Scalable Distributed Execution Environment for Large Data Visualization

Micah Beck, Huadong Liu, Jian Huang and Terry Moore

University of Tennessee
Deptment of Computer Science
Knoxville, TN 37996-3450, USA
{mbeck, hliu, huangj, tmoore}@cs.utk.edu

Abstract

To use heterogeneous and geographically distributed resources as a platform for parallel visualization is an intriguing topic of research. This is because of the immense potential impact of the work, and also because of its use of a full range of challenging technologies. In this work, we designed an execution environment for visualization of massive scientific datasets, using network functional units (NFU) for processing power, logistical networking for storage management and visualization cookbook library (vcblib) for visualization operations. This environment is based solely on computers distributed across the Internet that are owned and operated by independent institutions, while being openly shared for free. Those Internet computers are inherently of heterogeneous hardware configuration and running a variety of operating systems. Using 100 such processors, we have been able to obtain the same level of performance offered by a 64-node cluster of 2.2 GHz P4 processors, while processing a 75GBs subset of a cutting-edge simulation dataset. Due to its inherently shared nature, this execution environment for data-intensive visualization could provide a viable means of collaboration among geographically separated users.

1. Introduction

The idea to use distributed and heterogeneous resources for large-scale data intensive applications came with great promise. Unfortunately, when subjected to real-world applications, most existing technologies in this venue fall short in a number of ways. It is now a general belief that a sufficient solution can only come from close collaboration by experts from a range of fields, such as distributed computing, networking, and storage as well as domain experts.

In this work, we use large-scale visualization as the driving application to motivate and test the development of several new technologies. Particularly we focus on devising a network functional unit (NFU) to provide a coarse-grained execution primitive for distributed computing, and to explore logistical networking as a storage fabric. A new visualization library (visualization cookbook library, vcblib) has also been developed as a portable primitive for execution on NFU. Although the scheduler that orchestrates parallel computing using vcblib operations on NFU and logistical networking storage was developed specifically for large-data visualization, its design and application are general enough for direct extension to other application domains.

The real-world potential impact of the execution environment is the following. Over the years, parallel visualization has become a crucial tool to understand cutting-edge simulation datasets. Current popular platforms of parallel visualization used by computational [1-2] researchers are primarily computer-clusters of various sizes. Our newly developed execution environment for large data visualization is one that is solely based on independent computers connected by the commodity Internet. The computers are distributed and shared, administered by different organizations and heterogeneous in nature. The following features are met:

First, the current standard functionalities of large-data visualization are factored into a small number of standard operations. An application scientist can program a parallel visualization run by simply editing an XML file through a text processing program, with issues like parallel efficiency, results aggregation and fault-tolerance in the wide-area transparently taken care of.

Second, with the underlying infrastructure shared in nature, a parallel visualization session can be launched on-demand, without requiring each job to go through batch queues and wait an undetermined amount of time before being launched.

Third, the overall shared infrastructure offers a performance on par with medium-sized clusters. Using 100 distributed processors, on some of which we were only allowed to use roughly 10% of the CPU resource, we have obtained a performance that roughly measures up to a 64-node 2.4 GHz cluster.

Finally, as a potential use of this distributed execution environment, application scientists can use the system as a large-scale cluster virtually replicated on all local sites. With this support, collaborating could be as straightforward as sending each other pointers to the data and a textual description of the visualization run (both in XML files).

The remainder of the paper discusses each of the three components after a general summary of the related background of the subject.

1-4244-0910-1/07/$20.00 ©2007 IEEE.

2. Background

The persistent mood of exhilaration in the research community over exponential increases in the capacity of computational resources has been tempered recently by the realization that a torrential influx of data from instruments, sensors and simulations is growing even faster than the resources needed to analyze it. The impact of this "data deluge," challenging enough by itself, is exacerbated by the fact that many data intensive projects today involve teams of collaborators spread out across geographically and organizationally distinct sites. Under these conditions, the escalation in data volumes, from giga, to tera, to peta-scale, raises daunting logistical problems for data management, both in relation to the people who want to work with the data and in relation to the resources available to them on any particular occasion. A system that addressed these conditions would have to enable a community of collaborators, distributed throughout the wide area, to get responsive answers to dynamic queries, analyses and visualizations applied to terascale or larger data sets.

Logistical Networking (LN) is an effort to integrate shared storage and processor resources into the communication fabric in order to address the mounting problems associated with managing massive data in application areas like content distribution and distributed data intensive scientific applications. The initial focus of LN research was on storage, and the goal was to make transmission and storage resources coordinate elements in a unified infrastructure. The *Internet Backplane Protocol* (*IBP*) was created to achieve this goal. IBP is a generic, best effort storage service that was designed by analogy with IP in the hope of producing a common storage service with similar characteristics, especially in regard to deployment scalability. We defined a "storage stack", analogous to the Internet stack, using a bottom-up and layered design approach that attempts to adhere to the end-to-end principle. IBP is the lowest layer of the storage stack that is globally accessible from the network. Just as IP is a more abstract service based on link-layer datagram delivery, so IBP is a more abstract service based on blocks of data (on disk, memory, tape or other media) that are managed as "byte arrays." By masking the details of the storage at the local level — fixed block size, differing failure modes, local addressing schemes — this byte array abstraction allows a uniform IBP model to be applied to storage resources generally. The use of IP networking to access IBP storage resources creates a globally accessible storage service.

As the case of IP suggests, in order to scale globally the service guarantees that IBP offers must also be weakened. First and foremost, this means that, by default, IBP storage allocations are time limited. When the lease on an IBP allocation expires, the storage resource can be reused and all data structures associated with it can be deleted. Forcing time limits puts transience into storage allocation, giving it some of the fluidity of datagram delivery and making it far more sharable and easier to scale. Additionally an IBP allocation can be refused by a storage resource in response to over-allocation, much as routers can drop packets; such "admission decisions" can be based on both size and duration. IBP allocation semantics also assume that a storage resource can become transiently unavailable, or even be permanently lost. In all cases the weak semantics of this "best effort" storage service mean that the level of service must be characterized statistically.

According to prevalent expectations in the community today, large-scale visualization should handle data sets at least on the order of tens of gigabytes. The sheer size of the data sets makes it indispensable to use parallel platforms. Previous work on parallel rendering algorithms, data management and parallel I/O abound in the literature. Very commonly, large-scale visualization is computed using large-scale parallel clusters of computers.

To date, few existing systems use remote processors or even un-orchestrated local resources for data intensive applications on the scale of large data visualization. To explore the possibility of using such an infrastructure for visualization, we chose to focus on isosurface extraction and volume rendering, which are the two predominant algorithms for volume visualization. The complexity of existing visualization packages, such as Vtk and Paraview, made them extremely challenging to deploy on distributed processors providing a weakened semantic, and require implementation of the end-to-end principles. In particular, one should note that we could not assume administrator privileges on any of those remote processors, making installed any of those existing visualization packages burdensome and oftentimes unrealistic. Hence we had to implement a selected set of standard visualization operations in a compact and self-containing library. This library is called the Visualization CookBook Library (vcblib). More information about vcblib is available at `http://www.cs.utk.edu/~seelab/vcb.php`.

3. Visualization Operations (vcblib)

The current vcblib implementation assumes that all data sets are on a regular grid. Curvilinear grid, irregular grids or unstructured data are not yet supported. It also assumes that the entire data set, no matter the dimensionality (i.e. 2, 3, 4 or more dimensions), is stored in a serialized continuous memory segment.

vcblib implements the fundamental visualization algorithms for isosurfacing and volume rendering. A primary focus in vcblib is to identify a narrow and well-defined API for each visualization function that is involved. Our design goal was to have visualization operations using similar types of semantics as in the standard C library.

In addition, we provide basic operations that can be used to compose higher-level visualization operations. For instance, a user may have a need to extract isosurfaces from only one octant of the entire domain of a volume. It is also

plausible that when handling a time-varying data set, a user may wish to slice the 4 dimensional space in the direction of the Z axis and extract isosurfaces in the 3D domain formed by the X, Y and time axes. To meet this need, vcblib needs to provide some necessary utility operations. From our research, we discovered that a core utility function that can be used in many visualization algorithms is a function to "grab" a block from within a volume. This is very useful for all block-based processing, such as dimension reduction, out-of-core processing and parallel rendering, etc. The API for this basic "grab" function is vcbGrablk.

As a fundamental function in vcblib. vcGrablk grabs a block of data from a volume. Basic descriptions of the volume include the number of bytes for each variable, the number of variables on each voxel, the number of dimensions and the size of each dimension. The target block to grab is specified by the lower and upper bounds of the block.

Besides grabbing a sub block from a volume, one can also use vcbGrablk to take arbitrary axis-aligned slices in high dimensional volumes. For instance, still using the example above that slices a 4D data in the direction of the Z axis, one can just set the lower and upper bounds in the Z direction to the Z value needed and full ranges for all other directions. Then the result is a 3D volume, stored in the default voxel order.

vcbGrablk allows a user to pick and choose which axis-aligned portion of the volume to focus on when another vcblib function is invoked. vcbGrablk has been proven to be simple to use, and very dexterous for different applications.

For scalar volume visualization, a user would either perform isosurface extraction (vcbMcube) or software volume rendering (vcbRaycast). For both, classic algorithms exist. We implemented the marching cube [9] algorithm for isosurface extraction. The APIs of the two functions are just the exact kind of a regular C function. They are invoked as part of a dynamic library on the remote processors.

The vcblib implementation is self-contained without external dependencies. The source code is entirely in C and is highly portable across platforms. Right now, it runs on all wide spread operating systems including Linux, various flavors of Unix, Windows and Mac OS X.

4. Distributed Storage (IBP/LoRS)

The server nodes of the infrastructure on which vcblib must run are managed by the Internet Backplane Protocol (IBP) [12]. IBP implements a generic, best effort network storage service that can scale globally. Two key characteristics of IBP storage are:

1. Allocations of IBP storage are limited in size and duration. An IBP allocation request can be refused in response to over-allocation and the storage resource can be revoked when the lease expires. Also, an IBP server may be restricted to use only idle disk resources ("soft" storage), ensuring that the host machine does not over commit resources.
2. Semantics of IBP storage are weaker than the typical storage service. IBP storage resource can be transiently unavailable or even permanently lost. With "soft" storage allocation semantics, resource can be revoked at any time before expiration.

IBP storage is managed by servers called "depots", on which clients perform remote storage operations. The depot was designed for simplicity and robustness by using a stateless protocol. IBP clients view a depot's storage resources as a collection of byte arrays. Clients initially obtain the use of a byte array by making a storage allocation on a depot. If the allocation is successful (depending on size and duration requested as well as the storage resources available), the depot returns three cryptographically secure URLs, called capabilities, to the client: one for reading, one for writing, and one for management. Capabilities may be passed from client to client, requiring no notification to the depot. The synchronous (blocking) IBP client calls fall into three different groups as shown in Table 1. A corresponding asynchronous (non-blocking) client API is also available.

Because of the limitations on allocation size, duration and semantics, IBP does not directly implement strong storage services such as conventional files. Instead, these services must be built on top of IBP. For example, the XML encoded exNode was created to aggregate and manage primitive IBP byte arrays. Basic middleware tools for using this network storage infrastructure have already been developed and are available at `http://loci.cs.utk.edu`. The Logistical Runtime System (LoRS) consists of a set of tools and associated APIs that allows users to draw on a pool of depots in order to enable the implementation of files and other storage abstractions with a wide range of characteristics, such as large size (through fragmentation), fast access (through caching), and reliability (through replication). LoRS tools also implement some transport layer services such as fault tolerance, encryption, and compression, at the end-points.

This basic description of IBP summarizes its storage management capabilities, which our distributed visualization service uses for state management. To supply the necessary processor resources, we extended IBP with a primitive yet powerful model of computation. After presenting the underlying design of vcblib in the next section, we describe this computational extension to IBP, called the Network Functional Unit, in Section 5.

Depot Management	`IBP_status`
Storage Management	`IBP_manage`
Data Transfer	`IBP_store IBP_load IBP_(m)copy`

Table 1. Synchronous IBP client API

5. Distributed Processors (NFU)

Besides vcblib and IBP storage depots, we also need a viable and scalable support of distributed computing on un-orchestrated distributed systems. It is general to treat computation as transforming data stored in byte arrays. Since we already have IBP to manage byte array storage, we have designed an extension to IBP to provide data transformation services. This extension is called Network Functional Unit (NFU), which is generic and best effort. The design of NFU adheres to the end-to-end principles [13-14] so that it can be shared across the network. Without the NFU mechanism, it would be extremely difficult to deploy the vcblib over several hundred of processors in the wide area network.

The NFU is an abstract service based on managing the underlying computational capabilities of the depot as "operations". Similar to IBP storage allocations, NFU operations are by default limited in size and duration. This means that there is a bound on the size of byte arrays that are arguments to any computation and a bound on the duration of execution. Semantics of NFU operations are weaker than typical process creation or procedure call on a local processor, as is necessary in order to model computations accessed across the network. Because of the restricted and weak semantics of NFU operations, abstractions with strong properties, such as reliability, unbounded size, and unbounded duration, must be constructed at a higher layer that aggregates primitive NFU operations below it. This is currently implemented through the use of scheduling for performance and fault tolerance algorithms implemented as part of the code that invokes NFU operations (e.g. vcblib).

5.1 NFU Operations

NFU operations are usually grouped as libraries so that they can be managed hierarchically. Libraries of NFU operations are classified as either *static* or *dynamic* as shown in Figure 2. Static NFU operations are built-in modules that are highly standard and useful to many applications in general. For instance, we have already deployed an NFU library of optimized BLAS operations as a static library. Static libraries are deployed by being compiled and linked as part of an IBP depot, and so require no further deployment action to be usable by that depot's clients. The processing logic of a static operation is defined by the operation provider and accepted/verified by the depot owner before deployment.

In contrast, dynamic NFU operations are implemented by code that is executed or interpreted by a particularly general static NFU operation. The code that defines a dynamic NFU operation is stored in an IBP allocation and passed to the appropriate static operation as an argument. Because of the dynamic nature of the operation, the code is not in general known to the depot owner before it is invoked, and may be delivered from an arbitrary client across the network. Thus the code that

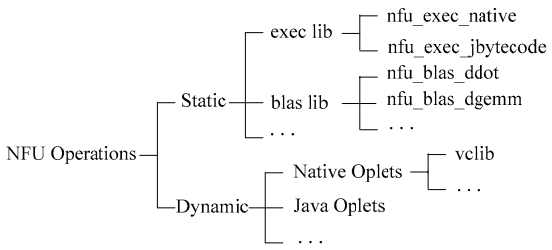

Figure 2. A taxonomy of NFU operations

defines a dynamic NFU operation is a kind of mobile code, which we refer to as an "oplet".

A static NFU library that loads and executes oplets from an IBP allocation, acting on arguments stored in other IBP allocations defines the NFU exec library. There are two generic static NFU operations to load and run oplets stored as machine dependent native code (native oplets, `nfu_exec_native`) and oplets stored as machine independent Java bytecode (Java oplets, `nfu_exec_jbytecode`). As we will discuss, the requirement of weak semantics means that the actions allowed by the execution environments must be restricted.

Deploying or updating a static NFU library is a manual administrative procedure that involves verifying the library and installing the executable image to a depot directory. The depot process will automatically load that library at runtime. Note that this is a heavyweight process since a level of trust must be established between the library provider and the depot owner. Further, each installation of an additional library takes some resources on the depot, whether a member operation is invoked or not, potentially diminishing the space available for other purposes such as caching, and complicating the management of the depot. Moreover, because static operations execute with the full privileges of the depot, the greater the number of statically installed operations the higher is the risk of depot misbehavior due to design or programming errors. For this reason the number of static operations deployed on a depot should be minimized. Scalability can be improved by constructing a minimally necessary set of operations comprised of generic functions that are basic and thus more likely to be reliable. It is up to the client to compose those base operations to implement all needed computations.

Right now, we are deploying vcblib as a dynamic library, i.e. NFU native oplets. As the process of production visualization gets further standardized and accepted by the field, we would like to eventually make vcblib part of the static library. This potential is facilitated by the low software complexity of the vcblib library. For instance, the entire binary vcblib library for a typical linux system only requires ~ 440 KB storage.

5.2 Resource Discovery

Partitions of the shared data set can be simply uploaded and replicated to available IBP depots using the LoRS tools that abstract and hide the underlying platform dependent details. However, the vcblib needs to be distributed to the set of IBP depots where it can be loaded to execution. Once we have the exNode representation of the data set, we make an `IBP_status` query to every depot pointed to by the exNode to get platform information of the depot, and also to check whether `nfu_exec_native` is installed. The client then constructs a list of matched depots for every platform, against which the vcblib has been cross-compiled, and uses `lors_upload` to distribute vcblib -- one complete copy per depot. This procedure is similar to how a RPM package is cross-compiled and downloaded to a target host.

6. Results and Discussion

In a distributed environment, fault-tolerance is crucial, and, hence, redundancy must be inherently incorporated. To do so, we employ k-way replication scheme [11], with k being a small number, e.g. 3. Supposing there are 100 depots in total, in this way, each data partition will be available on k randomly selected depots, and each depot will hold k% of the entire dataset. Each depots only processes job partitions that it has data for. This data distribution mechanism is built on top of IBP.

Then, the operations supported by vcblib on NFUs provide the visualization functions. An NFU-enabled depot accepts a visualization request (i.e. a vizSpec file and an exNode file, both in XML) across the Internet, performs the operation and returns the results to the requester.

With a means to process one single partition of the dataset, it is then the job of a scheduler, run on the client machine, to orchestrate a parallel run with parallel scalability and fault-tolerance. Fortunately, for many visualization applications, embarrassingly parallelism is often sufficient. In that scenario, the scheduler is rather straightforward to implement. In fact, in all of our experiments the schedulers have always been written in Unix Shell scripts. Details of our scheduler implementations have been published in [15].

Right now, we do not have any distributed resource provisioned specifically for distributed data-intensive visualization. However, just by using PlanetLab [3] resources heavily shared by a large community for diverse applications, we have achieved rather significant results.

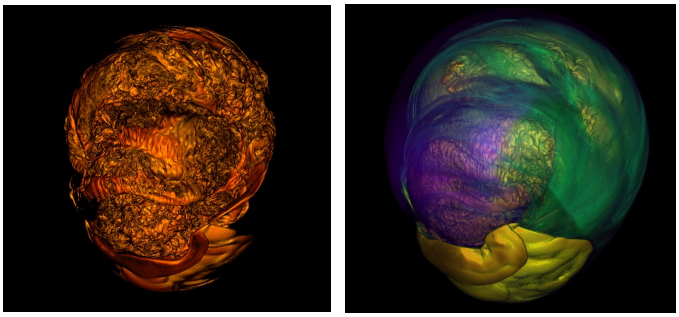

Figure 3. Sample images of the TSI data set, rendered with two different transfer functions.

For instance, with the latest TSI (Terascale Supernova Initiative) simulation dataset at $864 \times 864 \times 864$ spatial resolution, the time it takes to render a 30 time-step subset of the TSI dataset on 100 depots is about 240 seconds on average and 480 seconds in the worst-case, at a step size of 0.5 and an image resolution of 800 by 800. The same rendering takes 219 minutes on a dedicated 2.2 GHz P4 CPU. In other words, this performance is equivalent to that offered by a 32 to 64-node cluster of 2.2 GHz P4 processors, assuming a 90% parallel utilization (Figure 3.)

Most of the 100 NFU-enabled depots are hosted on PlanetLab nodes. PlanetLab nodes are server-class machines meeting a periodically revised hardware requirement. They are virtualized as "slices" and shared among a large user community [3]. In order to enable large scale sharing of PlanetLab nodes, each slice has limited storage and CPU share. Our tests were run without reserving CPU share through the PlanetLab Sirius Calendar Service, and thus there is no lower bound on the cycles available to us on any PlanetLab nodes in any runs.

Acknowledgement

This work was supported in part by NSF grant CNS-0437508. The authors thank the TSI project for motivating our research and providing our main test datasets. We also thank Larry Peterson for PlanetLab access.

References

[1] D. Keyes, "SCaLeS: Science Case for Large-scale Simulation," Office of Science, DOE June 2003.

[2] "Scientific Discovery through Advanced Scientific Computing," Office of Science, U. S. Department of Energy, Washington, D.C. March 24 2000.

[3] "http://www.planet-lab.org/."

[4] *The Visualization Toolkit User's Guide*, 4.2 ed: Kitware Inc., 2003.

[5] "Vis5d+. http://vis5d.sourceforge.net/."

[6] W. Bethel and J. Shalf, "Cactus and Visapult: An Ultra-High Performance Grid-Distributed Visualization Architecture Using Connectionless Protocols," *IEEE Computer Graphics and Applications*, 2003.

[7] VisIt, "http://www.llnl.gov/visit/home.html."

[8] A. Henderson, *The ParaView Guide: A Parallel Visualization Application*: Kitware Inc., 2004.

[9] W. Lorensen and H. Cline, "Marching Cubes: a high resolution 3D surface construction algorithm," presented at Proc. of SIGGRAPH, 1987.
[10] M. Meissner, J. Huang, D. Bartz, K. Mueller, and R. Crawfis, "A Practical Evaluation of the Four Most Popular Volume Rendering Algorithms," presented at IEEE/ACM Symposium on Volume Visualization, Salt Lake City, Utah, 2000.
[11] R. Samanta, T. Funkhouser, and K. Li, "Parallel Rendering with K-Way Replication," presented at IEEE Symposium on Parallel and Large-Data Visualization and Graphics, 2001.
[12] M. Beck, T. Moore, and J. S. Plank, "An End-to-end Approach to Globally Scalable Network Storage," presented at ACM Sigcomm 2002, Pittsburgh, PA, 2002.
[13] M. Beck, T. Moore, and J. S. Plank, "An End-to-End Approach to Globally Scalable Programmable Networking," presented at Future Directions in Network Architecture (FDNA-03), an ACM SIGCOMM 2003 Workshop, Karlsruhe, DE, 2003.
[14] J. H. Saltzer, D. P. Reed, and D. D. Clark, "End-to-End Arguments in System Design," *ACM Transactions on Computer Systems*, vol. 2, pp. 277-288, 1984.
[15] H. Liu, M. Beck, J. Huang, "Dynamic Co-Scheduling of Distributed Computation and Repliation," presented at IEEE/ACM CCGrid, Singapore, May 2006.