Bioinformatics research depends increasingly on high-performance computation and large-scale data storage. Also, datasets are often complex, heterogeneous, or incomplete. These two aspects make bioinformatics appropriate for visual analytics (VA). Many powerful scientific toolsets are available, including software libraries such as SciPy; specialized visualization tools such as Chimera; and scientific workflow tools such as Taverna, Galaxy, and the Visualization Toolkit (VTK). Some of them can handle large datasets. Others—typically, those originally designed for small, local datasets—haven’t been updated to handle recent advances in data generation and acquisition.

To help fill this technological gap, we developed DIVE (Data Intensive Visualization Engine), which makes big-data VA approaches accessible to scientific researchers (see Figure 1). DIVE employs an interactive data pipeline that’s extensible and adaptable. It encourages multiprocessor, parallelized operations and high-throughput, structured data streaming. DIVE can act as an object-oriented database by joining multiple disparate data sources. And, although we present bioinformatics applications here, DIVE can handle data from many domains.

The DIVE Architecture
DIVE is an API whose primary component is the data pipeline, which can stream, transform, and visualize datasets at interactive speeds. The pipeline can be extended with plug-ins; each plug-in can operate independently on the data stream.

Data exploration is supported through command-line interfaces, GUIs, and APIs. Figure 2 shows an example DIVE application. All these interfaces support scripting interaction. DIVE also supports typed events, letting users trigger targeted analyses from a point-and-click interface. Programmatically, DIVE inherits much functionality from the .NET environment, as we discuss later.

Finally, DIVE is domain independent and data-agnostic. The pipeline accepts data from any domain, provided an appropriate input parser is implemented. Currently supported data formats include SQL, XML, comma-and tab-delimited files, and several other standard file formats (see Figure 3).

Data Representation
Ontologies (see the related sidebar) are gaining popularity as a powerful way to organize data. We developed DIVE’s core data representation with ontologies in mind. The fundamental data unit in DIVE is the datanode. Datanodes somewhat resemble traditional object instances from object-oriented (OO) languages such as C++, Java, or C#. They’re typed, contain strongly typed properties and methods, and can exist in an inheritance hierarchy.

However, datanodes extend that traditional model. They can exist in an ontological network or graph; that is, multiple relationships beyond simple type inheritance can exist between datanodes. DIVE implements these relationships with dataedges, which link datanodes. Dataedges themselves are implemented by datanode objects and consequently might contain properties, methods,
and inheritance hierarchies. Because of this basic flexibility, DIVE can represent arbitrary, typed relationships between objects, objects and relationships, and relationships and relationships.

Datanodes are also dynamic; every method and property can be altered at runtime, adding much flexibility to the system. (The DIVE pipeline contains various data integrity mechanisms to prevent unwanted side effects, as we discuss later.) The inheritance model is also dynamic; as a result, objects can gain and lose type qualification and other inheritance aspects at runtime. This allows runtime classification schemes such as clustering to be integrated into the object model.

Finally, datanodes provide virtual properties. These properties are accessed identically to fixed properties but store and recover their values through arbitrary code instead of storing data on the datanode object. Virtual properties can extend the original software architecture’s functionality, allowing data manipulation, as we describe later.

Dataedges implement multiple inheritance models. Besides the traditional is-a relationship in OO languages, ontological relationships such as contains, part-of, and bounded-by can be expressed. Each of these relationships can support varying levels of inheritance (see Table 1):

- With OO inheritance, which is identical to OO languages such as C++ and Java, subclasses inherit the parent’s type, properties, and methods.
- With type inheritance, subclasses inherit only the type.
- With property inheritance, subclasses inherit only the properties and methods.

Like OO language objects, property-inheritance subclasses can override superclass methods and properties with arbitrary transformations. Similarly, type-inheritance subclasses can be cast to superclass types. Because DIVE supports not only multiple inheritance but also multiple kinds of inheritance, we implement casting by traversing the dataedge ontology. Owing to the coupling of the underlying data structure and ontological representation, every datanode and dataedge is implicitly part of a systemwide graph. This means we can use graph-theoretical methods to analyze both
Ontologies

An ontology is a semantically and syntactically formal structure for organizing information. As organized datasets’ size and complexity have grown, so has the need for formal semantics and syntax. In particular, the need for such formalisms is driven by the desire to handle these large, complex datasets programmatically. Ontologies enforce a strict formalism that guarantees that structured information is both meaningful and extensible. Once this is established, such information can be clearly reasoned with, built on, and discussed. An ontology can be represented as a graph in which nodes represent specific concepts and edges represent specific relationships.

Efforts such as the Semantic Web hold the promise of establishing a global formal ontology of everything. Although this is desirable, much work remains in smaller, more localized knowledge domains such as biology. For example, the US National Center for Biomedical Ontology’s BioPortal (http://bioportal.bioontology.org) currently indexes more than 300 ontologies. Projects such as BioPortal demonstrate that ontologies are becoming increasingly popular and that modern data analysis tools must be able to handle large sets of ontologically structured data.

References


Data Import

Data must be imported into DIVE before they are accessible to the DIVE pipeline. In many cases, DIVE’s built-in functionality handles this import. In the case of tabular data or SQL data tables, DIVE constructs one datanode per row, and each datanode has one property per column. DIVE also supports obtaining data from Web services such as the Protein Data Bank. Once DIVE obtains the data, simple mechanisms establish relationships between datanodes. Later, we describe a more sophisticated way to acquire structured data that uses native object parsing.

The Pipeline

DIVE’s pipeline is comparable to Taverna, Pipeline Pilot (http://accelrys.com/products/pipeline-pilot), Cytoscape, Galaxy, and, most similarly, the VTK. Although all these platforms are extendable, two factors led us to develop DIVE. This first was platform considerations, which we discuss later. The second was our focus on agile data exploration instead of remote, service-based workflows. Fortunately, all these platforms have made interop-
ability a priority. So, we can leverage Cytoscape’s graph capabilities or the VTK’s visualization capabilities while maintaining DIVE’s benefits by connecting their respective pipelines.

In the DIVE pipeline, plug-ins create, consume, or transform data. These plug-ins are simply compiled software libraries whose objects inherit from a published interface. The DIVE kernel automatically provides subsequent plug-in connectivity, pipeline instantiation, scripting, user interfaces, and many other aspects of plug-in functionality. Plug-ins move data through pins much like an integrated circuit: data originate at an upstream source pin and are consumed by one or more downstream sink pins. Plug-ins can also move data by broadcasting and receiving events. Users can save pipeline topologies and state to a file and share them.

When DIVE sends a datanode object through a branching, multilevel transform pipeline, it must maintain the datanode’s correct property value at every pipeline stage. Otherwise, a simple plug-in that scaled incoming values would scale all data, everywhere in the pipeline. The naïve option is to copy all datanodes at every pipeline stage, but this is extremely CPU- and memory-intensive and dramatically worsens the user experience.

To address this problem, DIVE uses read and write contexts. Essentially, this creates a version history of each transformed value. We key the history on each pipeline stage such that each plug-in reads only the appropriate values and not, for instance, downstream values or values from another pipeline branch. This approach maintains data integrity in a branching transform pipeline. It’s also parallelizable. In addition, it keeps an accurate account of the property value at every stage in the pipeline, with a minimum of memory use. Finally, it’s fast and efficient because the upstream graph traversal is linear and each value lookup occurs in constant time.

**Software Engineering Considerations**

We designed DIVE to provide a dynamic, scalable VA architecture. Although such an architecture doesn’t require a specific platform, we built DIVE on the Microsoft Windows platform and .NET framework because of several significant built-in capabilities. These capabilities include the

---

**Figure 2. The DIVE GUI with the Protein Dashboard pipeline loaded.** At the top is a data loader with which users can load and interact with protein structures and molecular-dynamics trajectories (see the “Molecular Dynamics” sidebar) from different sources. On the lower left is an interactive 3D rendering of a protein molecule, rendered using a cartoon representation for the protein backbone and a ball-and-stick representation for a subset of atoms selected through the scripting window at the bottom. On the lower right is one of many linked interactive charts that stream synchronized data from the Dynamicomics database.
Researchers commonly use molecular dynamics (MD) simulations\(^1\) to study protein structure and dynamics. Proteins are complex molecules consisting of amino acids (residues). Contacts between the constituent atoms exist when they’re within a defined distance from one another.

Proteins are responsible for much of the functional and structural activity in living tissue. In the human body, protein function is involved in such areas as muscular structure, metabolism, immune response, and reproduction. So, understanding how proteins work is critical to advancing the science of human health. An interesting facet of protein biology is that structure equals function; what a protein does and how it does it is intrinsically tied to its 3D structure (see Figure A).

During an MD simulation, scientists simulate interatomic forces to predict motion among the atoms of a protein and its environment (see Figure A). In most cases, the environment is water molecules, although scientists can alter this to investigate different phenomena. The physical simulation is calculated using Newtonian physics; at specified time intervals, the simulation state is saved. This produces a trajectory, a series of structural snapshots reflecting the protein’s natural behavior in an aqueous environment.

MD is useful for three primary reasons. First, like many in silico techniques, it allows virtual experimentation; scientists can simulate protein structures and interactions without the cost or risk of laboratory experiments. Second, modern computing techniques allow MD simulations to run in parallel, enabling virtual high-throughput experimentation. Third, MD simulation is the only protein analysis method that produces sequential time-series structures at both high spatial and high temporal resolution. These high-resolution trajectories can reveal how proteins move, a critical aspect of their functionality.

However, MD simulations can produce datasets considerably larger than what most structural-biology tools can handle. As computers become more powerful, MD simulations’ size and resolution are increasing. So, the logistical challenges of storing, analyzing, and visualizing MD data require researchers to consider new analysis techniques.

At the University of Washington’s Daggett laboratory, we’re studying protein dynamics as part of the Dynameomics project.\(^2\) This project aims to characterize the dynamic behaviors and folding pathways of topological classes of all known protein structures. So far, the project has generated hundreds of terabytes of data consisting of thousands of simulations and millions of structures, as well as their associated analyses. We store these data in a distributed SQL data warehouse. This warehouse currently holds \(10^4\) times as many protein structures as the Protein Data Bank,\(^3\) the primary repository for experimentally characterized protein structures. Dynameomics is currently the largest database of protein structures in the world.

### References


---

**Molecular Dynamics**

**Figure A.** Solvating and simulating a protein using molecular dynamics. (1) An all-atom depiction of a protein with a transparent surface. (2) The same protein solvated and in a water box. (3) Three structures of interest selected from a trajectory containing more than 51,000 frames. The red area shows the protein’s functional site and how it closes over time.

---

dynamic-language runtime, expression trees, and Language-Integrated Query (LINQ). .NET also provides coding features such as reflection, serialization, threading, and parallelism. Extensive documentation and details of these capabilities are at www.microsoft.com/net.
Many of these capabilities directly affect DIVE’s functionality and user experience. Support for dynamic languages allows flexible scripting and customization that would be difficult in less expressive platforms. These components are crucial for both the data model we described earlier and the scripting capabilities we describe later. Furthermore, LINQ is useful in a scripted data-exploration environment. Expression trees and reflection provide the underlying object linkages for the DIVE object parser (which we also describe later), and DIVE streaming heavily uses the .NET framework’s threading libraries. Finally, because .NET supports 64-bit computations and simple parallelism, DIVE can transparently scale with processor capabilities.

.NET also supports not only Microsoft-specific languages such as C#, Visual Basic, and F# but also more general languages such as Python and C++. This lets us author DIVE plug-ins in many languages. In addition, we can use these languages to develop command-line, GUI, and programmatic tools that embed and drive the DIVE kernel (as our case study shows later). .NET’s wide user base also provides multiple external libraries with which to jump-start our development efforts, including molecular visualizers, clustering and analysis packages, charting tools, and mapping software. In particular, one such library is the VTK, wrapped by the ActiViz .NET API (see www.kitware.com/opensource/avdownload.php).

Finally, for our Dymeomics project (see the “Molecular Dynamics” sidebar), we store data in a Microsoft SQL Server data warehouse. So, it made sense to adopt a software platform with deep support for these data services.

Object Parsing
Module-management systems such as the Java-based OSGi\textsuperscript{12} support module life-cycle management and service discovery. However, module authors often must be aware of the module-management system when creating a module. We aimed to make .NET assemblies written without a priori knowledge of DIVE accessible to the ontological data representation. We also didn’t require the life-cycle services of such module-management systems. So, we developed the DIVE object parser.

The parser automatically generates datanodes and dataedges from any .NET object or assembly (see Figure 4). Using reflection and expression trees, it consumes .NET object instances and translates them into propertied datanodes and dataedges. Usage patterns typically involve standard object creation by library-aware code, followed by automated object parsing and injection into the DIVE pipeline.

---

Figure 3. The DIVE architecture. The DIVE kernel acts as both a software library and runtime environment. In both cases, DIVE can import and export data and functionality from a variety of sources. Pipeline plug-ins use DIVE primarily as a software library, exploiting DIVE’s data-handling capabilities. DIVE tools are applications that instantiate and launch a DIVE pipeline for a specific analysis task. DIVE supports multiple types of interfaces.

Generic rules define the mapping between the .NET object hierarchy and DIVE data structures. Generally, complex objects such as classes are parsed into datanodes, whereas built-in .NET system objects, primitive fields, primitive properties, and methods with primitive return types are translated into properties on those datanodes. Interfaces, virtual classes, and abstract classes are all translated into datanodes. The .NET inheritance and member relationships are interpreted as OO and property inheritance dataedges, respectively; these dataedges then connect the datanode hierarchy.
Using this approach, the object parser recursively produces an ontological representation of the entire .NET instance hierarchy in DIVE. Additional rules handle other program constructs. For example, the parser translates static members into a single datanode. Multiple object instances with the same static member all map to a single, static datanode instance in the DIVE data structure. Public objects and members are always parsed, whereas private members, static objects, and interfaces are parsed at the user’s discretion.

Throughout this process, no data values are copied to datanodes or dataedges. Instead, dynamically created virtual properties link all datanode properties to their respective .NET members. So, any changes to the runtime .NET object instances are reflected in their DIVE representations. Similarly, any changes to datanode or dataedge proper-
ties propagate back to their .NET object instance counterparts. This lets DIVE interactively operate on any runtime .NET object structure.

With object parsing, users can import and use any .NET object without special handling. Furthermore, as we discussed before, the .NET application’s architect doesn’t need to be aware of DIVE to exploit its VA capabilities. For example, assume we have a nonvisual code library that dynamically simulates moving bodies in space (this example is available with the DIVE program download at www.dynameomics.org/dive). A DIVE plug-in, acting as a thin wrapper, can automatically import the simulation library and add runtime visualizations and interactive analyses. As the simulation progresses, the datanodes will automatically reflect the changing property values of the underlying .NET instances. Through a DIVE interface, the user could change a body’s mass. This change would propagate back to the runtime instance and immediately appear in the visualization. This general approach is applicable to many specialized libraries, taking advantage of their efficient data models. We describe an example of this later.

**Scripting**

To let users rapidly interact with the DIVE pipeline, plug-ins, data structures, and data, DIVE supports two basic types of scripting: **plug-in scripting** and **mscripting** (microscripting). In the DIVE core framework, C# is the primary scripting language. Externally, DIVE can host components written in any .NET language and, conversely, can be hosted by any .NET environment. Here we focus on C# scripting.

Both scripting types are controlled in the same way. The user script is incorporated into a larger, complete piece of code, which is compiled during runtime using full optimization. Finally, through reflection, the compiled code is loaded back into memory as a part of the runtime environment. Although this approach requires time to compile each script, the small initial penalty is typically outweighed by the resulting optimized, compiled code. Both scripting types, particularly mscripting, can work on a per-datanode basis; optimized compilation helps create a fast, efficient user experience.

Plug-in scripting is simpler and more powerful than mscripting and is the most similar to existing analysis tools’ scripting capabilities. Through this interface, the user script can access the entire .NET runtime, the DIVE kernel, and the specific plug-in.

We developed mscripting to give complete programmatic control to power users and simple, intuitive control to casual users. Essentially, mscripting is an extension of plug-in scripting in which DIVE writes most of the code. The user needs to write only the right-hand side of a C# lambda function. Here’s a schematic of this function:

```csharp
func(datanode dn) => ???;
```

The right-hand side is inserted into the function and compiled at runtime. The client can provide any expression that evaluates to an appropriate return value. Table 2 shows mscripting examples.

**Data Streaming**

DIVE supports the following two SQL data-streaming approaches.

*Interactive SQL*

This approach (see Figure 5a) handles the immediate analysis of large, nonlocal datasets; it’s for impromptu, user-defined dynamic SQL queries.

Interactive SQL employs user input to build an SQL query at runtime. DIVE facilitates this; for example, DIVE events can be thrown in response to mouse clicks or slider bar movements. Upon receiving these events, a DIVE component can construct the appropriate SQL query (which can consist of both data queries and analysis function execution), send it to the SQL database, and parse the resulting dataset. Depending on the query’s size and complexity, this approach can result in user-controlled SQL analysis through the GUI at interactive rates.

*Pass-Through SQL*

This approach (see Figure 5b) handles interactive analysis of datasets larger than the client’s local memory; it’s for streaming complex object models across a preset dimension.

Pass-through SQL accelerates the translation of SQL data into OO structures by shifting the location of values from the objects themselves to a backing store, an in-memory data structure. A backing store is essentially a collection of tables of instance data; each table contains many instance values for a single object type. Internally, object fields and properties have pointers to locations in backing-store tables instead of local, fixed values. A backing-store collection comprises all the tables for the object instances occurring at the same point, or frame, in the streaming dimension.

Once this approach creates a backing store, it generates copies of the backing-store structure with a unique identifier for each new frame. It then inserts instance values for new frames into the corresponding backing-store copy. This reduces
Visual Analytics for Biological Data

the loading of instance data to a table-to-table copy, bypassing the parsing normally required to insert data into an OO structure. This approach also removes the overhead of allocating and de-allocating expensive objects by reusing the same object structures for each frame in the streaming dimension.

Pass-through SQL enables streaming through a buffered set of backing stores representing frames over the streaming dimension. A set is initially populated client-side for frames on either side of the frame of interest. Buffer regions are defined for each end of this set. Frames in the set are immediately accessible to the client. When the buffer regions’ thresholds are traversed during streaming, a background thread is spawned to load a new set of backing stores around the current frame. If the client requests a frame outside the loaded set, a new set is loaded around the requested frame. Loaded backing stores no longer in the streaming set are deleted from memory to conserve the client’s memory.

A Case Study

A major research focus in the University of Washington’s Daggett laboratory is the study of protein structure and dynamics through molecular dynamics (MD) simulations using the Dynamosics data warehouse (see the “Molecular Dynamics” sidebar). The Dynamosics project contains much more simulation data than what typical, domain-specific tools can handle. Analysis of this dataset was the impetus for creating DIVE.

One of the first tools built on the DIVE platform was the Protein Dashboard, which provides interactive 2D and 3D visualizations of the Dynamosics dataset. These visualizations include interactive explorations of bulk data, molecular-visualization tools, and integration with external tools such as Chimera.

A tool implemented independently of DIVE and the Protein Dashboard is the Dynamosics API. Written in C#, it establishes an object hierarchy, provides high-throughput streaming of simulations from the Dynamosics data warehouse, contains domain-specific semantics and data structures, and provides multiple domain-specific analyses. However, it’s designed for computational efficiency and doesn’t specify any data visualizations or user interfaces.

We wanted to use the Dynamosics API’s sophisticated data handling and streaming while keeping the Protein Dashboard’s interactive vi-

| Table 2. μ-scripting examples. | | |
|---|---|---|---|
| Argument | Return type | Code | Comments |
| datanode dn | double | 3 | This is the simplest case of scripted numeric input. |
| | | dn.X | This is a simple per-datanode μ-script. |
| | | Math.Abs(dn.X) | The μ-script is given access to the full .NET library. |
| int | dn.X > 0 ? 1 : -1; | | Simple syntax can be powerful. |
| void | bool | | |
| | | { | Any .NET code is allowed, including complex, multistatement functions. |
| | | int hour = DateTime.Now.Hour; | |
| | | return hour < 12; | |
| datanode[] | Dynamic set | from dn in dns group dn by Math.Round(dn.X, 2) into g select new | This creates a histogram based on the datanode objects’ “X” property. |
| | | { | |
| | | bin = g.Key; population = g.Count(); | |
| | | from dn in dns where dn.X > Math.PI && dn.is_Superclass && dn.Func() == true select dn; | This filters a subset of datanodes on the basis of properties, methods, and inherited type. |
| | | from dn1 in dnSet1 join dn2 in dnSet2 on dn1.X equals dn2.X select new (X = dn1.X, Y = dn2.Y) | DIVE can act as an object-oriented database by joining multiple potentially disparate datasets. |
sualization and analysis, without reimplementing DIVE’s API. Through the object parser, DIVE can integrate and use the Dynameomics API structures without changing its own API. This process creates strongly typed objects, including Structure, Residue, Atom, and Contact as datanodes, with each datanode containing properties defined by the Dynameomics API. Semantic and syntactic relationships specified in the API are similarly translated into dataedges. Once processed, these datanodes and dataedges are available to the DIVE pipeline, indistinguishable from any other datanodes or dataedges. Figure 6 diagrams this dataflow.

With the Dynameomics data and semantics available to the DIVE pipeline, we can apply a VA approach to the Dynameomics data. As before, we can use the Protein Dashboard to interact with and visualize the data. However, because the data flows through the Dynameomics API, wrapped by DIVE datanodes and dataedges, we can load multiple protein structures from different sources, including the Protein Data Bank, to align the structures, and analyze them in different ways.

Furthermore, because the Protein Dashboard has access to additional data from the Dynameomics API, its own utility increases. For instance, it’s useful to color protein structures on the basis of biophysical properties such as solvent-accessible surface area or deviation from a baseline structure. By streaming the data through the pipeline, we can watch these properties (many of which were accessed through the data’s inheritance hierarchy) change over time.

**Discussion**

By necessity, most data analysis tools such as DIVE have some functional overlap; basic visualization and data analysis routines are simply required for functionality. However, several DIVE features are both novel and useful, particularly in a big-data, interactive setting. Here we discuss these features, their benefits, and how we see them integrating with existing technologies.

**Ontological Data Structure**

Besides simply representing the conceptual structure of the user’s dataset, DIVE’s graph-based data representation can effectively organize data. For example, using DIVE’s object model, we merged two ontologies from disparate sources. These two ontologies, represented as DIVE datanodes and dataedges, were merged through property inheritance. This allowed the second ontology to inherit

```sql
SELECT c1.step AS time_step,
    eucl_dist(c1.x, c1.y, c1.z, c2.x, c2.y, c2.z) AS distance
FROM coordinates AS c1
JOIN coordinates AS c2
ON (c1.step = c2.step AND c1.atom_id = c2.atom_id)
WHERE c1.step = (time_step)
    AND c1.atom_id IN (atom)
```

Figure 5. SQL streaming in DIVE. (a) Interactive SQL. On the left is an SQL template with tags for `time_step` and `atom`. This approach replaces the tags with input from GUI elements, and the final query calculates the distances between all user-selected atoms at the specified time. (b) Pass-through SQL. On the initial frame request, this approach constructs a datanode hierarchy around the .NET objects and then creates backing stores. On all subsequent frame requests, DIVE buffers SQL data directly into the backing stores using multiple threads. This approach then propagates large amounts of complex data through DIVE at interactive speeds by bypassing object-oriented parsing.
Visual Analytics for Biological Data

definitions from the first, resulting in a new ontology compatible with both data sources but amenable to new analysis approaches.

Besides these structural benefits, the datanodes are software objects that can update both their values and structures at runtime. Furthermore, the datanodes’ ontological context can also update at runtime. So, DIVE can explore dynamic data sources and handle the impromptu user interactions commonly required for visual analysis.

Object Parsing

As the case study showed, the ability to parse a .NET object or assembly distinct from the DIVE framework circumvents the need to add DIVE-specific code to existing programs. In addition, this lets us augment those programs with DIVE capabilities such as graphical interaction and manipulation. For the Dynameomics API, we integrated the underlying data structures and the streaming functionality into the Protein Dashboard without modifying the existing API code base. This lets us use the same code base in the DIVE framework and in SQL Common Language Runtime implementations and other non-DIVE utilities.

Streaming Structured Data

The most obvious benefit of DIVE is big-data accessibility through data streaming. Interactive SQL’s flexibility effectively provides a visualization front-end for the Dynameomics SQL warehouse. However, for datasets not immediately described by the underlying database schema or other data source, a more advanced method for streaming complex data structures is desirable.

We developed pass-through SQL to make hundreds of terabytes of structured data immediately accessible to users. These data are streamed into datanodes and can be accessed either directly or indirectly through the associated ontology (for example, through property inheritance). Furthermore, these data are preemptively loaded via back-
ground threads into backing stores; these backing stores are populated using efficient bulk transfer techniques and predictively cache data for user consumption. Finally, when the object parser is used with pass-through SQL, methods as well as data are parsed. So, the datanodes can access native .NET functionality in addition to the streaming data.

Preexisting programs also can benefit from DIVE’s streaming capabilities. For example, Chimera can open a network socket to DIVE’s streaming module. This lets Chimera stream MD data directly from the Dynameomics data warehouse.

Large-scale data analysis will remain a pillar of scientific investigation; the challenge facing investigators is how best to leverage modern computational power. DIVE and other VA tools are providing insights into this challenge. Although it’s unlikely that any general tool will ever supplant domain-specific tools, the concepts highlighted here—accessibility, extensibility, simplicity of representation, integration, and reusability—will remain important.

Acknowledgments
We performed the Dynameomics simulations using computer time through the US Department of Energy (DOE) Office of Biological and Environmental Research as provided by the National Energy Research Scientific Computing Center, which is supported by the DOE Office of Science under contract DE-AC02-05CH11231. We’re also grateful for support from Microsoft Research’s External Research Program (www.microsoft.com/science) (for Valerie Daggett), the US National Institutes of Health (grant GM50789 to Daggett), and the US National Library of Medicine (project 5T15LM007442, for Dennis Bromley and Steven Rysavy). Robert Su developed the Protein Dashboard.

References

Steven J. Rysavy is a PhD candidate in the University of Washington’s Biomedical and Health Informatics division. His research interests include protein structure prediction and visual analytics. Rysavy received an MS in computer science from San Francisco State University. Contact him at rysavy@uw.edu.

Dennis Bromley is a PhD candidate in the University of Washington’s Biomedical and Health Informatics division. His research interests include drug discovery and visual analytics. Bromley received a BA in computer science from Harvard University. Contact him at dbromley@uw.edu.

Valerie Daggett is a professor in the University of Washington’s Department of Bioengineering and in the university’s Biomedical and Health Informatics Program. Her research interests are protein dynamics, protein folding, and related diseases, as well as the development of tools to address these problems. Daggett received a PhD in pharmaceutical chemistry from the University of California, San Francisco and was a postdoctoral fellow at Stanford University with Michael Levitt. Contact her at daggett@uw.edu.