. One reason for this is that most design patterns use object composition but not inheritance. This is fortunate since Fortran95 does not natively support inheritance. The major difference between the implementation of such patterns in Fortran95 and in an object-oriented language is the explicit appearance of `select case` constructs to implement polymorphism. However, by separating the polymorphism from the implementation, this requirement becomes easily manageable. In fact, for many scientists, it is preferable to have such choices explicit and obvious than to have them implicit and hidden. Notice that we have implemented all the examples as skeleton code, which could actually delegate work to some legacy code. This separation of object design from low level implementation allows one to rapidly design and redesign such patterns until the desired program is achieved.

Acknowledgments

The work of C.D.N was carried out at the Jet Propulsion Laboratory, California Institute of Technology, under a contract with the National Aeronautics and Space Administration. The work of V.K.D. was carried out in part at the Jet Propulsion Laboratory, California Institute of Technology, under a contract with the National Aeronautics and Space Administration. It was also supported in part by the US Department of Energy, under the SCIDAC program.

Appendix A: Base Field Class

```
   module base_field_class
!
   implicit none
   private
   public :: field, new_field, init_field, delete_field
!
   type field
      integer :: psolve
      integer :: nx, ny, nz
   end type
!
   contains
!
   subroutine new_field(this,nx,ny,nz,psolve)
! assign initial values to type
   implicit none
   type (field) :: this
   integer :: nx, ny, nz, psolve
   this%nx = nx; this%ny = ny; this%nz = nz
   this%psolve = psolve
   write (*,*) 'calling new_field'
   end subroutine new_field
!
   subroutine init_field(this,f)
! allocate and initialize field
   implicit none
   type (field) :: this
   real, dimension(:,:,:), pointer :: f
   allocate(f(this%nx,this%ny,this%nz))
   write (*,*) 'calling init_field'
   end subroutine init_field
!
   subroutine delete_field(this,f)
! deallocate field
   implicit none
   type (field) :: this
   real, dimension(:,:,:), pointer :: f
   deallocate(f)
   write (*,*) 'calling delete_field'
   end subroutine delete_field
!
   end module base_field_class
```

Appendix B: Periodic Field Class

```
   module periodic_field_class
!
   use base_field_class
```

```
   implicit none
   private
   public :: PERIODIC, field, new_field, init_field, delete_field
   public :: fft, periodic_poisson, solve_poisson
!
   integer, parameter :: PERIODIC = 1
!
   contains
!
   subroutine fft(this,f,isign)
! perform fft on field f
   implicit none
   type (field) :: this
   real, dimension(:,:,:), pointer :: f
   integer :: isign
   write (*,*) 'calling fft'
   end subroutine fft
!
   subroutine periodic_poisson(this,rho,phi)
! solve for poisson equation in fourier space
   implicit none
   type (field) :: this
   real, dimension(:,:,:), pointer :: rho, phi
   write (*,*) 'calling periodic_poisson'
   end subroutine periodic_poisson
!
   subroutine solve_poisson(this,rho,phi)
! solve for poisson equation with periodic boundary conditions
   implicit none
   type (field) :: this
   real, dimension(:,:,:), pointer :: rho, phi
   call fft(this,rho,-1)
   call periodic_poisson(this,rho,phi)
   call fft(this,phi,1)
   end subroutine solve_poisson
!
   end module periodic_field_class
```

Appendix C: Dirichlet Field Class

```
   module dirichlet_field_class
!
   use base_field_class
   implicit none
   private
   public :: DIRICHLET, field, new_field, init_field, delete_field
   public :: fst, dirichlet_poisson, solve_poisson
!
   integer, parameter :: DIRICHLET = 2
!
```

```
   contains
!
   subroutine fst(this,f)
! perform fast sine transform on field
   implicit none
   type (field) :: this
   real, dimension(:,:,:), pointer :: f
   write (*,*) 'calling fst'
   end subroutine fst
!
   subroutine dirichlet_poisson(this,rho,phi)
! solve for poisson equation in fourier space
   implicit none
   type (field) :: this
   real, dimension(:,:,:), pointer :: rho, phi
   write (*,*) 'calling dirichlet_poisson'
   end subroutine dirichlet_poisson
!
   subroutine solve_poisson(this,rho,phi)
! solve for poisson equation with dirichlet boundary conditions
   implicit none
   type (field) :: this
   real, dimension(:,:,:), pointer :: rho, phi
   call fst(this,rho)
   call dirichlet_poisson(this,rho,phi)
   call fst(this,phi)
   end subroutine solve_poisson
!
   end module dirichlet_field_class
```

Appendix D: Field Strategy Class

```
   module fields_strategy_class
! fields with various boundaries
!
   use periodic_field_class, p_solve_poisson => solve_
      poisson
   use dirichlet_field_class, d_solve_poisson => solve_
      poisson
   implicit none
   private
   public :: PERIODIC, DIRICHLET
   public :: field, new_field, init_field, delete_field
   public :: solve_poisson
!
   contains
!
   subroutine solve_poisson(this,rho,phi)
! solve poisson equation
   implicit none
```

```
   type (field) :: this
   real, dimension(:,:,:), pointer :: rho, phi
   select case (this%psolve)
   case (PERIODIC)
      call p_solve_poisson(this,rho,phi)
   case (DIRICHLET)
      call d_solve_poisson(this,rho,phi)
   case default
      write (*,*) 'invalid psolve = ', this%psolve
   end select
   end subroutine solve_poisson
!
   end module fields_strategy_class
```

References

[1] E. Gamma, R. Helm, R. Johnson and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software,* Addison-Wesley, Reading, MA, 1995.

[2] A. Shalloway and J.R. Trott, *Design Patterns Explained,* Addison-Wesley, Boston, MA, 2002.

[3] S.J. Metsker, *Design Patterns Java Workbook,* Addison-Wesley, Boston, MA, 2002.

[4] V.K. Decyk, C.D. Norton and B.K. Szymanski, How to Express C++ Concepts in Fortran90, *Scientific Programming* **6**(4) (1997), 363.

[5] V.K. Decyk and C.D. Norton, A simplified method for implementing run-time polymorphism in Fortran95, *Scientific Programming* **12** (2004), 45.

[6] J.W. Parker, A. Donnellan, G. Lyzenga, J.B. Rundle and T. Tullis, Performance Modeling Codes for the QuakeSim Problem Solving Environment, Computational Science, in *Proc. International Conference on Computational Science* (*Part III*), Springer-Verlag, Berlin, 2003, 845–862.

[7] A. Donnellan, G. Lyzenga, J. Parker, C. Norton, M. Glasscoe and T. Baker, *GeoFEST User's Guide,* 2003, http://www.openchannelsoftware.org/.

[8] A. Donnellan et al., *Numerical Simulations for Active Tectonics Processes: Increasing Interoperability and Performance,* Tech. Rep., JPL, http://quakesim.jpl.nasa.gov/, Aug 2003.

[9] C.D. Norton and T.A. Cwik, Early Experiences with the Myricom 2000 Switch on an SMP Beowulf Class Cluster for Unstructured Adaptive Meshing, in: *2001 IEEE International Conference Conference on Cluster Computing,* D. Katz, T. Sterling, et al., eds, IEEE Task Force on Cluster Computing, IEEE Computer Society, Newport Beach, CA, October 8–11, 2001.

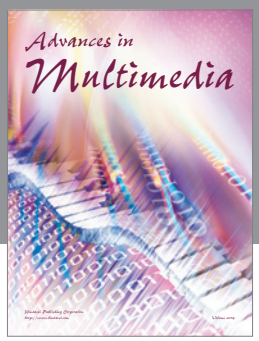
[10] C.D. Norton and T.A. Cwik, *Parallel Unstructured AMR and Gigabit Networking for Beowulf-Class Clusters,* Lecture Notes in Computer Science, vol. 2328, 2002.

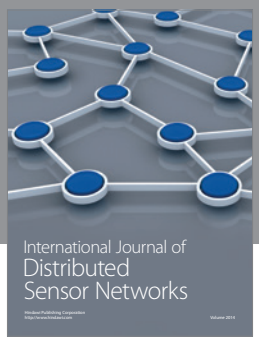
[11] T.M.R. Ellis, I.R. Philips and T.M. Lahey, *Fortran90 Programming,* Addison-Wesley, Reading MA, 1994.

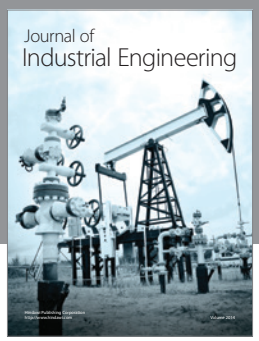
[12] C.D. Norton, B.K. Szymanski and V.K. Decyk, Object-Oriented Parallel Computation for Plasma Simulation, *Communications of the ACM* **38**(10) (October 1995), 88–100.

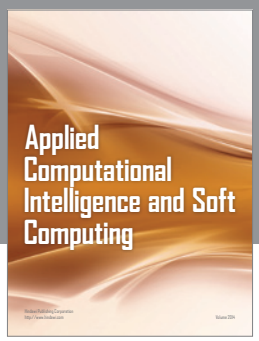
[13] C.K. Birdsall and A.B. Langdon, *Plasma Physics via Computer Simulation. The Adam Hilger Series on Plasma Physics,* Adam Hilger, New York, 1991.

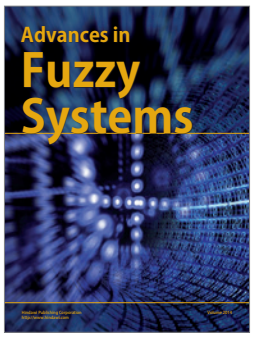
[14] P.C. Liewer and V.K. Decyk, A General Concurrent Algorithms for Plasma Particle-in-Cell Simulation Codes, *J of Computational Physics* **85** (1989), 302–322.

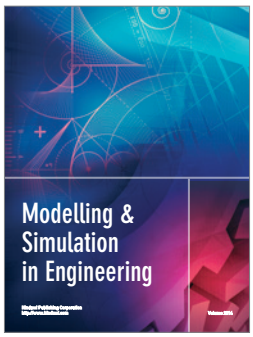


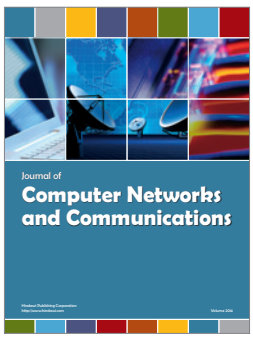












Hindawi
Submit your manuscripts at
http://www.hindawi.com