View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by CiteSeerX

Scientific Programming 12 (2004) 185–196
IOS Press

Design metrics in quantum turbulence simulations: How physics influences software architecture

Damian W.I. Rouson[a,*] and Yi Xiong[b,**]
[a]*Mechanical Engineering Department, The City College of The City University of New York, Convent Ave. at 140th St., New York, NY 10031, USA*
Tel.: +1 212 650 5210; Fax: +1 815 572 8203; E-mail: rouson@ccny.cuny.edu
[b]*Mechanical Engineering Department, The Graduate Center of The City University of New York, Fifth Ave. at 34th St., New York, NY 10015, USA*
E-mail: yxiongcuny@hotmail.com

Abstract. The information hiding philosophy of object-oriented programming encourages localizing data structures within objects rather than sharing data globally across different classes of objects. This emphasis on local data leads naturally to fine-grained data abstractions, particularly in scientific simulations involving large collections of small, discrete physical or mathematical objects. This paper focuses on a subset of such simulations where dynamically reconfigurable links bind the objects together. It is demonstrated that fine-grained data structures reduce the complexity of local operations on the data at the potential expense of increased global operation complexity. Two metrics are used to describe data structures: *granularity* is the number of instantiations required to cover the data space, whereas extent is the continuously traversable length of the data along a given direction. These definitions are applied to two abstractions for simulating the turbulent motion of quantum vortices in superfluid liquid helium. Several local and global operations on a fine-grained linked list are compared with those on a coarse-grained array. It is demonstrated that fine-grained data structures recover the simplicity of more coarse-grained structures if maximal extent is maintained as the granularity increases.

1. Introduction

As scientific software projects grow in size, more scientists and engineers are investigating object-oriented programming (OOP) as a means of managing code complexity [1,5,6]. The OOP strategy of constructing inheritance hierarchies reduces the amount of redundant code that must be written. The OOP philosophy of encouraging polymorphism simplifies the protocol for expressing similar functionality across different classes of objects. And the OOP techniques of encapsulation and information hiding reduce interdependencies between code modules.

It can be argued that inheritance and polymorphism attack problems that scale linearly with the size of the code. Given a piece of code that is inherited across G generations of a parent/child class hierarchy, inheritance reduces the number of times this code must be replicated from G to 1. Given P common procedures to be performed on C classes of objects, polymorphism reduces the size of the protocol for performing those procedures from CP to P. Note that the variables G, C and P each appear with a unit exponent in the above expressions.

By contrast, encapsulation and information hiding render an otherwise quadratic problem scale invariant. Consider that interactions between lines of code are mediated through access to shared data spaces (see Fig. 1).

*Corresponding author. Current address: US Naval Research Laboratory, 4555 Overlook Ave. SW, Washington, DC 20375, USA.
**Current address: Mechanical Engineering Department, University of California at Los Angeles, Los Angeles, CA 90095, USA.

ISSN 1058-9244/04/$17.00 © 2004 – IOS Press and the authors. All rights reserved

Given software with N lines of code accessing globally shared data, the maximum number of lines affected by changing one line is $N - 1$. Hence, the interdependencies between lines scales as $fN(N-1) = O(N^2)$, where f is the fraction of lines changed. If instead the data is encapsulated and hidden in small modules or objects with $M << N$ lines, the maximum number of lines affected is $M - 1$ independent of N. If M is held constant as N grows, the complexity of the design is scale-invariant.

This elimination of quadratic complexity suggests that at the intersection of data encapsulation and information hiding lies rich territory for exploring ways to significantly reduce program complexity. Encapsulation determines how data is partitioned. Information hiding determines how it is accessed. We present below one metric describing the partitioning: data structure *granularity*. We also present one describing the accessibility: data structure extent. The central argument of this paper is that the software design process simplifies as granularity is increased while maintaining maximal extent.

Although this paper focuses on quantum turbulence, we emphasize here that the issues raised apply to a broad class of problems involving the simulation of large sets of objects with dynamically reconfigurable interconnections, e.g. tracking points on a manifold undergoing tearing and reconnection. Such tearing and reconnection occurs at reaction sites on long chain molecules and along separation lines on droplets pinching off from a continuous stream.

In Section 2 below, we define granularity and extent. In Section 3, we apply these metrics to the object oriented design (OOD) of several data structures for simulating the turbulent motion of quantum vortices in superfluid liquid helium. The remainder of Section 3 presents the problem physics, the mathematical model, and two data encapsulation alternatives. One encapsulation strategy uses coarse-grained arrays and the other fine-grained linked lists. The relative merit of each is discussed in light of the resulting program logic. Section 4 presents conclusions.

2. Methodology

2.1. *Defining the data granularity metric*

Granularity is often discussed in terms of procedural parallelism [11,12]. In this context, it refers to the size of the independent, parallel instruction streams into which a given algorithm is decomposed. These streams can be processes, threads, code blocks, or even individual instructions. One can therefore measure the degree of procedural granularity by the number of instructions, floating-point operations or clock cycles per instruction stream. The higher the tally, the more coarse-grained is the code. The lower the tally, the more fine-grained is the code.

The above definition is inherently dynamic: procedural granularity determines run-time performance. An analogous measure to be introduced here is data structure granularity. This measure is static and fixed by the OOD process.

Before a formal definition is given, it is useful to consider a general class of scientific problems in which it is necessary to simulate the behavior of a large collection of small, discrete objects being modeled over a period of time. These objects can be physical, such as solid particles in a granular flow, or mathematical, such as grid cells in a finite element code. A straightforward OOD would typically create a one-to-one mapping between software objects and the corresponding physical or mathematical objects. The number of attributes that must be stored for each software object depends on the process being modeled and the marching algorithm used to advance in time. In most applications, however, the number of objects, N_o, will far exceed the number of attributes. For example, Rouson and Eaton [23] modeled the trajectory of $N_o = 72^3$ solid particles immersed in a turbulent carrier gas by solving ordinary differential equations of the form

$$\frac{d\mathrm{r}_{\mathrm{particle}}}{dt} = \mathbf{v}_{\mathrm{particle}}$$

$$\frac{d\mathrm{v}_{\mathrm{particle}}}{dt} = \frac{1}{\tau}(\mathbf{v}_{\mathrm{gas}} - \mathbf{v}_{\mathrm{particle}})$$

where $\mathbf{r}_{\mathrm{particle}}$ and $\mathbf{v}_{\mathrm{particle}}$ are the particle position and velocity, respectively, $\mathbf{v}_{\mathrm{gas}}$ is the gas velocity and τ is a particle time constant. A Navier-Stokes solver provided the gas velocity separately.

Rouson and Eaton approximated the ODE with a third-order Runge-Kutta scheme that required reserving memory for approximately 10 particle attributes per particle throughout each time step. We consider here the simpler case in which an explicit Euler marching algorithm is used. Then a three-dimensional (3D) position vector and a 3D velocity vector must be stored for each particle throughout each time-step. Thus, each particle exists in a six-dimensional state space.

We can now define data structure granularity:

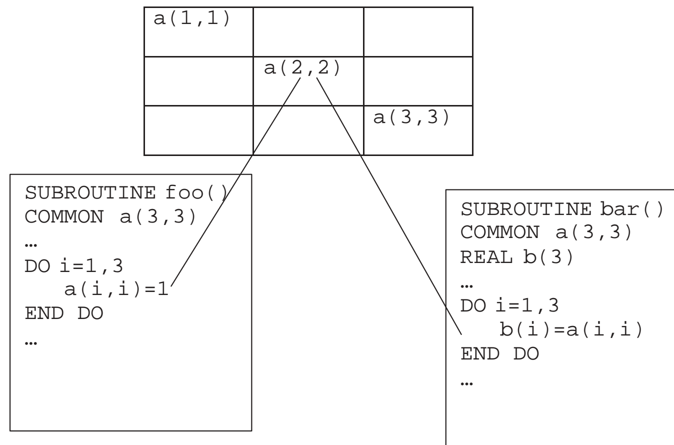


Fig. 1. Code line interactions mediated through shared access to data.

Definition 1: Given a class of objects, each instantiation of which has an N_s^-dimensional state space, we define *data structure granularity*, g, as the number, N_o, of objects required to cover the full $N_o \times N_s$-dimensional data space.

In the above particle simulations, g is high when there exists a one-to-one mapping between instantiations and particles, whereas g is minimal when all the particles are grouped into one object. In the latter case, we essentially recover the procedural programming limit but only within the particle class. Classes associated with, say, the carrier gas will still have information hidden from the particle class, and so some OOP benefits remain. In between these two extremes are implementations wherein the global collection of particles is grouped into clusters – possibly based on spatial proximity. Such groupings arise naturally in quantum turbulence and are discussed in Section 3, where we also demonstrate that high granularity simplifies local operations such as object removal.

Next we define data structure extent:

Definition 2: Given a set of objects covering the full data space, *data structure extent* is the continuously traversable length, ℓ_e, of the data in a given direction.

We refer to data as continuously traversable if we can access each successive datum by elementary operations such as incrementing a single index or accessing the target of a single pointer. For example, if the data space is covered by one object containing only a one-dimensional (1D) array, then the extent along the array's one dimension is the length of the array since we can traverse the data by incrementing the array's sole index. By contrast, if multiple objects cover the data space and we traverse the direction orthogonal to the object state space, then the extent is the number of continuously accessible objects. We refer to this extent as maximal, or global, when all objects can be so accessed. We demonstrate in Section 3 how maintaining global extent simplifies global operations such as time advancement.

2.2. Language choice

We present Fortran 90/95 code snippets in Section 3.3 below. Since this language choice is very unusual for OOP, we now explain its advantages. The reader may skip this section without loss of continuity.

Scientific software developers face a difficult choice between programming languages suited to mathematical modeling and those with explicit support for modern programming paradigms. The choice involves tradeoffs between development time and run-time performance. One must balance the explicit support for object-oriented programming (OOP) and garbage collection in Java against the performance penalties associated with Java's interpreted nature and its lack of operator overloading – a feature that numerical library developers have used to construct rich collections of mathematical class templates in C++ [1,5,21]. Likewise, one must balance the freedom allowed for pointer assignment in C++ with the difficulty it poses for optimizing compiler techniques such as register allocation. When considering Fortran 90/95, one must balance its rich mathematical constructs against its lack of explicit support for OOP.

Fortunately, many of the above choices will be less difficult soon as the Fortran 2000 standard provides

explicit support for OOP, including inheritance, polymorphism, and dynamic type allocation [13]. In the interim, a small set of researchers has developed techniques to facilitate OOP in Fortran 90/95 [1,7–10].

Advantages of Fortran 90/95 include its notational facilities for manipulating arrays, its intrinsic complex number type, and its ability to express both fine- and coarse-grained parallelism at the source code level [18]. Emulating these features in C++ can lead to a fourfold penalty in execution time [22]. Furthermore, Fortran 90/95 requires all variables accessed through a pointer be declared as pointer targets. This difference from C++ greatly aids optimizing compilers [18]. Finally, Decyk, Norton and Szymanski [10] showed that OOP abstractions lead to better cache utilization than does procedural programming in Fortran 77. They also demonstrated that OOP in Fortran 90 reduces execution times over C++ implementations. It even outperforms procedural programming in Fortran 77 on sufficiently large problems despite the run-time overhead associated with OOP.

3. Application

In the aforementioned simulation of solid particles, each particle moves independently of the others. There is therefore no reason to consider the relationship between them in the OOD process. Below we examine a case where connections exist between objects and demonstrate the influence of data structure granularity and extent on program design.

3.1. The physics of quantum turbulence

Simulating the dynamics of turbulent flow in superfluid liquid helium presents significant challenges in program design and computational cost. The memory requirements and execution time of turbulent flow simulations are well documented [20]. The addition of superfluidity does not appreciably alter the standard estimates; however, we argue it greatly influences program design. Before describing how, it will be useful to describe the physics.

Natural helium has two isotopic forms, ^{4}He (99.999%) and ^{3}He (0.001%). Liquid ^{4}He exists in two phases: Helium I and Helium II. Helium I is an ordinary liquid, while Helium II is a superfluid that forms below a temperature of approximately 2.17 K. Helium II flows without resistance through capillaries with diameters of the order of 10^{-6} m [3,4], yet its viscosity is not much less than that of helium gas [14]. Thus, Helium II is both viscous and inviscid. Tisza and Landau independently introduced the two-fluid model to explain this phenomenon [16,28]. They described Helium II as interpenetrating components of normal fluid and superfluid. The superfluid component behaves as a classical inviscid fluid. The normal fluid component behaves as a classical Newtonian fluid with a viscosity equal to that of Helium II.

Above a threshold driving velocity, He II contains vortices of quantum mechanical origin. Their unstable interactions are referred to as quantum turbulence or superfluid turbulence, and a network of such vortices is sometimes termed a vortex tangle.

3.2. Numerical simulation of quantum turbulence

The velocity of the normal fluid, $\mathbf{v}_n$, is governed by the Navier-Stokes equations supplemented by a differential statement of mass conservation for incompressible liquids:

$$\begin{aligned} \frac{\partial \mathbf{v}_n}{\partial t} + \mathbf{v}_n \cdot \nabla \mathbf{v}_n &= -\nabla p + \frac{1}{Re}\nabla^2 \mathbf{v}_n + \mathbf{f} \\ \nabla \cdot \mathbf{v}_n &= 0 \end{aligned} \quad (1)$$

where p is the mechanical pressure, Re is the Reynolds number, and $\mathbf{f}$ is the mutual friction force between the normal fluid component and the quantized vortices in the superfluid component. Equation (1) is approximated numerically by direct numerical simulation (DNS). In DNS, $\mathbf{v}_n$ is sampled on a grid with sufficient resolution to track all dynamically relevant scales of motion [19]. The resulting data are now widely recognized as being as accurate as experimental data at Reynolds numbers amenable to computation.

The velocity of the superfluid, $\mathbf{v}_s$, is governed by Eq. (1) without the viscous ($1/Re$) term. It can be shown that this implies the existence of a scalar potential such that $\mathbf{v}_s \equiv \nabla\phi$, where ϕ satisfies the Laplace equation $\nabla^2\phi = 0$. The linearity of the Laplace equation implies that the superfluid motion can be modeled by the superposition of discrete singularities. When the flow is rotational, these singularities take the form of vortex filaments and can be modeled by methods pioneered by Leonard [17].

In classical fluid mechanics, vortex filaments are an idealization. Viscous effects smooth out discrete singularities. In quantum turbulence, vortex filaments are real. The superfluid velocity matches that of an inviscid vortex down to a 1-Angstrom diameter filament core. Solutions to the nonlinear Schroedinger equation sug-

gest these cores are evacuated of any matter and thus have no internal structure [24]. Hence, vortex filaments are quite accurately modeled as 1D, curvilinear objects embedded in 3D space. Given that vorticity, $\nabla \times \mathbf{v}$, is tied to these vortices, Stokes' Theorem implies that filaments cannot end at a point in a simply connected region. Thus, each filament forms a loop, attaches to a boundary, or extends to infinity.

The equation of motion for each vortex filament can be shown to be of the form

$$\frac{d\mathbf{S}}{dt} = \mathbf{v}_s + \mathbf{v}_i + \alpha \mathbf{S}' \otimes (\mathbf{v}_n - \mathbf{v}_s - \mathbf{v}_i) - \alpha' \mathbf{S}' \otimes (\mathbf{v}_n - \mathbf{v}_s - \mathbf{v}_i)] \tag{2}$$

where $\mathbf{v}_s$ is the superfluid velocity imposed by boundary and initial conditions; $\mathbf{v}_i$ is the velocity induced by quantum vortices; $\mathbf{S}$ is the position of a point on the vortex filament; and $\mathbf{S}'$ is the first derivative of $\mathbf{S}$ with respect to arc-length along the vortex filament; and α and α' are temperature-dependent constants.

In the vortex filament method, a filament is represented by a series of mesh points along the vortex's centerline (see Fig. 2). The motions of these mesh points determines the filament motion and shape. In addition, two processes change the connectivity of the points. First, as the distance between mesh points changes, new points must be inserted and existing points removed to balance resolution requirements against computational cost and complexity. Second, when two filaments approach one another, their interaction tends to draw them closer, resulting in anti-parallel vortices [25,26]. Koplik and Levine showed that the filaments join and reconnect whenever two anti-parallel vortex filaments approach within a few core diameters of each other (see Fig. 3) [15].

Since the superfluid quantum vortices interact with the surrounding normal fluid, it is convenient to use the same algorithm to advance both fluids in time. Our DNS code for normal fluid turbulence employs the third-order Runge-Kutta scheme of Spalart, Moser and Rogers [27]. Applying this algorithm to Eq. (2) requires storing two copies of the position, $\mathbf{S}$, for each mesh point. (For simplicity, we store only one copy in the next section, corresponding to explicit Euler time advancement.) Also, mesh point ordering information must be stored for the reconnection and remeshing algorithms. Thus, the global state space is set by the physics. Different data abstractions represent different local partitionings of the state space. The next section presents choices near two granularity extremes and discusses the importance of data structure extent.

3.3. Data structure and software architecture

Central to the OOD process is the choice of abstract data types. Two alternatives will now be considered. First consider using three 1D arrays, `x(1:N), y(1:N)`, and `z(1:N)` to store mesh point position and two 1D arrays for connectivity information: `next(1:N)` and `last(1:N)`, where `N` is the number of mesh points. If these arrays are stored in one data structure, its state space will be of dimension $N_s = 5N$. In Fortran 90/95, the abstract data type is then of the form

```
TYPE vortex_tangle
  PRIVATE ! Prevent public access
  to the state.
  REAL, DIMENSION(N) :: x, y, z
  INTEGER, DIMENSION(N) :: next,
  last
END TYPE vortex_tangle
```

Only one instantiation is required, so the granularity, $g = 1$, is minimal and the abstraction is coarse-grained. However, the state remains private so some benefits of object orientation remain. The superfluid data is hidden from other classes, such as those describing the normal fluid. (Following the approach of Decyk, Norton and Szymanski [8], a class comprises a Fortran MODULE containing an abstract data type and procedures with an argument "this" of the corresponding type.)

At the beginning of a simulation, each mesh point is assigned a unique integer identifier, `ID`, from 1 to `N`. For a given `ID M`, the variables `x(M)`, `y(M)`, and `z(M)` store the corresponding mesh point's Cartesian coordinates, while `next(M)` and `last(M)` store the `ID` of the points immediately after and before `M`, respectively. Here, "after" and "before" refer to physical ordering on the vortex filament as opposed to logical order in the array. Most values stored in `next` and `last` range from 1 to N. A value of 0 for either can be used to indicate a filament end point, which can occur only at a boundary as explained in Section 3.2.

In a fine-grained data abstraction, one might define a node type as

```
TYPE node
  PRIVATE
  REAL :: x, y, z
  TYPE(node), POINTER :: next,
  last
END TYPE node
```

The `x, y` and `z` components of a node are the mesh point coordinates. The next and last components are pointers to other nodes. The node state space is five-dimensional, so N instantiations are required to cover

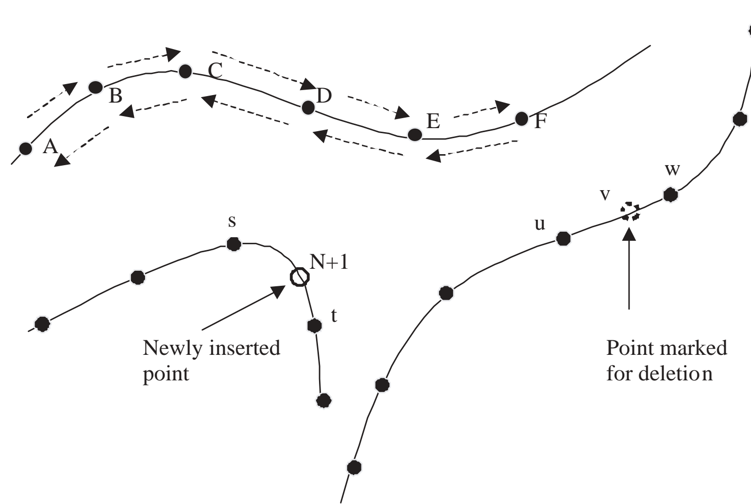


Fig. 2. Mesh points on three filaments, including one inserted and one to be deleted.

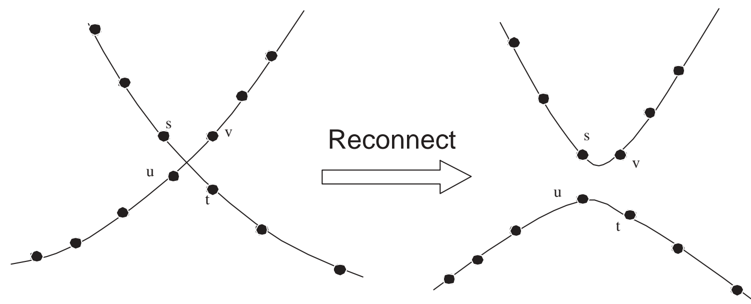


Fig. 3. Reconnection of vortex filaments.

the $5N$-dimensional data space. The granularity, $g = N$, is high and the abstraction is fine-grained. A vortex tangle could be implemented as a linked list of nodes as will be discussed below.

Figure 2 shows several linked nodes: `B%last` points to node `C` and `B%next` points to `A`. The pointers `next` and `last` establish a bi-directional linked list so when `B` is visited, `A` and `C` can be referenced as `B%next` and `B%last`, respectively. Then the component `x` of `A` can be accessed by `B%next%x`. A similar statement can be made for `C`, and `D` can be referenced as `B%last%last`. If `F%last` points to `A`, and `A%next` points to `F`, mesh points `A` to `F` form a vortex ring. If `A%last` and `F%next` are disassociated using the Fortran NULLIFY intrinsic, then points `A` to `F` form a vortex line ending at `A` and `F`.

Figure 4 shows how the above data structures fit into the overall software architecture as described using the Unified Modeling Language (UML) [2]. For the coarse-grained implementation of Fig. 4(a), the `vortex_tangle` contains array components covering the requisite superfluid data space as well as a field component representing the normal fluid velocity. The relationship between the `vortex_tangle` and field classes is indicated by the UML aggregation symbol. Note that the state of each class is private, but the functionality is public as indicated by the UML + and – symbols. Public functions in `vortex_tangle` include move, remesh and reconnect, which solve the quantum vortex equation of motion (2), remesh the vortex tangle, and reconnect vortex rings, respectively. The primary `field` component is a four-dimensional array storing the three Cartesian vector components of the normal fluid velocity field at each point on a rectangular grid. The only public function shown in `field` is `Navier_Stokes`, which solves Eq. (1). Finally, both classes implement certain generic, or polymorphic, behavior, including `new`, `delete`, and `print` for construction, deconstruction, and output, respectively. (Following Decyk, Norton and Szymanski [8],

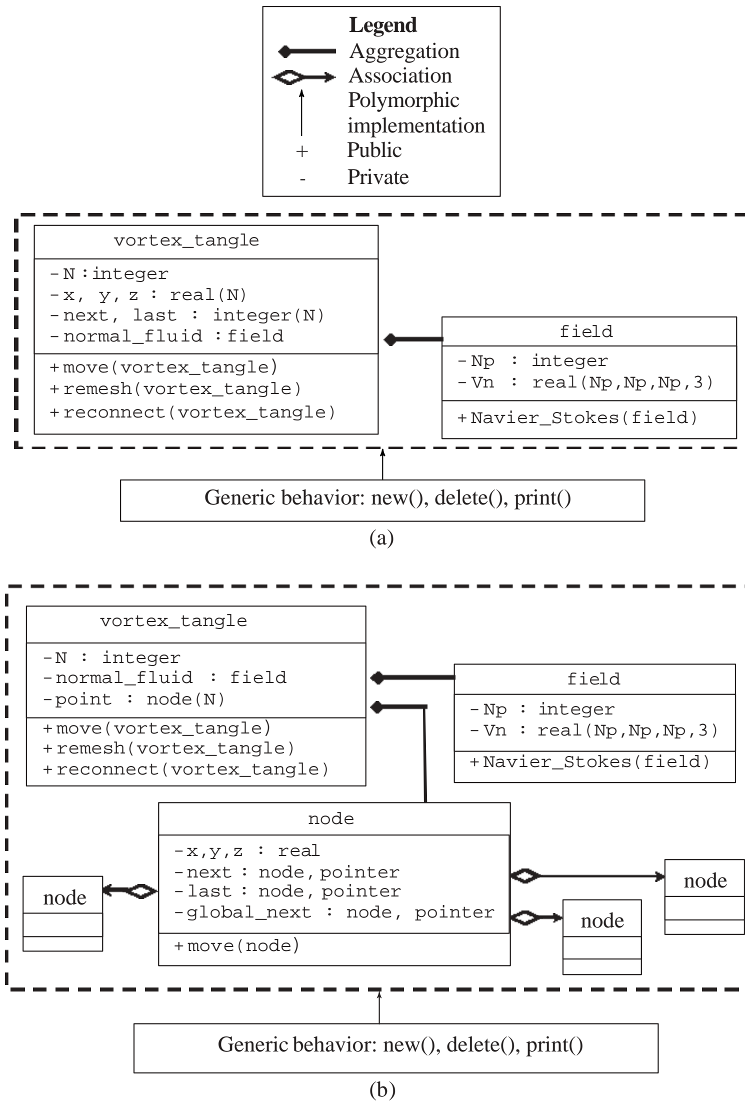


Fig. 4. Software architectures: (a) coarse-grained and (b) fine-grained.

constructors and destructors are implemented as Fortran module procedures with generic interfaces.)

For the fine-grained architecture of Fig. 4(b), the `vortex_tangle` class contains a `field` component and an array of node components. The array holds the initial set of nodes. However, since the node connectivity information is stored in pointers (`next` and `last`), a node can be inserted or removed without reallocating the array. The node component `global_next` will be discussed in Section 3.5. The relationship between `vortex_tangle` and the initial node array is indicated with the UML aggregation symbol; whereas the relationship between individual nodes in the linked list is shown with the UML association symbol. Other aspects of the design, including the polymorphic behavior and the `field` class, remain as in the coarse-grained design.

3.4. Local operations: Remeshing and reconnecting

Connections between vortex points change in two ways: remeshing and reconnecting. In remeshing, points are inserted or removed to balance resolution requirements against computational cost. In reconnections, two vortex lines are torn asunder and the loose ends are cross-connected. Each of these operations is local. For some local operations, the higher data local-

ization of the fine-grained abstraction requires simpler logic.

First, consider reconnecting points `u, v, s`, and `t`, where `u` is immediately after `v` initially, and `t` is immediately after `s`. After reconnection, `t` is immediately after `u`; whereas `v` is immediately after `s` (see Fig. 3). No operations on the position arrays are necessary. We assume a `vortex_tangle` has been instantiated as follows:

```
TYPE(vortex_tangle) ::  tangle
CALL new(tangle)
```

Then for the coarse abstraction, the operations on the connectivity arrays are

```
tangle%next(u)=t
tangle%last(t)=u
tangle%next(s)=v
tangle%last(v)=s
```

For the fine-grained design, operations on the connectivity pointers are

```
tangle%point(u)%next => point(t)
tangle%point(t)%last => point(u)
tangle%point(s)%next => point(v)
tangle%point(v)%last => point(s)
```

Comparison indicates the two designs require similar amounts of logic for reconnection.

Next consider inserting a new point with `ID` value `N+1` between points with `ID` values `t` and `s`, where `t` immediately precedes `s` spatially on a filament but not necessarily in storage (see Fig. 2). Assuming the constructor "`new`" creates space at the `N+1` slot in the `vortex_tangle` component arrays, the operations on the coarse-grained structure are

```
CALL new(tangle,N+1)
tangle%x(N+1)=X_new
tangle%y(N+1)=Y_new
tangle%z(N+1)=Z_new
tangle%last(N+1)=s
tangle%next(N+1)=t
tangle%next(s)=N+1
tangle%last(t)=N+1
```

In the pseudocode above, the declaration and constructor call are written for clarity. In an actual implementation, the constructor might determine the insertion point `ID` based on a free space list or, if new space is needed, it might add memory in large blocks, rather than just in the `N+1` slot.

For the fine-grained abstraction, point insertion requires redirecting pointers in the nodes with `ID` values `s`, `t` and `N+1`:

```
CALL new(tangle,N+1)
tangle%point(N+1)%x = X_new
tangle%point(N+1)%y = Y_new
tangle%point(N+1)%z = Z_new
tangle%point(N+1)%next => point(t)
tangle%point(N+1)%last => point(s)
tangle%point(s)%next => point(N+1)
tangle%point(t)%last => point(N+1)
```

Again we have assumed a constructor named new allocates memory if necessary, but we reiterate that the actual `ID` of the new point could vary depending on available free space. The logical complexity of point insertion appears to be the same for both abstractions.

Finally, consider removing a mesh point with `ID v` initially between points with `ID` values `u` and `w` (see Fig. 2). Here, "between" refers to physical ordering along a vortex filament as opposed to ordering in storage. Since the data associated with `v` is no longer needed, it is desirable to compress the array or to store a free space map. The following code adjusts the connectivity arrays, maps the free space, and compresses out the free space:

```
TYPE(vortex_tangle) ::  tangle
LOGICAL, DIMENSION(N) ::  free_space
= .FALSE.
tangle%next(u)= w
tangle%last(w)= u
free_space(v) = .TRUE.
DO i =v,N
  tangle%x(i) = tangle%x(i+1)
  tangle%y(i) = tangle%y(i+1)
  tangle%z(i) = tangle%z(i+1)
  tangle%next(i) = tangle%next(i+1)
  tangle%last(i) = tangle%last(i+1)
END DO
```

The above loop section shows that compression dominates the operation count for point insertion and affects all array elements after `v` in memory. Thus, point removal potentially has global effects on the coarse-grained data structure.

For the fine-grained abstraction, both compression and free space mapping are viable options since removing an object from a linked list has only local effects on the list. Below we redirect the relevant pointers and eliminate free space with the deconstructor `delete`:

```
LOGICAL, DIMENSION(N) ::  free_space
= .FALSE.
tangle%point(u)%next => point(w)
tangle%point(w)%last => point(u)
free_space(v) = .TRUE.
delete(tangle%point(v))
```

Hence, management of the fine-grained data structure is potentially much simpler than for the coarse-grained one.

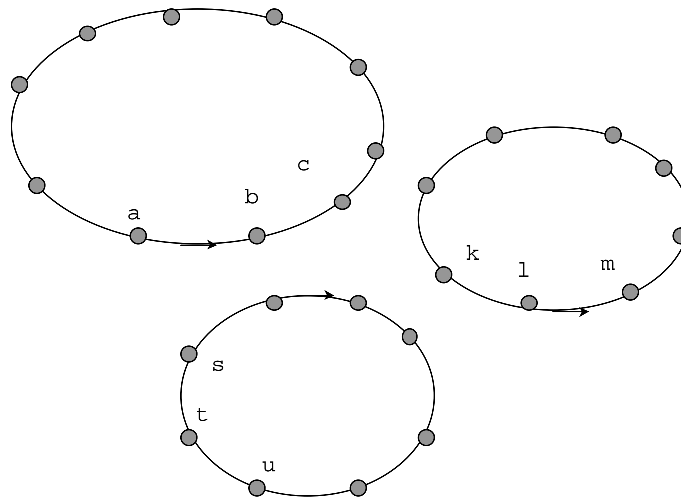


Fig. 5. Distinct vortex ring linked lists.

The above analysis suggests the fine-grained implementation potentially simplifies the code and facilitates better memory management. In each of the above cases, however, the resolution requirements and the physics predetermine a small set of points to be inserted, removed or reconnected. The next section demonstrates that the coarse-grained abstraction requires much simpler logic for operations visiting a broader collection of points unless attention is paid to maintaining global extent.

3.5. Global operations: Time advancement

In the context of the vortex filament model, remeshing and reconnecting are instantaneous events that scramble the links between a subset of the mesh points at the end of each time step. Time advancement itself, however, occurs in lock step for all mesh points. This begs the question of how to visit all points in the simulation after the links between them have been rearranged. Most importantly, how can we access points on newly formed closed loops? We address this issue for both abstractions below.

Even if there is only one vortex ring initially, reconnections can break it into many rings. For coarse-grained structures, this presents no difficulty. The coarse-grained structure's entire vortex tangle is continuously traversable via array index increments. Thus, the data structure extent is global.

For the fine-grained abstraction, the situation is considerably more complicated. Consider the configuration in Fig. 5. Beginning with point `a`, one can visit `b` by accessing `point(a)%next`. One then visits `c` by accessing `point(a)%next%next`. Eventually this process returns us to `a`, and therein lies the rub: how does one reach points `k`, `l`, and `m` on a separate vortex ring? One could visit each object in the point array, but this could be wasteful if the array contains substantial amounts of free space. (Note: for the fine-grained design, "compression" refers to freeing the memory in the objects representing deleted points without necessarily changing the point object array.)

The crux of this problem lies in the limited data structure extent resulting from reconnections that pinch off separate loops. The most straightforward way to increase the extent is to link the individual rings artificially as in Fig. 6, where designated jump points on each ring contain a link to a point on another ring. For example, one could redefine the `node` type as follows:

```
TYPE node
  PRIVATE
  REAL ::  x, y, z
  TYPE(node), POINTER ::  next,
  last, next_ring
END TYPE node
```

For most points, `next_ring` could be disassociated; whereas for one point on each ring, `next_ring` would target a point on another ring. For global operations, one could start with any point, traverse its ring fully, and then access the target of the ring's jump point via its `next_ring` component. This process could be repeated and end once the number of points visited equals the number of points in the vortex tangle.

The latter data structure, however, still does not have global extent because it uses one pointer (`next`) to traverse individual rings and another pointer

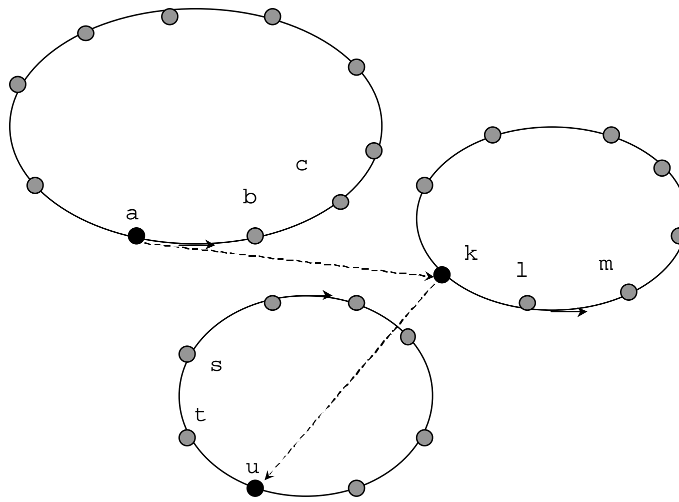


Fig. 6. Vortex rings with jump points (black).

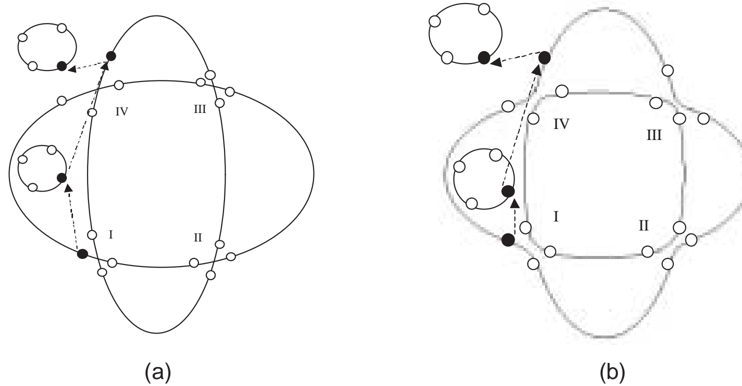


Fig. 7. Reconnection with uneven distribution of jump points: (a) before and (b) after.

(next_ring) to jump to other rings, so we cannot access all the data by repeating one elemental operation. Although a seemingly trivial distinction, the importance of this argument becomes clear when examining the four-vortex tangle of Fig. 7. Each ring has one designated jump point shown in black. Two rings are close enough for reconnection at the four points marked with Roman numerals. The resulting reconnections create two new rings. One inherits two jump points. Another gets none. Clearly one jump point is now redundant. More importantly, one vortex ring has now been completely disconnected from the tangle and will subsequently be lost from the simulation. The most straightforward solution is costly: for each reconnection involving a jump point, the resulting rings must be traversed to identify redundancies. When a redundancy is identified, one jump point on the corresponding ring must become a generic point, while a generic point on the other loop must become a jump point. For a dense tangle, the resulting computational overhead is cost prohibitive. In fact, since all point pairs can potentially reconnect, we have likely traded an $O(N^2)$ debugging problem for an $O(N^2)$ operation count. Worse yet, other topological changes require different logic. For example, if a reconnection combines two rings into one, a redundant jump point must be eliminated but no new jump point need be created.

Ultimately the simplest solution is to create a global list of all mesh points that bears no relation to their physical connectivity. One could implement such a global list as follows:

```
TYPE node
  PRIVATE
  REAL ::  x, y, z
  TYPE(node), POINTER ::  next,
  last, global_next, global, last
```

```
END TYPE NODE
```

The resulting design couples the high granularity of the fine-grained abstraction with the global extent of the coarse-grained abstraction. Such a design reduces the computational overhead associated with certain local operations such as point removal, while simultaneously reducing the overhead associated with global operations such as time advancement. Memory associated with removed points can be freed without compressing the associated arrays. Each point in the tangle can be visited by repetitively accessing the `global_next` pointer targets, ending after the number of points visited equals the number of points in the simulation.

4. Conclusion

This paper has focused on a class of scientific simulations involving large sets of interconnected objects. In particular, we have explored the implications of dynamically reconfiguring the links between these objects. Data structure granularity and extent have been introduced as useful metrics for determining which data abstractions lead to less computational overhead. Granularity was defined as the number of instantiations required to cover the full data space. Extent is defined in a given direction as the continuously traversable length of the data that are accessible through elemental operations, such as incrementing a single array index or accessing a single pointer target. In the direction orthogonal to the object state space, the extent is the number of continuously traversable objects. When all objects are so accessible, the extent is global.

Coarse-grained and fine-grained abstractions have been presented for a representative problem: quantum turbulence simulation. At one level of approximation, quantum turbulence can be modeled as a tangled collection of vortex filaments interacting with each other and with surrounding normal fluid. The filaments can be represented by a discrete set of connected points. As the simulation progresses, points must be inserted and removed and filaments must be torn and reconnected. In addition to these local operations, such global operations as time advancement require visiting all points.

We have shown that increasing data structure granularity reduces the complexity of certain local operations at the potential expense of increased global operation complexity. We have further demonstrated that maintaining global extent reduces the global operation complexity, making fine-grained abstractions ultimately more attractive than their coarse-grained counterparts.

Acknowledgements

This work was supported in part by Award No. 02061 52 from the National Science Foundation and Award No. 64444-00 33 from the Professional Staff Congress of the City University of New York (CUNY). The authors would like to thank Prof. Joel Koplik of Levich Institute at the City College of CUNY for many useful discussions related to superfluidity, Dr. Charles Norton of Jet Propulsion Laboratory for comments regarding the manuscript, and Dr. Nagi Mansour of NASA Ames Research for supporting the revision of this article during the first author's tenure as a NAFEO Faculty Fellow at Ames.

The first author also thanks Prof. Oyekunle Olukotun of Stanford University for suggesting software design metrics as a research subject and Dr. John J. Kenney of Spirent PLC for introducing him to software architecture.

References

[1] E. Akin, *Object-oriented programming via Fortran 90/95*, Cambridge University Press, Great Britain, 2003.

[2] S.S. Alhir, *UML in a Nutshell*, O'Reilly Media, Inc., 1998.

[3] J.F. Allen and H. Jones, *New Phenomena with Heat Flow in Helium II*, Nature, no. 3562, pp. 243–244.

[4] J.F Allen and A.D. Misener, Viscosity of liquid helium below the lambda-point, *Nature* **141** (1938), 74–75.

[5] J.J. Barton and L.R. Nackman, *Scientific and engineering C++: an introduction with advanced examples*, Addison-Wesley, Massachusetts, 1994.

[6] G. Buzzi-Ferraris, *Scientific C++*, Addison-Wesley, Massachusetts, 1998.

[7] V.K. Decyk, C.D. Norton and B.K. Szymanski, Expressing object-oriented concepts in Fortran 90, *ACM Fortran Forum* **15** (1997), 13–18.

[8] V.K. Decyk, C.D. Norton and B.K. Szymanski, How to express C++ concepts in Fortran 90, *Scientific Programming* **6** (1997), 363–390.

[9] V.K. Decyk, C.D. Norton and B.K. Szymanski, High performance object-oriented programming in Fortran 90, in: *Proc. Eighth SIAM Conference On Parallel Processing for Scientific Computing March 14–17, 1997*, M. Heath et al., eds, Minnesota, 1997.

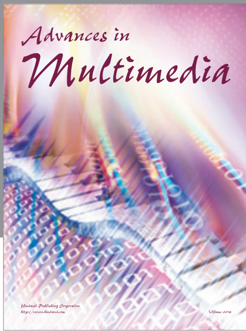
[10] V.K. Decyk, C.D. Norton and B.K. Szymanski, How to support inheritance and run-time polymorphism in Fortran 90, *Computer Physics Communications* **115** (1998), 9–17.

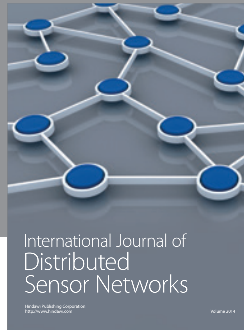
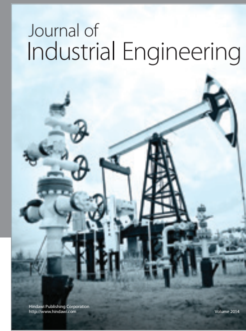
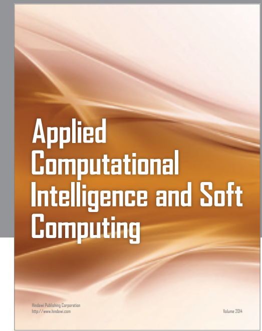
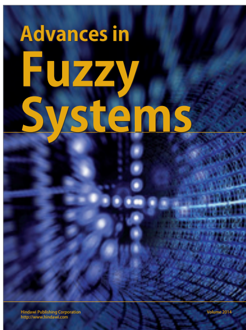
[11] G. Golub and J.M. Ortega, *Scientific computing: an introduction with parallel computing*, Academic Press, Missouri, 1993.

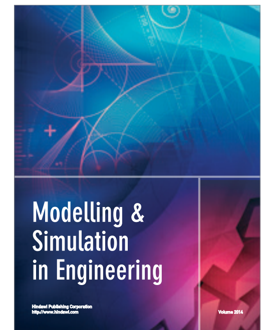
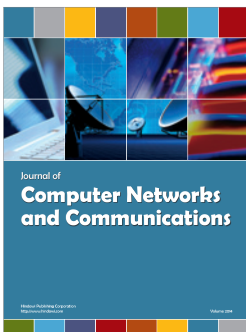
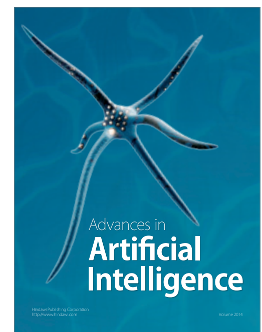
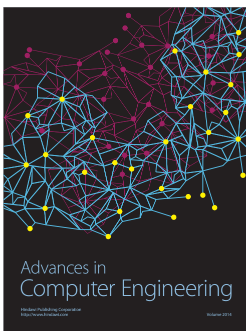
[12] K. Hwang and Z. Xu, *Scalable parallel computing: technology, architecture, programming*, McGraw-Hill, New York, 1998.

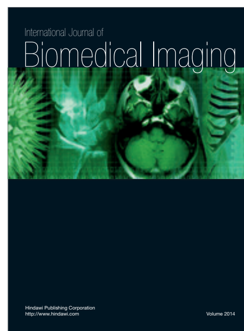
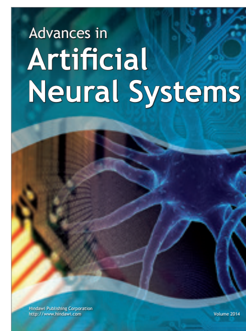
[13] J3 Fortran Standards Technical Committee, ISO/IEC 1539 Working Draft, International Organization for Standards/International Electrotechnical Committee, Geneva, Switzerland. 2001.

[14] P. Kapitza, Superfluidity observed, *Nature* **141** (1938), 74–75.
[15] J. Koplik and H. Levine, Vortex reconnection in superfluid helium, *Physical Review Letters* **71** (1993), 1375–1378.
[16] L.D. Landau, The theory of superfluidity of Helium II, *Journal of Physics USSR* **5**(71) (1941), 71.
[17] A. Leonard, Vortex Methods for flow simulation, *Journal of Computational Physics* **289** (1980), 289–335.
[18] M. Metcalf and J. Reid, *Fortan 90/95 Explained*, Oxford University Press, Great Britain, 1999.
[19] P. Moin and K. Mahesh, Direct numerical simulation: a tool in turbulence research, annual, *Annual Review of Fluid Mechanics* **30** (1999), 539–578.
[20] S.B. Pope, *Turbulent Flows*, Cambridge University Press, Great Britain, 2000.
[21] W.H. Press, W.T. Vetterling, B.P. Flannery and S.A. Teukolsky, *Numerical Recipes in C++: The Art of Scientific Computing*, 2nd Edition, Cambridge University Press, Great Britain, 2002.
[22] A.D. Robison, C++ gets faster for scientific computing, *Computers in Physics* **10** (1996), 458–462.
[23] D.W.I. Rouson and J.K. Eaton, On the preferential concentration of solid particles in a turbulent channel flow, *Journal of Fluid Mechanics v* **428** (2001), 149–159.
[24] D.C. Samuels, Vortex filament methods for superfluids, in: *Quantized Vortex Dynamics and Superfluid Turbulence*, C.F. Barenghi, R.J. Donnelly and W.F. Vinen, eds, Telos Press, New York, 2001.
[25] K.W. Schwarz, Three-dimensional vortex dynamics in superfluid 4He: line-line and line-boundary interactions, *Physical Review B* **31** (1985), 5782–5804.
[26] K.W. Schwarz, Three dimensional vortex dynamics in superfluid 4He: homogeneous superfluid turbulence, *Physical Review B* **38** (1988), 2398–2417.
[27] P.R. Spalart, R.D Moser and M.M. Rogers, Spectral methods for the Navier-Stokes equations with one infinite and two periodic directions, *Journal of Computational Physics* **6** (1991), 297–324.
[28] L. Tisza, The λ-transition explained, *Nature* **141** (1938), 643–644.

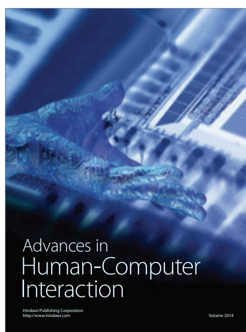
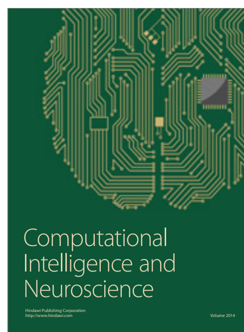

Advances in
Multimedia

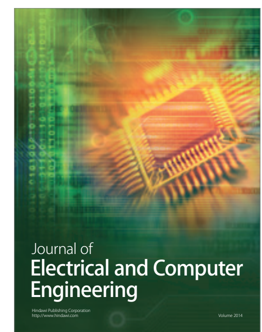
The Scientific
World Journal

International Journal of
Distributed
Sensor Networks

Journal of
Industrial Engineering

Applied
Computational
Intelligence and Soft
Computing

Advances in
Fuzzy
Systems

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

Modelling &
Simulation
in Engineering

Journal of
Computer Networks
and Communications

Advances in
Artificial
Intelligence

Advances in
Computer Engineering

International Journal of
Computer Games
Technology

International Journal of
Biomedical Imaging

Advances in
Artificial
Neural Systems

Advances in
Software Engineering

Journal of
Robotics

Advances in
Human-Computer
Interaction

Computational
Intelligence and
Neuroscience

International Journal of
Reconfigurable
Computing

Journal of
Electrical and Computer
Engineering