Developing Data-Intensive Applications in the Grid

Grid Forum Advanced Programming Models Group White Paper

Tahsin Kurc†, Michael Beynon†, Alan Sussman†, Joel Saltz†‡

† Dept. of Computer Science
University of Maryland
College Park, MD 20742
{kurc, beynon, als, saltz}@cs.umd.edu

‡ Dept. of Pathology
Johns Hopkins Medical Institutions
Baltimore, MD 21287

Abstract

This white paper reports on some of the issues in developing data-intensive applications in a Grid environment. In the context of this paper, data-intensive applications are those that explore, analyze, visualize, and otherwise manipulate very large scientific datasets. The paper presents several motivating applications from diverse scientific and engineering fields and a short survey of programming models/tools for developing data-intensive applications on the Grid.

1 Introduction

The primary objective of generating data through simulations or sensors is to better understand the causes and effects of physical phenomena. As a result, exploration, analysis and manipulation of large datasets play an important role in many disciplines of scientific research. Output data from simulations or sensor readings often consist of vast amounts of numerical values, making it difficult for scientists to reach useful conclusions and gain insight into their problems from the raw data. Understanding can be achieved by running analysis codes that will explore and extract useful information, and that will transform and reduce data into more suitable forms and manageable sizes that can be consumed by humans or by other programs. In many cases, analysis is a post-processing operation performed repeatedly on the stored data. Applications that query, analyze and manipulate very large datasets have emerged as one of the main consumers of the system resources. We refer to those applications as data-intensive applications.

Data-intensive applications are well understood when using tightly coupled parallel machines, or homogeneous clusters. With the current trend toward using heterogeneous collections of disparate machines collectively (the Grid) for a single application, the same techniques used in the tightly coupled, homogeneous cases are not sufficient. This paper reports on some of the issues involved in developing data-intensive applications in the Grid and presents some of the programming
models and tools for developing such applications. The organization of the paper is as follows. Section 2 presents several motivating applications. Characteristics common to the data-intensive applications and challenges in developing the data-intensive applications in the Grid are discussed in Section 3. Finally, we present a description of the Maryland DataCutter framework, and several other programming tools in Sections 4 and 5.

2 Motivating Data-intensive Applications

2.1 Satellite Data Processing

Earth scientists study the earth by processing remotely-sensed data continuously acquired from satellite-based sensors. A typical analysis [2, 9] processes satellite data for ten days to a year (for the AVHRR sensor, ten days of data is about 4 GB) and generates one or more composite images of the area under study. Generating a composite image requires projection of the region of interest onto a two-dimensional grid; each pixel in the composite image is computed by selecting the “best” sensor value that maps to the associated grid point. An earth scientist specifies the projection that best suits her needs. Sensor values are pre-processed to correct the effects of various distortions, such as instrument drift, atmospheric distortion and topographic effects, before they are used.

2.2 Virtual Microscope and Analysis of Microscopy Data

The Virtual Microscope [5, 16] provides a realistic digital emulation of a high power light microscope. The raw data for such a system can be captured by digitally scanning collections of full microscope slides under high power. The digitized images from a slide are effectively a three-dimensional dataset, since each slide can contain multiple focal planes. The size of a slide with a single focal plane can be up to several gigabytes, uncompressed. Hundreds of such digitized slides can be produced in a single day in a hospital. At a basic level, the Virtual Microscope can emulate the use of a physical microscope, including continuously moving the stage and changing magnification. The processing for the Virtual Microscope requires projecting high resolution data in the region of interest on the slide onto a grid of suitable resolution (governed by the desired magnification) and appropriately compositing the pixels mapping onto a single grid point, to avoid introducing spurious artifacts into the displayed image. Used in this manner, the Virtual Microscope can support completely digital dynamic telepathology [23]. In addition, it enables new modes of behavior that cannot be achieved with a physical microscope, such as simultaneous viewing and manipulation of a single slide by multiple users.

2.3 Coupling of Environmental Codes: Water Contamination Studies

Powerful simulation tools are crucial to understand and predict the transport and reaction of chemicals in bays and estuaries [19]. Such tools include a hydrodynamics simulator, such as ADCIRC or UBEST [10, 20], which simulates the flow of water in the domain of interest, and a chemical transport simulator, such as CE-QUAL-ICM [8], which simulates the reactions between chemicals in the bay and transport of these chemicals. For each simulated time step, each simulator generates a grid of data points to represent the current status of the simulated
region. For a complete simulation system, the hydrodynamics simulator needs to be coupled to the chemical transport simulator, since the latter uses the output of the former to simulate the transport of chemicals within the domain. As the chemical reactions have little effect on the circulation patterns, the fluid velocity data can be generated once and used for many contamination studies. The output data from a large grid at a single time step may be several megabytes, and thousands of time steps may need to be simulated for a particular scenario. The grids used by the chemical simulator are often different from the grids the hydrodynamic simulator employs, and the chemical simulator usually uses coarser time steps than the hydrodynamics simulator. Therefore, running a chemical transport simulation requires retrieving the hydrodynamics output in the region of interest (i.e., a region of the grid over a specified time period) from the appropriate hydrodynamics datasets stored in the database, averaging the hydrodynamics outputs over time, and projecting them into the grid used by the chemical transport simulator.

2.4 Visualization and Analysis of Astronomy Simulations

Simulations provide powerful tools for understanding and predicting the occurrence and effects of physical phenomena in space. For example, space weather simulations [14, 15, 21] model interactions between the solar wind and the Earth’s magnetosphere and ionosphere. The energy output of the Sun dominates conditions on and near the earth. Intermittent energy releases in the solar corona cause dramatic and harmful effects in the magnetosphere and upper atmosphere, which are collectively known as Space Weather. These phenomena may adversely affect power distribution networks and other electrical systems, the performance and reliability of earth orbiting spacecraft, and the safety of manned space flights. The space weather simulations require fast access to the solar wind data, access to a powerful computer to run the simulation codes, and fast, reliable access to the results for analysis. The magnetohydrodynamics equations, which model solar wind and magnetospheric plasma, are solved in a region of space containing the solar wind and the Earth’s magnetosphere. This region is essentially a large cylinder, 100 $R_E$ (radius of the earth) in radius and 380 $R_E$ long. Typical simulations use a grid with 50 points in the radial direction, 48 points in the poloidal angle, and 64 points in the azimuthal angle. The storage of the eight plasma parameters as double precision floats for a single timestep requires 10MB for this grid resolution. Simulations of higher resolution grids are possible on massively parallel architectures. The results of a simulation of a magnetic storm, lasting several days, can generate datasets up to a terabyte in size.

Visualization is the key to understanding the results of the simulation codes. A standard overview animation is produced to begin the analysis of the event, such as a magnetic storm. This visualization uses a cut plane through the simulation grid, depicting density and a translucent surface depicting the outer boundary of the magnetic barrier to the solar wind. Based on the standard animations, particularly interesting periods are selected for detailed analysis. For these relatively short periods, cut planes are moved and rotated to capture the plasma sheet, and individual field and flow lines are traced. These operations only need access to the data within the subset of the simulation spatial domain being viewed, but at a range of time steps. Initial visualizations can be done on a lower resolution grid to find interesting regions both in time and space quickly. The subsequent visualizations can increase the resolution for more accurate analysis of smaller regions in the dataset.
3 Developing Data-intensive Applications in the Grid

As is seen in the example applications in Section 2, applications usually access a subset of the entire dataset. As the data values in a scientific dataset are often associated with points in a multi-dimensional space, the subset can usually be defined compactly as a spatial range query, which is a multi-dimensional box in the underlying dataset space. Subsets can encompass contiguous regions of space, as for retrieving the satellite data covering a particular geographical region. Subsets can also be defined once features of interest are categorized using spatial indices. For instance, subsetting can be carried out to retrieve the simulation data associated with shocks in fluid simulations, or tissue regions with particular cell types in microscopy datasets.

There are various situations in which application-specific non-spatial subsetting and data aggregation can be applied to the targeted data subsets. Some data analysis requires values for only some of the variables at a data point. For example, an analysis code may only use the pressure value at a grid point in a hydrodynamics simulation, and may ignore the values for velocity and momentum. In other cases, there may be a need to obtain an application-dependent low resolution view of a dataset. For example, the analysis may need to be performed using coarser time steps, which requires the stored velocity values to be averaged over several time steps. In these cases, the size-reducing aggregation, subsetting and transformation operations could be applied to data elements at the data server where they are stored, before returning them to the client where the analysis program is run.

Now consider the following application scenario: A scientist wants to compare the properties of a 3D reconstructed view of a raw dataset recently generated at a collaborating institution, with the properties of a large collection of reference datasets. The 3D reconstruction operation involves retrieving portions of 2D slices from the regions in question, and then performing feature recognition and interpolating between the slices to extract the important 3D features. A description of these features and the associated properties are then compared against a database of known features, and some appropriate similarity measure is computed. The final result is the set of reference features that are close in some way to those found in the new raw dataset, along with the corresponding view renderings to visualize.

There are several challenging issues when the application is executed in a Grid environment:

- **Distributed resources.** The resources (new raw dataset, reference database, and the scientist) can all be at distributed locations in a wide-area network as seen in Figure 1. The reference database is likely to be stored in an image library, since the dataset is large and useful to many users. The new raw dataset is stored at the site where the sensor readings were taken.

- **Heterogeneous resources.** The characteristics, capacity and power of resources, including storage, computation, and network, vary widely. The application should be structured to accommodate the heterogeneous nature of the Grid. If the hosts containing the data are low-power archival systems that make the execution of the 3D reconstruction code prohibitively expensive, it becomes less clear how to structure the application for efficient execution. Ideally we would like to execute portions of the application at strategic points in the collection of machines.

- **Shared resources.** The distributed resources can be shared by many applications. Therefore, the application should be optimized in its use of the shared resources. In addition, the
application should be adaptive to the changes in the availability of the resources. For instance, it may not be efficient or feasible to perform all the processing at the data server, if the server becomes overloaded. In that case, the application should be able to move all or part of its computations to other machines.

A set of possible locations for performing the computations in the example application are indicated in the figure by question marks. For example, if the portion of code that performs the range select on the new raw dataset could be run on the host where the data lives, the amount of data to be transmitted over the wide-area network (WAN) would be reduced. The computation farm is an ideal location for the feature recognition and 3D reconstruction due to the parallelism inherent in the codes. Given the set of features that were identified, it would be efficient to perform the selection of similar features from the reference database on the data server where the database is located. The low end PC where the scientist is located can be used to collect the 3D rendering and the similar feature information for interactive presentation to the scientist.

The efficiency in the example scenario depends on the ability to embed computations in the streams of data from the data sources to the client, and to execute portions of application computations in a distributed fashion on machines that are best suited for the computation. Therefore, a programming model based on composing applications from a set of computational objects (program components or tasks) would be a viable approach for developing data-intensive applications in the Grid. In addition, a runtime system is needed to mask the many complexities in implementing the programming model. The frameworks and tools that are discussed in the following sections implement some form of this programming model.

4 DataCutter

DataCutter [6, 7, 11] is an application framework, under development at University of Maryland, that provides support for developing data-intensive applications that make use of scientific datasets
in remote/archival storage systems across a wide-area network. To make efficient use of distributed shared resources, the application processing structure is implemented as a set of distributed processes, called filters, that adhere to the filter-stream programming model. DataCutter uses these distributed processes to carry out a rich set of queries and application specific data transformations. Filters can execute anywhere (e.g., on computational farms), but are intended to be run on machines where the location provides an efficiency advantage. For example, given a filter that greatly reduces the size of the data it receives before sending it to the next filter, an efficient location would be close to the archival storage server. DataCutter also provides support for subsetting very large datasets through multi-dimensional range queries. It uses a multi-level hierarchical indexing scheme, based on R-tree indexing methods, to ensure scalability to very large datasets.

4.1 The Filter-Stream Programming Model

The basic ideas underlying the filter-stream model [7] are to (1) constrain application components to allow for location independence, which is necessary for execution in a distributed environment, and (2) expose application communication patterns and resource requirements, allowing a runtime system to aid in efficient execution. The programming model is loosely based on the stream-based programming model, originally developed for Active Disks [3, 22].

In the filter-stream programming model, part of an application is represented by a collection of filters. A filter is a portion of the full application that performs some discrete function. Filters can pre-disclose dynamic memory and scratch space needs so that the required space can be allocated by the underlying runtime system on behalf of the filter. Communication with other filters is solely through the use of streams. A stream is a communication abstraction that allows fixed sized untyped data buffers to be transported from one filter to another. A simple example of this model is Unix system pipes, where the standard output of a process is used as standard input for another process. Unix pipes represent a linear chain of filters, each of which have a single input stream and a single output stream. The filter-stream model allows for arbitrary graphs of filters with any number of input and output streams.

The process of manually restructuring an application using this model is referred to as decomposing the application. The main goal in choosing the appropriate decomposition is to achieve efficient use of limited resources in a distributed and heterogeneous environment. A particular granularity of the application decomposition into filters is not mandated by the model. Given a set of filters, the runtime mapping of filters onto various hosts in a wide-area grid environment is referred to as placement. The choice of placement represents the main degree of freedom in affecting application performance by, for instance, placing filters with affinity to data sources near the sources, minimizing communication volume on slow links, placing computationally intensive filters on less loaded hosts, etc. Note that a placement decision is not assumed to be static, and the programming model supports the notion of stopping a set of filters and replacing them with possibly a new set of filters with a different placement. A runtime system infrastructure, called the DataCutter Filtering Service, provides support for the execution of applications that are structured in the filter-stream programming model.

The lifetime of a filter is defined by the amount of work required by the application. Within this lifetime, a filter can process multiple logically distinct portions of the total workload. This is referred to as a unit-of-work, and provides an explicit time when adaptation decisions may be made.
while an application is running. A unit-of-work starts with the submission of a work description to a running set of filters, and ends when the last filter finishes processing the work. A collection of running filters that operate collectively to process a unit-of-work is referred to as a filter instance. An application may have multiple concurrent filter instances.

### 4.2 Filters

A filter is specified by the code to execute, and a description of the input and output streams it will use. Currently, filter code is expressed using a C++ language binding by sub-classing a provided filter base class. This base class provides a well-defined interface between the filter code and the runtime system. The description of input and output streams is specified in a separate configuration file.

The interface for filters consists of an initialization function, a processing function, and a finalization function. The `init()` function is called when the filter is instantiated, and is passed parameters with the command line arguments used when the application was started. This is where a filter would request scratch memory space for use during later processing, for example. The `process()` function is called to handle data arriving on the input streams in buffers from the sending filter. The parameter passed to the process function contains arrays of descriptors for the sets of input streams and output streams this filter can use. The filter can only read and write from/to the provided streams. No new streams can be created by the filter at runtime. The `finalize()` function is called after all processing is finished and the filter is ready to terminate. This is where a filter would release any internal resources in use.

```cpp
class MyFilter : public AS_Filter_Base {
public:
  int init(int argc, char *argv[]) { ... };
  int process(stream_t *st) { ... };
  int finalize(void) { ... };
}
```

A filter cannot change the source of its input streams nor the sinks of its output streams. This has two advantages. First, a filter does not need to handle buffering and scheduling for its own communication, thereby reducing the complexity of filters. Second, the location of filters is transparent, allowing filters to be placed at different locations initially and relocated as system resource constraints change.

Filters are the unit of placement. Each filter can potentially be executed on a different host. In addition, a filter’s location may change at discrete application-defined intervals during the course of execution. Note this does not imply true migration of code and state, but rather placement can be recomputed and the filter can be stopped on the original host and a new copy re-instantiated on the new host. There is a limited mechanism for a final state transfer by a single buffer transfer from the old instance to the new instance. This approach avoids many of the details involved in check-pointing and process migration, while retaining most of the benefits. Filters need to be structured appropriately to handle such events. For cases when this is not desirable, a filter can be pinned to a particular host, which means the filter will always be placed on that host. This host affinity limits placement flexibility, but is useful for some situations, such as when runtime libraries, auxiliary data files, or other required resources exist only on a particular host.
Filters may choose to pre-disclose the amount of memory and local disk space required as scratch space for the filtering operation. If a filter chooses to do so, minimum values for both the memory and the disk scratch space are specified by the filter in the init() method, using the scratch space class provided by the DataCutter infrastructure. The granted scratch space is allocated on behalf of the filter by the filtering service. The filter can make use of the granted scratch space in the process() method as needed via the methods in the scratch space class. If the minimum requested space cannot be allocated, the runtime system signals the application and exits all of the filters. The memory scratch space is accessed using the methods with the same semantics as malloc/free. The disk scratch space is accessed by a file stream (see the next section for a description of file streams).

One of the potential benefits of exposing resource requirements in this way is that the runtime system can achieve a better placement of filters. For example, a filter can be run on a machine with enough memory to avoid paging, and two filters requesting large scratch spaces can be placed on two different machines. In addition, the runtime system can potentially use this information to perform better scheduling of co-located filters on a machine.

4.3 Streams

A stream is an abstraction used for filter communication. Since the placement of filters is largely unknown until runtime, this mechanism is used to achieve location-independent filter code because stream names are used rather than endpoint locations on specific hosts. The name of a stream is unique across all the streams used in an application or in a set of related applications. A stream is optionally associated with an XML 1 DTD (Document Type Descriptor) [24]. The XML DTD is a schema to define the data layout in a buffer in the stream.

For each filter, a list of input and output streams is required. This discloses all potential filter communication pairs for the entire execution of the application. Given a set of filters with stream connectivity information, a task graph can be built where the nodes represent the filters, and the edges represent stream connections.

There are two kinds of streams in DataCutter: file streams and pipe (or inter-filter) streams. File streams are used to access files. A pipe stream is the means of uni-directional data flow between two filters, from the upstream filter to the downstream filter. Bi-directional data exchange can be achieved by creating two pipe streams in opposite directions between two filters. The current prototype implementation uses TCP for stream communication, but any point-to-point communication library could be employed, such as Nexus [17].

A pipe stream is used to specify how filters are logically connected, and to provide the glue at runtime to attach an input stream for one filter to an output stream of another. All transfers to and from streams are through a fixed size buffer abstraction. A buffer represents a contiguous memory region. The size of the actual data in a buffer can be smaller than the maximum size of the buffer. Thus, the buffer descriptor contains a pointer to its start, the length of the portion containing useful data, and the maximum size of the buffer. A filter has to disclose a minimum value and an optional maximum buffer size value for each of its input and output streams in the init() method, when it is instantiated. The maximum value is a hint to the filtering service and represents the optimal buffer size desired by the filter. If there are multiple filters running on the same machine, the

---

1 XML stands for the Extensible Markup Language, a World Wide Web Consortium standard for encoding data.
filtering service may use the maximum value to optimize the buffer space that can be allocated to all filters. The filtering service checks if the required minimum buffer space can be allocated at both ends of the stream. If the required space cannot be allocated, the filtering service signals the application with an error code and exits the application’s filters.

File streams are used to access files. The file stream abstraction further insulates the filter code from specifics about the host system. This provides a measure of safety between co-located filters, since one filter cannot access another filter’s scratch disk space. The permanent disk storage presents a uniform file system to all filters, similar to a traditional file system. Thus a filter with sufficient authorization can read files in permanent disk storage written by another filter.

The names of the files to be accessed can be defined by the user, but the files are opened or created by the filtering service. The names of the files are declared in a stream configuration file. The stream configuration file for an application defines how filters are connected via streams and what are the end points of streams— an end point may be a user-defined file.

To provide support for applications that access files through range queries, access to a file is done through segments— in those applications dataset files are partitioned into segments, and each segment is associated with a bounding box in the underlying multi-dimensional space. A segment may contain meta-data (e.g., a bounding box if the segment is returned as a result of a range query), and may cover the entire file. A filter views a segment as a contiguous chunk of data, yet the segment may actually be stored differently in the storage system. The size of a segment may be greater than the size of available memory, hence segments are accessed in fixed size buffers as for pipe streams. The size of the buffer for a file stream is determined when the filter is instantiated as is done for pipe streams— the size of the buffer is the same for all segments accessed by the file stream. We can think of a file as a directory in Unix, and segments as files in this directory. Therefore, the file stream class contains methods to open, create, close and destroy segments in a file.

4.4 The Runtime Environment

The execution service performs all the steps necessary to instantiate filters on particular machines, connect all the logical stream endpoints, and call the interface functions to allow filters to run. The description of where to instantiate filters is provided by a placement specification. This can be statically defined by the application, or can be determined automatically by the filtering service. An application filter binary must exist on every host, and it must contain at least the code for the filters that will execute on that host. For convenience, the binary can contain code for all filters, and those filters not intended to run on a given host will not be instantiated at runtime. This application binary is linked with the filter runtime system. In the current prototype, the filter service runtime system creates one POSIX thread for each new application filter object it creates on a host. The thread calls the init() method function passing the command line arguments. Next, the thread calls the process() method. When this returns, all streams are closed and the finalize() method is called.

Control starts in the application’s main() function as in any C/C++ program. This initial execution is called the Console Process, which makes use of the DataCutter filtering service API to instantiate filter instances, create and send unit-of-work descriptions to the filter instances, and to optionally collect the results from the work. The overall design and runtime handling of
filter instance creation and shutdown is asynchronous, and avoids blocking and barriers whenever possible. The handling of application buffers between filters is optimized to allow the pipelining, by overlapping read/write operations to streams with filter computation of later buffers. Overall, the runtime environment is optimized to make filter instances as inexpensive as possible, allowing for greater benefits from adaptation policies.

5 Other Programming Tools

In this section we present several other programming tools that can be used for developing data-intensive applications in a Grid environment.

5.1 ABACUS

The ABACUS framework [1], under development at Carnegie Mellon University, addresses the automatic and dynamic placement of functions in data-intensive applications between clients and storage servers. The framework consists of an object-based programming model for developing applications and a runtime system to instantiate, monitor and dynamically migrate objects between clients and storage servers.

5.1.1 The Programming Model

In ABACUS, a data-intensive application is implemented by composing together functionally independent, communicating objects. An object can be either a mobile object or a stationary (non-migratable) object. Clients and storage servers are the two kinds of nodes in ABACUS, where the objects can be executed. Stationary objects are pinned to the client or the storage server. A storage server provides storage objects, which are pinned to the respective storage server and enable access to the persistent storage through a flat file interface. A console object is pinned to the client node where the application is started, and encompasses the part of the application that interfaces with the user or the rest of the application not implemented in ABACUS. Application processing is initiated at the console and invocations for object instantiations and method invocations are propagated to the rest of the system and the other objects by the ABACUS runtime. Applications also can declare other objects as stationary. The mobile part of the application program lies between stationary objects placed at the storage servers and the console node. Data flows from the storage objects to the console object through the mobile objects. A mobile object can be dynamically placed at the storage server or at the client by the runtime system during the execution.

An object is explicitly declared as a mobile object by the application developer or the user. A mobile object may contain a state, thus is required to implement special methods so the runtime system can migrate the object and instantiate it somewhere else without losing the state of the object before the migration. The private state of the object is internal to the object and can only be accessed by other objects or the ABACUS runtime system through the exported object interface. The private state can contain internal objects and references to external objects. It is the responsibility of the object to save its internal state and the state of embedded objects before it is migrated. For this, each mobile object implements a CheckPoint() method, which saves the internal state of the object and the embedded objects into a memory buffer or to a file, and a Restore() method,
which restores the state of the object. The runtime system calls the `Checkpoint()` method of the object before the migration, transfers the contents of the memory buffer or the file to the location where the object is migrated, and calls the `Restore()` method of the object after the migration. An object performs processing when one of its methods is invoked, and interacts with and moves data to other downstream objects by invoking their methods. Data from an upstream object to a downstream object is moved in application-specific chunks, rather than the entire dataset, at each invocation of the methods of the downstream object.

Mobile objects of the same type (e.g., cache objects that implement cache for a filesystem) are grouped together and managed by mobile object managers. Object managers are required to implement a `CreateObj()` method, which is used by the ABACUS runtime to instantiate an object.

### 5.1.2 The Runtime System

The ABACUS runtime system is responsible for monitoring resource consumption of objects, the status of the system, and for instantiating and migrating mobile objects dynamically. It consists of binding managers and resource managers in each node in the system. Binding managers, as in most distributed object systems, are responsible for instantiation and locating of mobile objects, and for providing support for location transparent invocation of their methods. Resource managers are responsible for monitoring resources, collecting statistics about the resource usage of objects and about the availability of resources, and carrying out placement decisions.

Applications instantiate objects by making requests to the ABACUS runtime system through the console object. The runtime system creates an object in memory by calling the object manager for that type of object. The object manager creates the object and returns a `manager reference` for the object to the runtime system. The runtime system assigns a system-wide unique `runtime identifier` to the object, which provides a means for indirection and is used by the other objects to invoke the object without knowing its location. In each node, a hash-table, called `location table`, is maintained that maps each runtime identifier to a `(node, object manager, manager reference)` tuple. When objects are moved back and forth between the server nodes and the client node, the source node, the target node and the home node, where the object is instantiated initially, are updated to reflect the new location of the object.

In the current prototype, objects are implemented as C++ classes and are statically linked to ABACUS system on server and client nodes. The method calls to remote objects use the DCE RPC mechanism of synchronous blocking calls. Object migration is carried out by moving the application-defined state buffer and instantiating the object on the new node, rather than moving the object codes.

### 5.2 The dQUOB and ACDS Systems

The ACDS [18] (Adapting Computational Data Streams) from Georgia Institute of Technology is a framework that addresses construction and adaptation of computational data streams in an application. Computational data streams model application processing as computational objects associated with data streams. A computational object performs filtering and data manipulation. Data streams characterize data flow from data servers or from running simulations to the clients
of the application. The runtime system of the ACDS performs migration, splitting and merging of streams to adapt for runtime variations such as data volume and availability of resources.

The implementation of the ACDS stream components uses two C++ classes. The first is the *basic stream component* class, which provides communication support through its *channel event interface* class. The second is the *adaptable stream component* class, which provides necessary components to carry out adaptations. To ensure correctness when streams are migrated from one node to another node in the system, computational stream components implement functions that save and restore their internal state.

The dQUOB [12] (*dynamic QUery OBjects*) system, also from Georgia Institute of Technology, consists of a runtime and compiler environment that allows an application to insert computational entities, called *quoblets* into data streams. The data streams targeted by the dQUOB system are those in large-scale visualizations, in video streaming to a large number of end users, and in large volumes of business transactions. The dQUOB system uses a SQL-like declarative language, which defines a *rule-action* style programming model. Rules are the queries into the dataset, while actions are the user-defined functions, called quoblets, embedded as part of the query. In effect, the processing required by an application is represented as a query with user-defined functions. The dQUOB compiler transforms a given query expressed in this language into a Tcl script. The resulting script contains references to the user-defined functions and can be executed on the streams flowing from the data servers (sources) to the clients to carry out application-specific transformations and filtering.

5.3 Dv

Dv [4, 13] is a framework for developing applications for distributed visualization of large scientific datasets on a computational grid. It is under development at Carnegie Mellon University. The Dv framework is based on the notion of *active frames*. An active frame is an application-level mobile object, which contains application data, called *frame data* and a program, called *frame program* that processes the data. An active frame program implements a single \( \text{run()} \) method, which computes an output frame from an input frame data, and returns the network location of the destination node for the output frame.

Active frames are executed by *active frame servers* running on the machines at the client and remote sites. An active server consists of an interpreter that executes frame programs, and an optional collection of application specific libraries that can be called from within a frame program. At runtime, a server listens to a well-defined port and receives an active frame from the network. The server then extracts the frame data and the frame program, and executes the \( \text{run()} \) method on the input frame data. After executing the \( \text{run()} \) method, the output frame data and the output frame program are packed and sent to the destination node by the active frame server.

The overall application consists of a *request server*, a *local Dv client*, and a set of *active frame servers*. The request server receives frame requests (e.g., region of interest in the dataset, viewing parameters, etc.) and accesses the datasets. The local Dv client also is an active frame server, but it runs additional code to interface with the user. The data and programs flow from the request server to the local Dv client through the intermediate active frame servers, which can be located at the remote sites or the local site.
References


