SANDIA REPORT

SAND2010-7471
Unlimited Release
Printed September 2010

Scientific Data Analysis on Data-Parallel Platforms

Craig Ulmer
Greg Bayer
Yung Ryn Choe
Diana Roe

Prepared by
Sandia National Laboratories
Albuquerque, New Mexico 87185 and Livermore, California 94550

Sandia National Laboratories is a multi-program laboratory managed and operated by Sandia Corporation, a wholly owned subsidiary of Lockheed Martin Corporation, for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-AC04-94AL85000.

Sandia National Laboratories

Telephone:	(865) 576-8401
Facsimile:	(865) 576-5728
E-Mail:	reports@adonis.osti.gov
Online ordering:	http://www.osti.gov/bridge

Telephone:	(800) 553-6847
Facsimile:	(703) 605-6900
E-Mail:	orders@ntis.fedworld.gov
Online ordering:	http://www.ntis.gov/help/ordermethods.asp?loc=7-4-0#online

SAND2010-7471
Unlimited Release
Printed September 2010

Scientific Data Analysis on Data-Parallel Platforms

Craig Ulmer
Greg Bayer
Yung Ryn Choe
Diana Roe

Sandia National Laboratories
P.O. Box 969 MS9152
Livermore, CA 94551-0969
cdulmer@sandia.gov
gbayer@sandia.gov
yrchoe@sandia.gov
dcroe@sandia.gov

Abstract

As scientific computing users migrate to petaflop platforms that promise to generate multi-terabyte datasets, there is a growing need in the community to be able to embed sophisticated analysis algorithms in the computing platforms' storage systems. Data Warehouse Appliances (DWAs) are attractive for this work, due to their ability to store and process massive datasets efficiently. While DWAs have been utilized effectively in data-mining and informatics applications, they remain largely unproven in scientific workloads. In this paper we present our experiences in adapting two mesh analysis algorithms to function on five different DWA architectures: two Netezza database appliances, an XtremeData dbX database, a LexisNexis DAS, and multiple Hadoop MapReduce clusters. The main contribution of this work is insight into the differences between these DWAs from a user's perspective. In addition, we present performance measurements for ten DWA systems to help understand the impact of different architectural trade-offs in these systems.

Acknowledgments

The authors gratefully acknowledge that this would not have been possible without the help and equipment access we received from Netezza, XtremeData, and LexisNexis. From Netezza we thank Matt Campbell for performing tests on TwinFin and Sridhar Srinivasan and Jim Searman for their application development suggestions. From XtremeData, we thank Ali Sajanlal and K.T. Sridhar for adapting our initial implementation to a UDF and performing tests on our behalf. From LexisNexis we thank Matt Barsalou and Larry Deters for their support in developing ECL versions of the algorithms as well as optimizing and running the codes on their DAS system. We also acknowledge Jeffery Howard, David Barrish, Gavin Halliday, Nisha Bhatt, and John Simmons at LexisNexis for their support in this project. Finally, we thank Andy Yoo at Lawrence Livermore National Laboratory for his help identifying DWA machine capabilities.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

In nearly all scientific domains, researchers employ modeling and simulation tools to help answer complex research questions. In general, scientific users typically model a situation that is of interest and then utilize simulation tools to evaluate the model. Once a simulation completes, visualization and analysis tools are used to help users interrogate the results generated by the simulation in order to gain insight into the details of what took place during the simulation. Depending on model fidelity and computational complexity, simulations running at full scale on high-end computing platforms may take days or weeks to complete and can produce terabytes to petabytes of output data. The shear volume of this data makes it unwieldy to manage and analyze through traditional, offline approaches.

1.1 Mesh-Based Simulations

Sandia National Laboratories has a long history of using simulation tools on high-performance computing (HPC) platforms to solve complex problems relating to national security. In the Advanced Simulation and Computing (ASC) [16] effort, Sandia has developed many parallel simulation codes [4] for evaluating mechanical, thermal, and electrical properties of complex systems that are subjected to harsh environmental factors. These simulations help determine how well a system will be able to complete its intended function under different operating conditions. Simulation ultimately provides analysts with an opportunity to gain insight without necessarily requiring the expensive testing of actual components.

Many mechanical and thermal simulation tools at Sandia are based on meshed representations of physical objects. For example, finite element method (FEM) [20] codes discretize each physical object in a simulation into a mesh of elements and then apply numerical analysis on each element to determine how the object interacts with other objects as the simulation progresses. Mesh-based simulation datasets are generally comprised of two types of data: structural data and variable data. Structural data provides geometric information about the objects and is organized in a hierarchical form: a simulation contains multiple objects, an object is defined by its mesh, a mesh is defined by its elements, and an element is defined by its vertices. While all elements in a mesh typically share the same geometric shape (e.g., hexahedron or tetrahedron), Sandia's applications typically employ unstructured meshes with non-uniform elements. The variable data portion of a dataset contains data values that were

Table 1.1. A 100M element simulation can generate many terabytes of data.

Table		Rows	Size
Structural Data	Element	100 Million	3.2 GB
	Vertex	100-800 Million	2.4 GB - 19.2 GB
Variable Data	Element	20 Billion	2.5 TB
	Vertex	20-160 Billion	2.5 TB - 20 TB

calculated during the simulation. A variable (e.g., pressure, temperature, or displacement) is associated with either elements or vertices in the mesh. A variable's data is generated at each timestep of the simulation for all elements or vertices. As such, variable data is often much larger than structural data in a dataset.

High-fidelity simulations can produce datasets that are very large. While typical production runs may be on the order of just a few million elements, leading edge computing platforms have been used to process simulations with more than 100 million elements. As a means of illustrating how quickly datasets grow, Table 1.1 lists the amount of data that would be hosted in a hypothetical dataset for a 100M element simulation, where 16 element and 16 vertex variables are traced for 200 timesteps. While variable data is much larger than structural data, it is important to note that the structural data can be larger than main memory in a single host computer. Many existing data analysis tools cannot process datasets this large because their analysis algorithms were written in an in-core fashion that assumes structural data can be housed in host memory. As such, there is a strong need in the scientific community for post-processing analysis tools that can handle massive datasets in an out-of-core manner. At Sandia, ParaView [9] has become the distributed analysis framework of choice. However, other solutions are of interest.

1.2 Capability Computing Storage

A universal constant in HPC is that scientific users will always need more computing power than available systems offer. When new HPC platforms are brought online, they are strategically designed to improve either the computing capacity of a site or its computing capability. Capacity computing utilizes a new HPC platform's resources to process existing simulations in a shorter amount of time than was previously possible. In contrast, capability computing systems are designed to allow users to increase the fidelity of their simulation models. Capability systems typically push the boundaries of what can be accomplished through modeling and simulation, and are therefore the focus of a considerable amount of research in HPC.

Leading-edge capability systems are custom-built, massively-parallel processing (MPP) systems. In order to increase the amount of computing that can take place in a given footprint, capability system architectures typically separate computing resources from storage resources. For example, Sandia's Red Storm [14] platform employs close to 13,000 diskless compute nodes to perform an application's parallel processing, a separate 1.7PB disk farm for housing persistent storage, and 320 I/O nodes for handling an application's disk requests. Storage systems in HPC systems typically employ a cluster file system such as Lustre [8] to allow data to be striped across many disk nodes and cached in memory in order to hide the high cost of disk access. As such, today's storage systems are often dedicated clusters running a parallel storage application that services load/store requests in a high-performance manner.

1.3 Data Warehouse Appliances

The availability of massive customer databases in industry has resulted in a strong demand by businesses for data mining tools and systems that can allow a company to discover consumer trends in real time. In response to these data mining needs, a number of companies have developed stand-alone products known as data warehouse appliances (DWAs) that enable users to accomplish data analysis operations on parallel platforms without requiring a detailed knowledge of parallel processing techniques. DWAs typically provide (1) a large number of parallel storage devices that enable high-performance disk use, (2) near-storage processing in order to execute computations where the data resides, and (3) a programming interface that allows users to pose data processing commands. DWAs are very appealing to industry because they can be treated as appliances. In many cases, a company can simply purchase one or more racks of a SQL-based DWA and get genuine speedups over a sequential database without having to make significant modifications to their SQL programs.

There are many DWA systems available today. We divide these systems into two categories, based on their programming interfaces. First, SQL-based DWAs are designed to be plug-in replacements for existing database systems. SQL is well known and provides users with a robust data-processing language that fosters data-parallel operations that can in turn be parallelized by a database. Netezza, XtremeData, Oracle, Teradata, and Greenplum all offer SQL-based DWA products. Second, dataflow-based DWAs are designed to provide a flexible platform for manipulating large-scale datasets when users are more willing to express their data-processing commands in other data-parallel languages. Examples of these systems include Hadoop, the LexisNexis DAS, Sector and Sphere [13], and IBM's InfoSphere [17].

Chapter 2

DWAs for Scientific Analysis

A number of institutions in the scientific community are currently designing and deploying a new generation of petaflop-class computing hardware that will be capable of processing larger, higher-fidelity models than ever before. While these capability systems will enable researchers to conduct higher-quality science, we make three observations about trends in the HPC landscape that directly impact how scientific users will make use of capability systems in the next decade:

Increased Dataset Sizes: Increases in simulation size and fidelity correspond to increases in the amount of data generated by a simulation. With current capability systems already generating multi-terabyte datasets, we expect that simulations in the near future will routinely generate tens to hundreds of terabytes of data.

Constant Disk and LAN Rates: While fast solid-state storage devices (SSDs) are slowly making their way to market, traditional hard drives are still the dominant storage option for massive datasets. Given that hard drive transfer rates have not improved significantly in a number of years, future storage systems will employ larger disk arrays in order to satisfy simulation I/O needs. Similarly, link rates for transferring data out of a capability system are not likely to improve substantially in the upcoming years. As such it will become increasingly more expensive to move data out of the compute platform.

Capability Computing Consolidation: While the per-core cost for computing systems has dropped dramatically over the years, the total cost to build, power, and maintain a capability system has not. Under pressure to construct new resources as cost effectively as possible, multiple institutions and laboratories are pooling their efforts to create shared capability platforms. These consolidations emphasize the need for better distance computing practices, where processing and analysis are conducted on equipment that is housed at a location that may be far away from the user.

Based on these observations, we assert that the age-old model of moving data to the user's analysis code is infeasible and must instead be reversed. In order to accommodate data analysis on massive datasets, the storage systems for capability systems must be enhanced to provide processing within the storage system. Similarly, these new systems must take advantage of data-parallel programming interfaces that shift the burden of parallel programming from the user to the storage system when possible. DWAs represent an attractive option for this work, because they are already widely deployed to solve similar problems in other fields.

While DWAs have been successfully utilized to solve many informatics problems, there have been relatively few efforts that have examined whether they are applicable in scientific problems involving massive datasets. The intent of this work is to gain insight into the trade-offs involved in utilizing a DWA to analyze scientific datasets. For this paper we have adapted two mesh-based analysis operations to execute on four different DWA families. In addition to providing implementation details for each DWA, we present performance measurements for the algorithms on both prototype and production DWA hardware to help identify both the strengths and weaknesses of each system. Finally, we present a number of observations based on our experiences with the DWAs in hopes of motivating future work in this field.

Chapter 3

Data Warehouse Appliances

For the purposes of this paper we chose to focus on four different DWA platforms that are in use today. For SQL-based DWAs we selected both Netezza and XtremeData database systems. For dataflow-based DWAs we selected the LexisNexis Data Analytics Supercomputer (DAS) and a Hadoop cluster. We adapted our algorithms to run on each system, and then measured performance on prototype and entry-level production systems.

3.1 Netezza

With the introduction of the Netezza Performance Server (NPS) product line in 2002, Netezza became one of the first companies to promote data warehouse systems that were truly standalone appliances for data analytics [10]. The NPS products can be described as "custom-everything" systems because they employ both custom hardware and custom software to implement high-performance, parallel databases.

In terms of hardware, Netezza employs a large number of disk blades to perform database operations in parallel. As illustrated in Figure 3.1(a), each blade is equipped with a processor, a field-programmable gate array (FPGA) co-processor, DRAM, and a commodity hard drive. Blades are interconnected through a commodity Gigabit Ethernet (GigE) network. While adding to the expense of the system, the FPGA co-processor enables Netezza to perform many unique operations in real time that other systems cannot. For example, the FPGA can be configured to perform data decompression and column filtering as data is read off the disk. These capabilities enable the hardware to deliver data to the processor at rates that are higher than what the disk channel could physically support. Observing that hard drives are often a bottleneck in I/O intensive applications, this optimization can result in a significant performance gain for large data applications.

In terms of software, Netezza employs a parallel database engine that allows the system to stripe data across disk blades and execute a query in parallel. Rather than rely on indexing to help the database locate relevant records, Netezza performs scans on entire tables. While this brute-force approach may at first appear to ignore the fundamentals of modern database design, it results in a scalable system with predictable performance. If a user increases the dataset size or needs better performance, the user simply adds more nodes. Similar to other

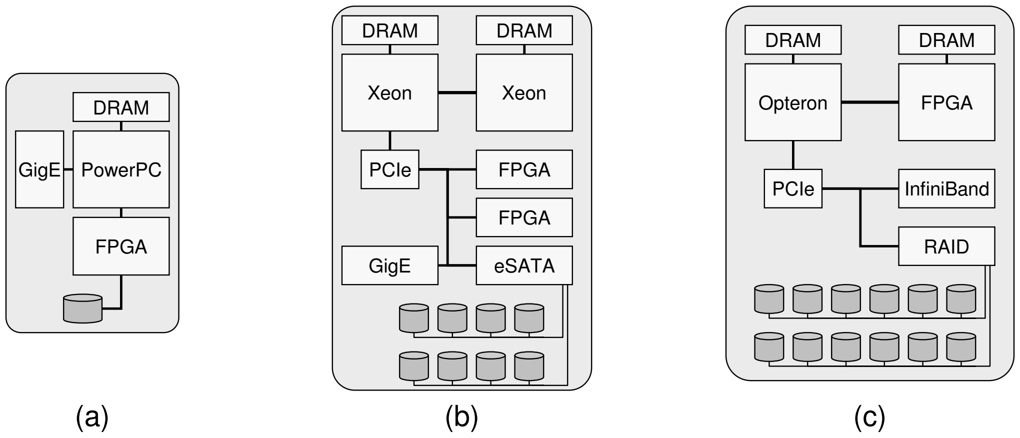

Figure 3.1. Node architectures for (a) Netezza Mustang, (b) Netezza TwinFin, and (c) XtremeData dbX.

databases, Netezza allows users to supplement the SQL syntax with user-defined operators. These operators are written in C++ and are either user-defined functions (UDFs) or user-defined aggregates (UDAs). UDFs operate on a single row at a time, producing one output value for a given list of column inputs (e.g., compute the squared sum of a set of input values). UDAs perform the same work as a UDF, but also provide a means of collapsing the results for all rows into a single value (e.g., find the row in a table with the second largest squared sum value for specific columns). In our experience, UDFs and UDAs are a convenient way of expressing data-processing computations that would otherwise be difficult to construct using only the built-in operators provided in SQL.

During the course of our work we have had the opportunity to experiment with two generations of Netezza products, each optimized for different design goals. In the older-generation "Mustang" architecture, Netezza targeted power efficiency and matched an embedded processor to every disk in the system in order to achieve high node densities. The Mustang's blade employed a PowerPC for computation and housed a SATA controller for the disk inside the FPGA. While this architecture performed well for many text-based operations, the PowerPC's lack of floating-point hardware made it underpowered for many scientific applications. In response to these criticisms, Netezza has recently developed a new generation of hardware named "TwinFin" that leverages commodity x86 hardware (Figure 3.1(b)). A TwinFin blade houses two FPGA coprocessors (via PCIe) and eight disks (via eSAS).

3.2 XtremeData

XtremeData's dbX product [2] is a SQL-based DWA that utilizes both parallel processing and custom FPGA hardware accelerators to increase the rate at which the database can satisfy analytical queries. XtremeData's design approach in the dbX platform can be described as "custom when needed", as commodity hardware and software is widely utilized throughout the DWA. As illustrated in Figure 3.1(c), the individual nodes are server-class workstations, each equipped with multiple x86 cores, large amounts of DRAM, and a large RAID array of disks. Workstations are interconnected through a high-speed InfiniBand (IB) network.

The one piece of custom hardware employed in the dbX architecture is an FPGA accelerator designed by XtremeData. This accelerator card plugs into an AMD processor socket on the motherboard and communicates with the CPUs in the node via a high-bandwidth, low-latency HyperTransport connection. XtremeData's accelerator features an FPGA that is faster and has more capacity than Netezza's current FPGA accelerator. XtremeData's FPGA accelerator has benefited from several revisions, and has been productized as a standalone device that reconfigurable computing researchers have utilized in a variety of ways.

In terms of software, XtremeData has adapted the PostgreSQL database to run in parallel on the dbX platform. This software has been modified to make use of the FPGA accelerator for specific, time-consuming operations such as joins, sorts, groupings, and orderings. While the dbX's FPGA accelerator programming interface is not available to end users, XtremeData's application engineers can be contracted to port high-value, application-specific database operations to the FPGA. The dbX's software is designed to orchestrate the flow of data from disks to the FPGA in a pipelined manner, and performs dynamic load balancing at runtime for every query step.

3.3 LexisNexis DAS

As a company, LexisNexis has a long history of providing both large-scale archival content and data processing systems that allow users to perform real-time searches on massive datasets. In response to the growing demand for more flexibility in archival searches, LexisNexis constructed a parallel processing platform called the Data Analytics Supercomputer (DAS) [3]. DAS can be described as "custom software for commodity hardware", as it employs a generic Linux cluster for the underlying hardware and a custom-built software framework for managing the flow of data through the system. Observing that SQL can be restrictive in terms of programming ease and parallel performance, LexisNexis developed the Enterprise Control Language (ECL) as a means of expressing dataflow operations, and then constructed compiler and linking tools to efficiently map computations to DAS resources. The framework employs system software and middleware components to provide a custom execution environment and distributed filesystem.

The DAS is comprised of two different compute environments: the Data Refinery (or "Thor") and the Data Delivery Engine (or "Roxie"). The Data Refinery employs a large number of nodes to perform the core computations specified by the ECL query. Data is stored in the Data Refinery in a record-oriented, distributed file system that supports multiple data file formats. Data is partitioned automatically by the system and is replicated at least once on a neighboring node in order to provide fault resilience. The remaining nodes in the system implement the Data Delivery Engine, which caches and indexes query results to improve the interactive performance of the system. Due to the nature of our experiments, we did not utilize the Data Delivery Engine in our work.

ECL is a high-level declarative language that is oriented towards dataflow computations. ECL can be challenging for new users to master as (1) program execution is derived from the sequence of dataflows and transformations rather than the order of the program's statements and (2) transformations can be applied either locally or globally. However, ECL offers a rich syntax that features a variety of built-in transformations. Programming is also simplified through an interactive IDE and debugging tools.

3.4 Hadoop Clusters

Hadoop [6] is an open source framework for performing data-parallel operations on commodity cluster hardware. Hadoop was initially constructed by researchers at Yahoo! as an open source Java clone of the Google File System (GFS) [12] and Google's MapReduce [11] framework, but has since grown into the basis for a number of different out-of-core data processing projects. At its core, Hadoop is comprised of two components. First, the Hadoop distributed file system (HDFS) [7] provides a scalable, general-purpose file system for distributing data across the local disks in a cluster in an efficient and reliable manner. HDFS operates on large blocks of data (64 MB by default) and is responsible for transferring and synchronizing data between nodes in the cluster as needed by applications. Second, Hadoop provides a MapReduce framework for performing computations, where map tasks perform computations on independent regions of data and reduce tasks combine the results of map tasks. In between map and reduce phases of a job, users may also apply a reduce task on local data with a combiner task in order to decrease the amount of data that is transmitted over the network. The Hadoop MapReduce framework requires users to organize their data into a key-value data format. This format enables the framework to provide users with built-in support for common data processing operations (e.g., sorting key-value lists), and encourages users to organize their data and computations in a way that maps well to parallel-processing platforms.

Hadoop has received considerable interest in the data processing community recently because it is very accessible. In addition to being open source and free to use, Hadoop can be installed and evaluated on a wide variety of platforms. Hadoop was designed with low-cost, commodity cluster hardware in mind. While Hadoop clusters typically expect compute nodes to be equipped with local disks, researchers have demonstrated that high-end, disks-on-the-side clusters can also be utilized effectively [18]. Hadoop is also supported in several commercial cloud-computing endeavors, such as Amazon's elastic compute cloud (EC2) [1]. Thus it is possible for researchers to pay a vendor to run a large Hadoop application at scale if needed. The main drawback of utilizing a Hadoop cluster as a DWA is that users must convert their data processing applications to a form Hadoop can process. Additionally, users often find it is challenging to configure Hadoop in a way that maximizes performance.

3.5 DWA Test Systems

As a first step in evaluating whether DWAs can be utilized effectively in mesh analysis applications, we acquired access to DWA test systems for Netezza, XtremeData, LexisNexis, and Hadoop. For each of these systems we were able to utilize two different hardware configurations. At a high level the two hardware configurations selected for each DWA in this work target (1) an prototype-level system suitable for a single researcher and (2) a production-level system suitable for a small work group. Due to the hardware differences between DWA products (e.g., CPU speed and count), it is challenging to make a fair performance comparison between these systems without a normalization factor such as price, power, or physical size. However, the intent of this paper is to report on our mesh analysis experiences with DWAs and not a detailed performance comparison study. As such, we present performance measurements in terms of observed wall clock timings, and find that a coarse-grained comparison between two levels of platforms is sufficient. The specific systems utilized in this work are described as follows.

Netezza: For Netezza's prototype-level system we utilized a Netezza Performance Server 10050 housed at Sandia. This half-rack system employs 54 Mustang-generation blades (PowerPC, 1GB DRAM, FPGA, and a SATA disk). For the production-level system Netezza provided access to a TwinFin6 system with 6 blades (dual quad-core Xeon CPUs, 16 GB DRAM, two dual-core FPGAs, and 8 eSAS disks). Both systems utilize GigE internally for communication and a separate head node for providing an ODBC API to the SQL database.

XtremeData: XtremeData provided access to an 8-node system (dbX 1008) and a 16-node system (dbX 1016). The nodes in these systems employed substantial compute power (a six-core Opteron CPU, 32 GB DRAM, an in-socket FPGA, and 12 disks). An InfiniBand network is utilized for communication between nodes. A separate head node controls the system and hosts external interfaces.

LexisNexis DAS: The prototype-level DAS system is a Sandia system comprised of 20 nodes (quad-core Xeon CPUs, 4 GB DRAM, and two hard drives), 10 of which perform Data Refinery computations. LexisNexis provided access to a production-level system comprised of 60 nodes (quad-core Xeon CPUs, 8 GB DRAM, and one hard drive), 32 of which perform Data Refinery computations.

Hadoop Clusters: Hadoop's accessibility made it possible to perform tests on a number of different clusters. Initial tests were performed on a retired Sandia cluster named Decline that is disk-full and comprised of 32 functional nodes (dual AMD Opteron processors, 4 GB DRAM, a pair of SATA hard drives, and GigE network). In order to test code portability and scaling, we performed additional Hadoop experiments on Amazon's EC2 cloud computing platform. Amazon's systems differed from the Decline cluster in that compute resources

Table 3.1. Different DWAs utilize different hardware components.

Platform	Compute Nodes	Cores/ Node	Memory/ Node	Disks/ Node	FPGAs/ Node
Netezza Mustang	54	1 PowerPC	1 GB	1	1
Netezza TwinFin6	6	8 x86	16 GB	8	2
XtremeData dbX 1008	8	6 x86	32 GB	12	1
XtremeData dbX 1016	16	6 x86	32 GB	12	1
LexisNexis DAS-20	10	4 x86	4 GB	2	0
LexisNexis DAS-60	32	4 x86	8 GB	2	0
Hadoop-Decline	32	2 x86	4 GB	2	0
Hadoop-Amazon-32	32	2 x86	1.7 GB	1	0
Hadoop-Amazon-128	128	2 x86	1.7 GB	1	0

are virtualized and shared among many users. After experimenting with Amazon's small, medium, and large hardware configurations, we selected the medium node configuration as it most closely resembles Decline's hardware. Amazon's description of a medium-sized node states that it features two virtual Xeon CPUs from 2007, 1.7 GB of DRAM, and "moderate" I/O performance. We selected 32 and 128 nodes to represent the prototype- and production-level DWAs. Hadoop versions 0.19.0 and 0.20.1 were used in the Decline and Amazon clusters, respectively. On both systems, HDFS was configured to use the default block size (64 MB).

Characteristics for the different DWAs are summarized in Table 3.1.

3.6 DWA Architectures

Based on the hardware characteristics listed in 3.1, we observe that there are three general architecture strategies utilized in DWAs.

Large Scale, Embedded Nodes: As seen with Netezza's Mustang architecture, one approach to building a high-end DWA is to construct a low-power processor-disk blade and then replicate the blade to a massive scale. One argument for this approach is that many information retrieval systems are I/O bound and that embedded processors can keep pace with rotational hard drives. A criticism of this approach is that these systems may perform poorly on large data applications where data is frequently exchanged between nodes. This approach has recently been revitalized in research projects such as the Fast Array of Wimpy Nodes (FAWN) [5].

Medium Scale, Generic Nodes: As observed in both Hadoop clusters and the LexisNexis DAS, many DWA architects favor a commodity approach that utilizes as many generic workstations as possible. A common rule of thumb in this approach is simply to match the number of disk spindles to the number of processor cores for a node in order to create a balanced architecture. This approach is economical at medium scales and accessible to a wide audience of researchers.

Small Scale, High-Performance Nodes: The third approach is simply to use a smaller number of high-end workstations, and add hardware accelerators as needed. Netezza TwinFin and XtremeData exemplify this approach, packing as many resources into a single multiprocessor system as possible. While expensive, these systems perform well in both low and high data sharing applications. Reducing the number of nodes in a system may also simplify system management tasks and reduce power and real estate requirements in the data center.

Chapter 4

Mesh Schemas

The first challenge in adapting mesh analyis algorithms to run on DWAs is defining a suitable way to represent the mesh's data in the DWA's native storage format. An ideal schema strikes a balance between representing the data in a space-efficient manner and representing it in a way that makes it easy for users to access efficiently. For example consider the problem of defining the coordinates for all elements in a mesh. Given that elements often share vertices with their neighbors, the space-efficient means of representing this information is to use two tables (Figure 4.1): one for holding the coordinates of all the unique vertices and another for defining the vertex indices that belong to each element. This approach is commonly utilized in both computer graphics applications and mesh data files. While the lookup tables minimize data replication, an application must perform two lookups whenever an element's coordinates are required in a calculation.

Node ID	X	Y	Z

Node lookup table

Element ID	Node 1	Node 2	Node 3	Node 4	Node 5	Node 6	Node 7	Node 8

Element lookup table

Figure 4.1. Space-efficient schema for structural data.

We initially selected a space-efficient schema for storing our mesh datasets, expecting that smaller data on disk would ultimately yield better I/O performance. However, early experiments revealed that the amount of time required to assemble input data from different tables could significantly hamper performance on all of the DWAs we utilized. Additionally, the task of performing two lookups to determine an element's coordinates added extra complexity to an analysis application. In response to these issues, we investigated alternate schemas that replicated data in the dataset to reduce the number of lookups that are required to perform common operations. In particular, we decided to extend the element lookup table to also include the coordinates of all the nodes (Figure 4.2). While this approach increases the amount of data stored on disk, it greatly simplifies the programming interface for common operations and improves performance by reducing the number of lookups that must be performed in key operations. Although the availability of node coordinate information in the extended element table means the node table is not strictly required, we still make it available in order to improve the speed of node-specific operations.

Node ID	X	Y	Z

Node lookup table

Element ID	Node 1 ID, X,Y,Z	Node 2 ID, X,Y,Z	Node 3 ID, X,Y,Z	Node 4 ID, X,Y,Z	Node 5 ID, X,Y,Z	Node 6 ID, X,Y,Z	Node 7 ID, X,Y,Z	Node 8 ID, X,Y,Z

Extended element lookup table

Figure 4.2. Lookup-efficient schema for structural data.

In addition to structural information, mesh datasets also typically store time-varying data that represents changes to variables of interest during a simulation. A variable can be associated with either the mesh's vertices or its elements. For example, simulations where a mesh physically moves in space over time usually capture structural changes with a displacement variable that is associated with the mesh's vertices. At each time step, the variable provides position updates for each vertex. We considered multiple schemas for housing time-varying data. We first considered a unified approach that stored both node and element data in a single table (Figure 4.3). While this approach simplified the data structures by routing all requests through a single table, it greatly increased data size and access time because each entry required its own identifier information. Additionally, this format was inappropriate when data values used different numerical formats. Ultimately we switched to an approach that separated element and node data into two tables. In order to save space, all variables for a particular node or element at a particular timestep are recorded in a single row in the table (Figure 4.4).

Timestep ID	Variable ID	Node/Element ID	Variable Value

Node and Element variable data table

Figure 4.3. A unified schema for storing variable data.

Timestep ID	Node ID	DISP X	DISP Y	DISP Z	NVAR1	NVAR2

Node variable data table

Timestep ID	Element ID	EVAR1	EVAR2	EVAR3

Element variable data table

Figure 4.4. Time-sequence data is stored in node and element variable data tables.

We constructed multiple dataset generators to create data for the experiments presented in this paper. These generators allowed us to vary dataset size, and also provided us with reference data that could be moved between platforms without legal issues. Each dataset generator produces a set of tab-separated text files for the various fields in the dataset. For the SQL-based DWAs, this information was easily digested into the databases using built-in command line tools that automatically convert the data to the format specified by the database schema. Similarly, the LexisNexis DAS's ability to read from CSV files was employed to convert data into a format that could then be "sprayed" on compute nodes. For Hadoop it is desirable to convert the dataset to a binary format (i.e., Hadoop sequence files) that Hadoop applications can access. We constructed a program to perform this conversion and then inserted them into HDFS, which automatically replicates and distributes the data. A simple Hadoop reader class was then constructed for reading data from the binary format.

Chapter 5

Threshold Volume Calculation

The first data analysis operation that we adapted to run on our DWAs was a threshold volume calculation. Analysts often need to determine how large a particular effect is within a mesh at a specific timestep. For example, one analyst working on safety factors for hydrogen refueling stations for automobiles conducted a simulation (Figure 5.1) where an open nozzle leaked hydrogen gas into a room with multiple chambers. The analyst computed the total volume a particular mole fraction of the gas occupied at each timestep, as well the rate of change for the volume between timesteps. These numbers quantified the amount of leakage that took place during the simulation and were also used to determine when the simulation had reached a steady-state solution.

5.1 Algorithm and Data

The threshold volume calculation can be implemented in a simple, brute-force manner through a marching-cube [15] style approach that sums the contribution of each element to a timestep's total volume estimate. For this work we assume that elements are hexahedra and that the threshold data variable (e.g., hydrogen mole fraction) is associated with the mesh's vertices. For a particular timestep, the algorithm examines each element in the mesh and determines if the variable at the element's vertices is above a specified threshold value. A hexahedron element that has multiple vertices exceeding the threshold is decomposed into six tetrahedra. Each tetrahedron that has all four vertices above the threshold is included in the final volume estimate. Tetrahedron volume estimation is accomplished through straightforward vector math. Our work also assumes that meshes deform as time in the simulation progresses, and that the dataset includes displacement information for each vertex at every timestep. Therefore, in order to accurately compute the threshold volume at a particular timestep, the mesh's vertex coordinates must first be updated by their displacement values.

A synthetic mesh dataset generator was constructed for the threshold volume calculation. It created datasets comprised of many, independent hexahedral elements that are randomly placed and annotated with randomly-generated variable data. The data generator was configured to produce a variety of meshes, ranging from 1M elements to 100M elements. Given that the random data values in these datasets cause different workloads for the volume calculation algorithm, particular attention was placed on utilizing the same dataset

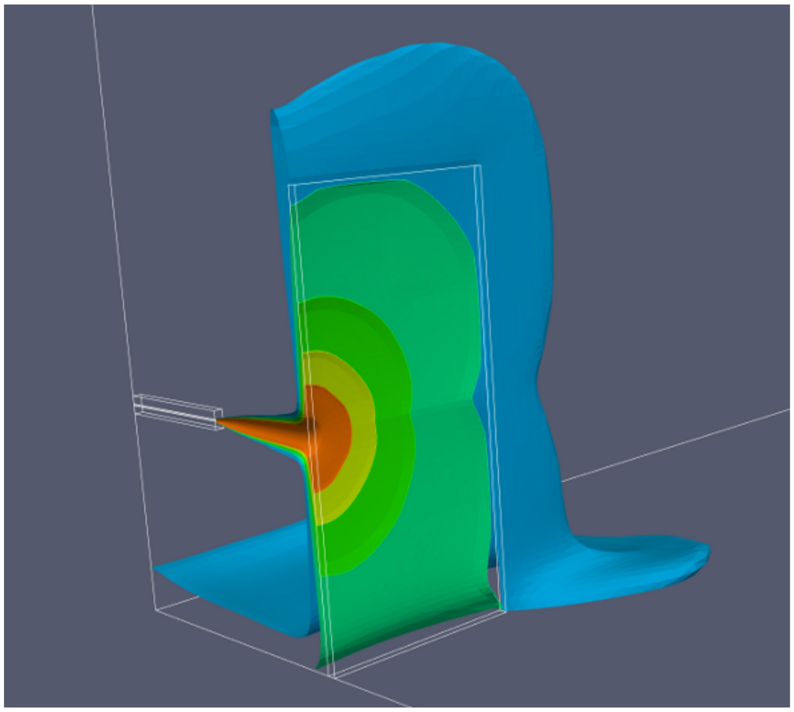

Figure 5.1. Multiple threshold volumes in a simulation.

and algorithm input parameters on all DWAs. These parameters were selected to give a challenging workload where large amounts of data processing could not be avoided.

5.2 Netezza and XtremeData Implementations

For the SQL platforms we implemented the volume calculation utilizing two different strategies that employed the same three phases of operation. First, a simple threshold operation is performed on the vertex data to create a smaller temporary table consisting only of element vertices that are above the desired cutoff. Second, the relevant data necessary for the computation is assembled. Finally, a running sum for the threshold volume is computed.

The difference between the two implementations is in the manner that data is assembled. The first strategy calculates the volume on a per-element basis. It assembles all relevant values (e.g., coordinate displacements and vertex variable data values) for each element with one statement for the volume calculation. This operation requires eight joins: one for each vertex in the element. The volume is calculated through a single SQL statement that is the union of six subqueries (one for each tetrahedron) and no additional joins. We refer to this implementation as the 1X8-join approach.

The second strategy calculates the volume on a per-tetrahedral basis. The vertex data is

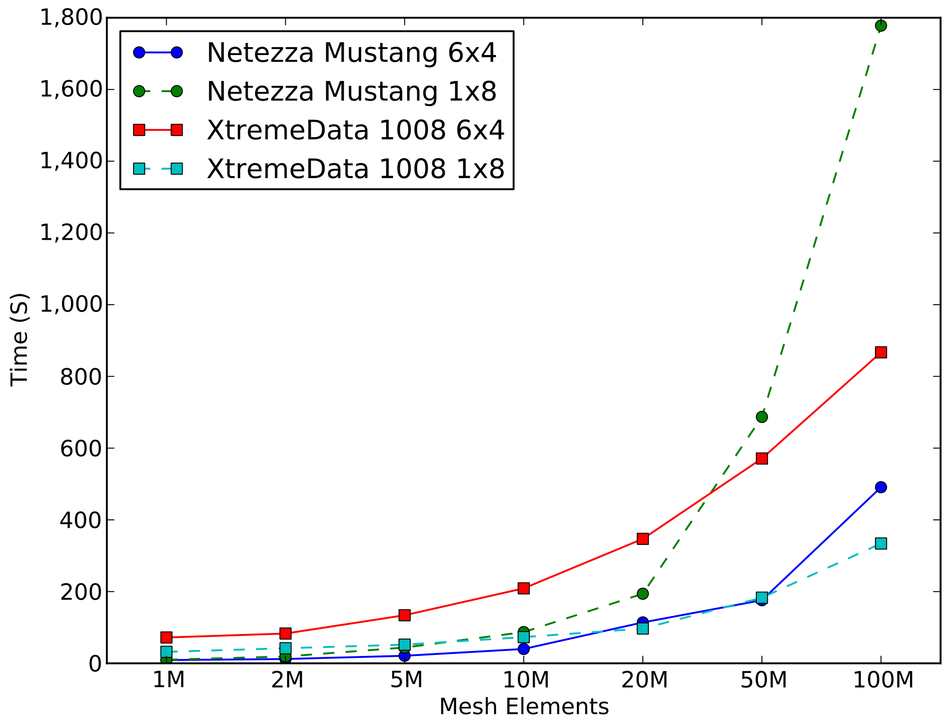

Figure 5.2. Joining strategy affects performance in the database systems.

joined during each tetrahedral volume subquery. This operation requires only four joins for each tetrahedral volume calculation (one for each vertex). Processing a full element therefore requires six subqueries, each with four joins. We refer to this implementation as the 6X4-join approach.

Both algorithms were executed on both the Netezza Mustang DWA and the XtremeData prototype system. As illustrated in Figure 5.2, the 6X4-join approach performed better than the 1X8-join approach in most cases. However, 1X8-join did prove to be slightly better for small dataset sizes on the XtremeData prototype. Given that this difference was minimal, we selected the 6X4-join approach as the primary implementation for comparison in both systems.

5.3 LexisNexis DAS Implementation

The LexisNexis DAS implementation of the threshold volume calculation begins with the distribution of the dataset's Element and Node values to different computers in the system.

This work is accomplished through a DISTRIBUTE function, which uses the data's Element id value as a key for load balancing the distribution. Next, compute nodes filter the Node data values to eliminate entries that have data values less than the specified threshold. Once this downselect takes place, Element and Node entries are merged via a JOIN operation. A ROLLUP operator then gathers all of an element's node data values into a single record for a given timestep. Finally, the individual element volumes are calculated through a PROJECT operation and then combined through a SUM operation.

5.4 Hadoop Implementation

The Hadoop implementation of the threshold volume calculation performs all of its work in a single pass. A number of map tasks are used to extract and join information from the binary input data files in parallel and generate all of the data values that are needed for the computation. Thresholding is performed during this map operation in order to remove unnecessary data values as early as possible. These map tasks then compute the individual volumes of elements as they are read. As each volume is calculated, it is appended to the output key-value list using a key of "1". A local combiner task is then used to condense all entries with the same key ("1") to a single key-value. Finally, a global reduce operation merges the results of each node's combiner into a single sum for the entire cluster.

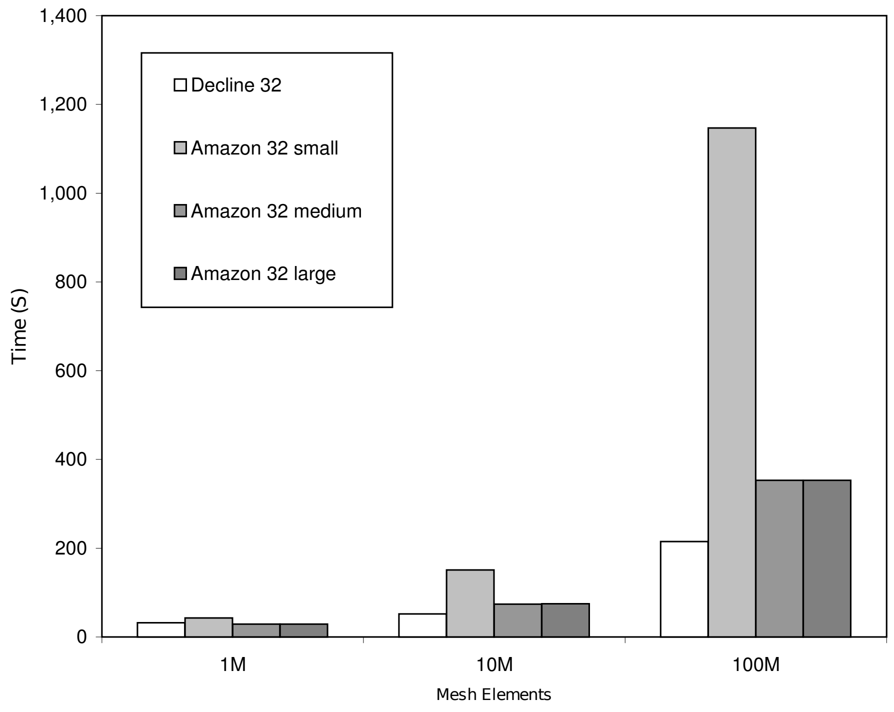

Figure 5.3. Hadoop's volume threshold performance varied for different node configurations.

Hadoop performance measurements were conducted on both the local Decline cluster and Amazon's EC2 resources. For the Amazon systems, testing was performed on "small", "medium", and "large" node configurations, with 32 nodes in each configuration. A performance comparison of these configurations is presented in Figure 5.3. The medium configuration proved to be the most economical choice among the Amazon configurations, although the Decline cluster generally provided the best performance. We attribute part of this gain to having exclusive access on Decline, while all Amazon nodes are shared with other users. Based on these measurements, we selected Amazon's medium configuration in our more detailed experiments.

5.5 Performance Comparison

Following initial prototyping and verification experiments, a suite of detailed performance measurements were conducted on the DWAs. In preparation for these tests, datasets were ingested offline to a native format. While static information was flattened to a single, wide table to remove the need for additional lookups, dynamic data was stored in its original form, forcing the analysis codes to assemble data values as needed at runtime.

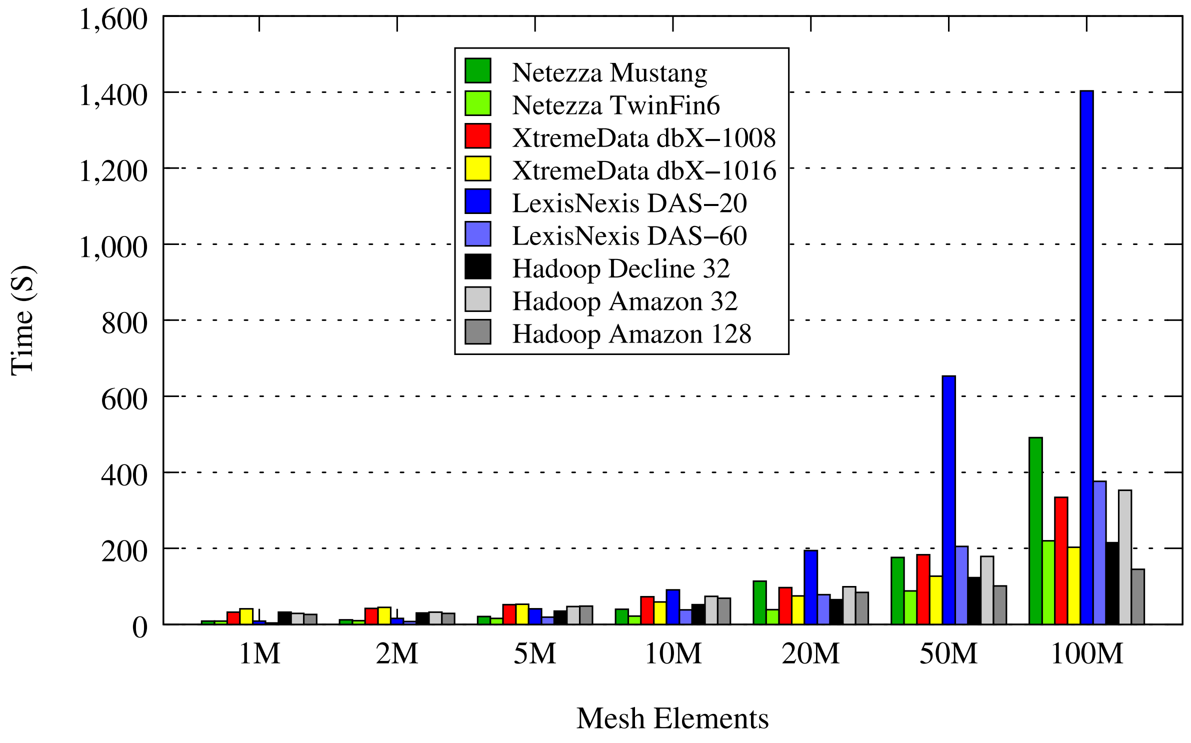

Figure 5.4. Threshold volume calculation for different DWAs.

The DWA performance timings for various dataset sizes are presented in Figure 5.4. As a benchmark, the threshold volume calculation represents a brute-force calculation that largely depends on the DWA's ability to read data off disk and perform a simple set of floating-point computations. While the Netezza Mustang system provided good I/O performance, its lack of floating-point hardware significantly degraded performance. The Netezza TwinFin provided much better performance and was able to achieve the same or better performance than other systems that utilized many more nodes. The XtremeData systems did not perform as well as expected. This slowdown can be partially attributed to our use of a nonstandard join that has not yet been optimized in XtremeData's architecture. The LexisNexis DAS-60 system proved to be significantly faster than the DAS-20. While the DAS-60 lagged the other systems in this test, the speedup between the DAS-20 and DAS-60 is promising. Finally, the Hadoop implementations performed well in this test due to their natural ability to handle brute-force calculations. The 128-node Amazon "medium" cluster provided a 2.4x speedup over the 32-node Amazon "medium" cluster. However, it is important to note that Hadoop's high startup costs caused it to perform poorly on all small dataset sizes.

Chapter 6

Element Pairing

The second analysis algorithm that we adapted to our DWAs is an element-pairing algorithm that is part of a larger application that quantifies how much damage is caused when two objects collide. In the example simulation containing this algorithm, an indestructible wedge object is rammed into a deformable object, as pictured in Figure 6.1. In order to simulate realistic fractures, the deformable object is modeled as two separate meshes that are bonded together at the surface where the collision takes place. Analysts are interested in observing how far element pairs in the seam move apart as the simulation progresses. The distances between these element pairs help quantify cracks, tears, and holes created in the collision.

The central challenge in implementing this analysis is automatically generating the list of element pairs that are pressed against each other at the start of the simulation. Due to the way the meshes are constructed, we cannot assume that a pair of touching elements will share the same vertices or vertex coordinates. Additionally, numerical precision issues dictate that the distance between the faces of two touching elements will be small, but not necessarily zero. Equipped with no additional information about where the meshes are actually touching, the hardship of this pairing problem is that it may require on order of NxM element face distance comparisons.

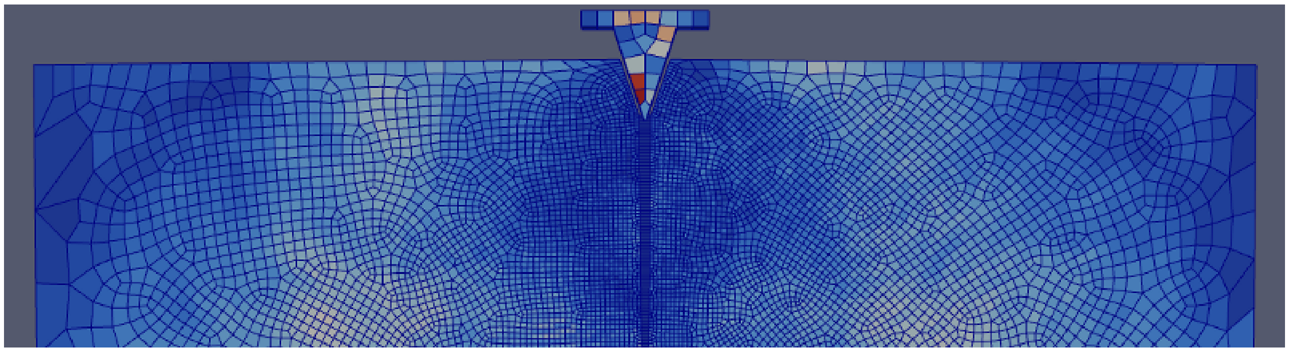

Figure 6.1. Element pairing along a fracture.

6.1 Algorithm and Data

Our approach to implementing the element pairing algorithm involves computing the distances between the exterior faces of the two meshes, and then selecting the pairs with minimum distance between their faces. This work is divided into three phases. First, all faces are generated for each element in each mesh (e.g., six faces per hexahedron). Second, the face list for a mesh is reduced to the set of exterior faces for the mesh by removing all faces that appear more than once in the list. It is important to note that this reduction must take into account the fact that a single face can be represented by different combinations of its vertices (e.g., ABCD is the same as CDAB). Finally, each exterior face for the first mesh is paired to the face in the second mesh that is closest, geographically. For simplicity, we use the distance between face centers as the distance metric.

For the purposes of evaluating the computing performance of the DWAs in the worst case, we chose to deliberately ignore obvious optimizations that reduce the dataset early in the analysis process. For example, we developed a simple bounding-box filter for the first stage of execution that removed all faces in the first mesh that were too far away from the bounding box of the second mesh to be considered. This filter greatly reduced the face list sizes and thus dramatically improved performance. We exclude this filter from the experiments reported in this paper because (1) there are always canonical examples where these types of filters fail and (2) we wanted to present the pairing portion of the algorithm with a sizable amount of work to evaluate how well the systems perform at scale.

A second synthetic dataset generator was constructed for the element pairing applications. This generator produces datasets that are comprised of two meshes that are placed in close proximity to each other. The meshes are offset by small amounts in each direction to reflect a more realistic example.

6.2 Netezza and XtremeData Implementations

The Netezza and XtremeData implementations of the element pairing algorithm were written as a sequence of SQL queries that store intermediate data values in temporary tables in the database. The first task of generating all possible faces for a mesh is performed through six insert statements (one for each face in each hexahedron). Each insert requires five joins to assemble the necessary data: one to locate all of the elements belonging to a mesh and four to procure the coordinates of the face's four vertices. During the insertions, the vertices for a face are averaged together in order to generate a center point that can be used during the final distance estimation task.

The second task of reducing the face lists to just the external faces proved to be more challenging because of the vertex ordering problem. Our solution was to generate a unique key for each face that could be represented as a single data value in the database. This operation sorts each face's vertex IDs through a series of min/max comparisons. The sorted

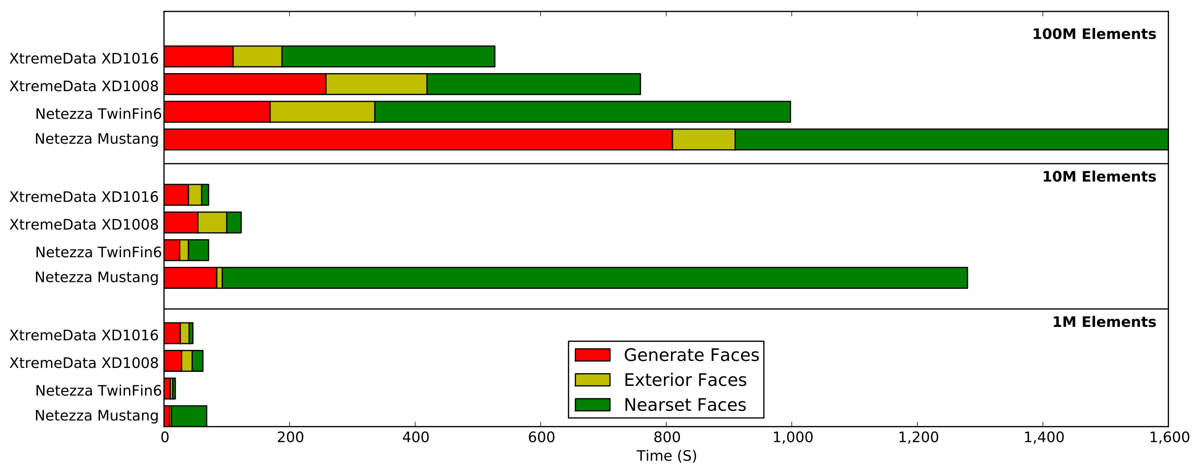

Figure 6.2. Performance breakdown for element pairing on SQL platforms, truncated to 1,600 seconds. The 100M Netezza Mustang run completed in 25,695 seconds.

IDs are then packed into a string that uniquely identifies a face. The unique keys allow us to utilize SQL's "Group By" and "Having (Count=1)" directives to reduce the face list to entries that appear only once in the dataset, and are thus exterior faces. In order to speed up subsequent operations, the unique key for each exterior face is replaced with a unique integer ID.

The final task of locating pairs of touching faces is the most time-consuming portion of the algorithm as it involves NxM comparisons. Similar to the previous step, we utilize the SQL "Group By" clause to compute the distance between every exterior face in one mesh (N) and every exterior face in the other mesh (M), followed by a MIN() aggregate function to reduce the data to the minimal pairings. Through experimentation, we were able to verify that the MIN aggregate properly discards non-minimal values as it steps through the dataset, as opposed to computing all values and then selecting the minimum results.

The initial SQL implementation was debugged on the Netezza platforms and then ported to the XtremeData platforms. While the SQL code functioned properly on XtremeData, the Cartesian join performed poorly. XtremeData's application engineers volunteered an alternative approach that employed a UDF to minimize data transfer for the operation. While this optimization made XtremeData much more competitive, the same optimization had negligible impact when it was ported back to the Netezza system. We also investigated additional UDF optimizations for the first task. However, these optimizations did not significantly impact overall performance due to the long length of the third task.

Performance experiments were conducted on the final version of the algorithms for various mesh sizes. Details for the three phases of execution are presented in Figure 6.2. While the Netezza platforms performed well in the first two phases, XtremeData's third phase performance was significant enough to give it the lead in the 100M element case. The Netezza Mustang platform performed poorly in the final phase due to the communication complexity of distributing data values among a large number of slow nodes. Timing for the 100M case was aborted after 12 hours of execution.

6.3 LexisNexis DAS Implementation

The LexisNexis implementation followed the same three phases of operation as the database systems. The first task of generating the six faces of each element was accomplished by calling ECL's NORMALIZE function six different times to generate the full set of faces for a mesh. The second phase of reducing the data to the set of exterior faces was then accomplished through ECL's SORT and ROLLUP functions, followed by a filter operation to remove non-unique faces. The final task of finding neighboring faces was then accomplished through two operations. First, the PROJECT operation iterated over all faces so that a simple data transformation could be applied to generate a center point for each face. Second, a DENORMALIZE operation enabled all exterior face centers in one mesh to be compared to all faces in another. It is important to note that ECL provided built-in operators that matched the functional requirements of the algorithm, and that the DAS system software automatically managed the movement of data during the computation.

6.4 Hadoop Implementation

The Hadoop version of the element pairing algorithm merges the first two phases of the algorithm into a single MapReduce task. Map tasks read in element data from data sequence files and decompose elements into a collection of faces. Multiple reduce tasks are then used to remove all but the exterior faces in the mesh. In order to identify faces regardless of vertex order, a key is generated for each face that is comprised of the ordered list of vertex IDs.

The final phase performs all distance comparisons between exterior faces. This operation proved to be challenging to implement in MapReduce because it involves comparing results from two different data streams. Similar to other researchers faced with the same problem, we decided to implement this join operation somewhat outside the MapReduce paradigm: we designed a map task that loaded one mesh's face values into memory and then streamed the other mesh's face values through for comparison. In order to handle memory limitations, this phase was designed to support an iterative approach that loads only a portion of the first mesh into memory at a time. If the first mesh fits entirely in memory and multiple output files are acceptable, there is no need to perform a reduce operation here. The performance of the join in this phase was further improved by leveraging Hadoop's Distributed Cache feature to efficiently distribute data to all maps in the cluster.

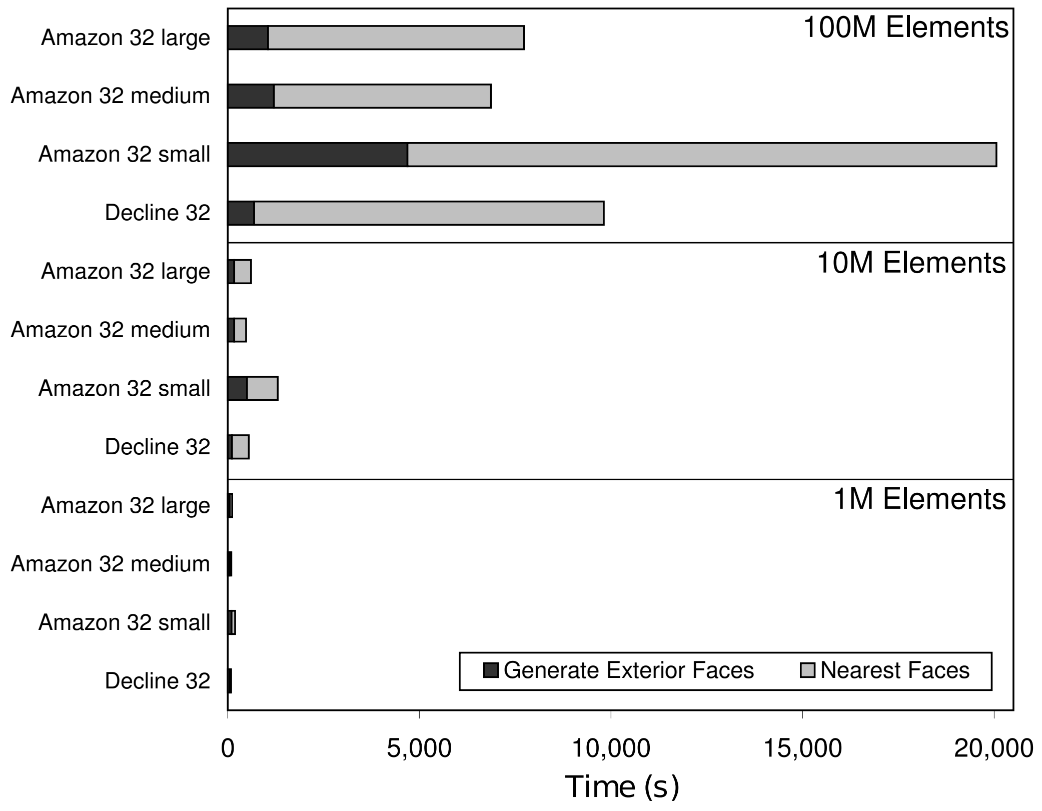

Figure 6.3. Performance breakdown for element pairing on Hadoop platforms.

Performance measurements for different 32-node Hadoop clusters are presented in Figure 6.3. Similar to the other platforms, the nearest faces phase of the work consumes the majority of the processing time in the algorithm and scales with the computational capabilities of the platform. For example, Amazon's medium nodes offer slightly faster CPUs than the large nodes and therefore achieve better performance during this stage. Similarly, the first phase of generating exterior phases is largely I/O bound. As such, the Decline cluster performs better during this phase because its local disk resources are dedicated and not shared in the way Amazon's clusters are provisioned.

6.5 Performance Comparison

Following initial prototyping and verification experiments, a suite of detailed performance measurements were conducted for the element pairing algorithms on each DWA. Datasets were once again ingested offline in a manner that flattened the static portion of the dataset to improve access efficiency.

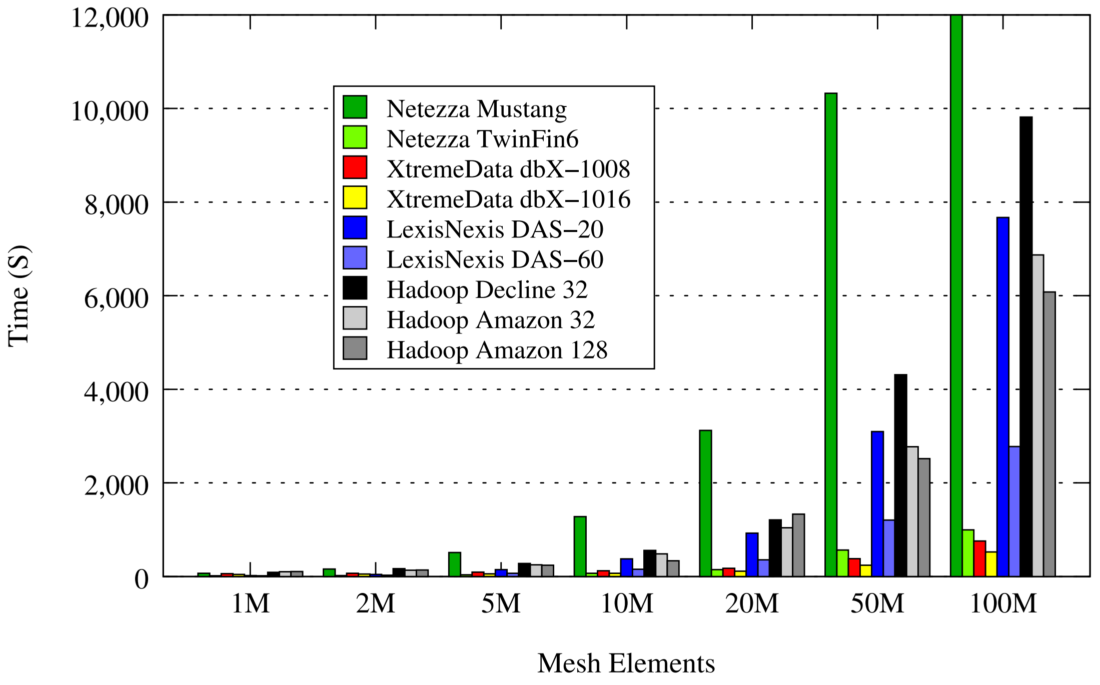

Figure 6.4. Element-pairing performance for different DWAs. For the 100M experiment, the Netezza Mustang finished in 26,605 s.

Figure 6.4 depicts the results of the element pairing experiments for the different DWAs. While Netezza's Mustang architecture performed the worst on all tests, its TwinFin6 platform consistently achieved good performance. XtremeData's dbX platform achieved the best performance in all test cases larger than 20M. LexisNexis DAS was competitive and demonstrated good scaling properties. Finally, while Hadoop performance did improve when cluster size was increased from 32 nodes to 128, the overall Hadoop timings were slower than those for the other DWAs. Part of this loss can be attributed to the out-of-band communication that the implementation had to do to overcome the limitations of the programming interface.

Chapter 7

Observations

This work has provided many insights into utilizing DWAs for scientific dataset analysis. The following are observations that originate from this effort.

Start Costs

All of the DWAs we investigated have relatively high startup costs for performing operations. While the database systems and LexisNexis can process trivial queries in less than a second, Hadoop has a startup cost of 10-20 seconds. However, these overheads emphasize the importance of performing complex computations on the DWAs when possible.

Floating-Point Limitations

The PowerPC processors in the older Netezza Mustang system do not provide hardware support for floating-point computations and therefore must perform the calculations in software. This issue is problematic for scientific datasets, which are largely comprised of floating-point data. We conducted an experiment replacing floating-point data with integer values on the Mustang system and found the integer version completed in a third of the time of the floating-point version. This observation raises issues for the large scale, embedded nodes design style. All of the other DWAs (including Netezza TwinFin) utilize x86 processors and do not have floating-point issues.

Tunability

In order to improve performance, we went through several development iterations on each platform. On the Netezza and XtremeData systems, the major complaint was that there were relatively few means by which we could refactor the algorithm. Most SQL-based DWAs are designed to do optimization for the user, and therefore do not provide many programming paths that a user can explore to improve performance. In contrast, Hadoop provides the opposite environment: users can easily be overwhelmed by the variety of ways they can refactor their algorithms to MapReduce operations. Similarly, Lexis's environment offers a rich suite of algorithmic primitives, which allow for many different implementation choices. As with Hadoop, this flexibility can result in a steeper learning curve.

Programming Interfaces

An ongoing question for our work has been whether or not SQL, MapReduce, and ECL are sufficient data-parallel languages for implementing nontrivial scientific analysis functions. The two adaptations presented in this paper confirm that basic algorithms can be adapted to these languages and executed in a parallel environment. However, it is important to note that the majority of our effort involved finding ways to get around language limitations. For our Netezza Mustang SQL implementations, we feel that performance was compromised in the final phase of the element pairings because there was no obvious way to do an all-to-all computation efficiently. The performance of this phase was greatly improved on the other SQL-based platforms, but this positive result was somewhat at the mercy of the query optimizers provided by each DWA. Hadoop's MapReduce provides excellent low-level control, but we continuously struggle with the problem of merging two data streams when random access is required. Similar to other developers, we solve this problem by going outside of the MapReduce paradigm. In contrast to the limiting simplicity of MapReduce, LexisNexis's powerful and flexible suite of primitives and predefined functions often obscures how a particular algorithm will map onto the underlying platform. As a result, we needed to work with experienced LexisNexis designers to achieve highly optimized performance.

Debugging

Debugging is challenging on any parallel platform. For Netezza and XtremeData we found SQL debugging to be relatively straight forward, due to the robustness of the SQL standard. However, UDFs must be developed with caution as programming errors can freeze a node or crash the system. Hadoop provides a number of facilities for debugging and automatically generates detailed statistics about jobs through a web interface. Hadoop also allows users to debug a job on a local machine before it is run on a cluster. Similarly, LexisNexis's interactive development environment and web-based debugging and profiling tools were invaluable.

Portability

Ideally, scientific users need to be able to easily move analysis programs from one platform to another without drastic changes. Developing in SQL is appealing in that sense because SQL-compliant code should be portable to different databases. We initially prototyped our SQL algorithms on a MySQL database and then changed the program to use the Netezza or XtremeData databases. While transitions between databases were relatively smooth, we did notice that some operations performed better in one database than another. Database vendors may also provide proprietary commands that have a huge impact on performance (e.g., Netezza's "Generate Statistics" command), but make code less portable. Similarly, UDFs written for one DWA are generally not usable on another platform. Hadoop on the other hand can run on a wide variety of platforms, as evidenced by our experience on Amazon's EC2. As discussed in the tunability observations, the challenge is often refactoring programs to maximize cluster resources. Although similar to SQL in some ways, the proprietary nature of LexisNexis's ECL results in limited portability.

Fault Tolerance

An important consideration in the design of many DWAs is resilience to inevitable hardware failures. At a minimum, data must be protected from loss and ideally analysis work can continue in the face of failures. Hadoop provides excellent fault tolerance, as each component in the framework is designed with the assumption that any other component could fail at any time. Our testing benefited from the software's automatic recovery mechanisms when nodes failed and HDFS's triple redundancy when hard disks overheated. Netezza's database products also exhibit excellent fault tolerance. Netezza system software replicates data among nodes and automatically reconstructs data on replacement nodes when a blade fails. Netezza's switch to commodity hardware is welcomed, as it allows incremental replacement, as opposed to swapping out a full board. While we did not observe any hardware failures with LexisNexis, we note that its system software automatically replicates data in the cluster to improve reliability. Finally, XtremeData's hardware currently utilizes RAID controllers to handle reliability at the device level.

Chapter 8

Conclusions

Scientific applications are generating massive datasets that are difficult to analyze utilizing traditional, offline approaches. An emerging class of systems known as Data Warehouse Appliances provides an opportunity to improve the scale at which automatic data analysis operations can be performed through the use of parallel storage hardware and data-parallel programming interfaces. In this paper we have explored how two scientific data analysis algorithms can be adapted to five different DWA architectures. We have confirmed that the data-parallel programming interfaces for these platforms are sufficient for implementing our out-of-core algorithms, and that the parallel hardware of these systems could be utilized. However, it is important to note that the APIs for these platforms required a good bit of planning and experimentation in order to achieve good parallel performance.

Our inspection of leading-edge DWAs has found that while no system is the clear winner in all cases, each system excels in its own focus area. Netezza's aging Mustang architecture yields good performance in brute-force, integer applications and has scaled well in other work [10]. The Netezza TwinFin architecture provides a vast improvement over Mustang and performs well in nearly all our tests. The XtremeData dbX platform provided the best performance in the most challenging experiment. LexisNexis performed at levels in between the SQL and Hadoop systems and offered the most flexible means of specifying dataflow computations. Other researchers have reported that LexisNexis offers exceptional speedups in graph analysis algorithms [19]. Finally, Hadoop clusters can deliver excellent performance in brute-force applications, provide high-levels of fault tolerance, and are highly accessible to researchers.

Appendix A

Language Examples

This investigation required us to adapt algorithms to different programming languages. In this appendix we provide code fragments from the different DWA implementations. Our intention with these listings is not to provide fully functional implementations that can be run on each DWA. Instead, the listings are presented to provide a high-level programmer's view of the different languages.

A.1 Structured Query Language (SQL)

The Structured Query Language (SQL) is a well-known language for performing operations on large data stores. The following code listings are an implementation of the 6x4-join version of the threshold volume calculation. As described in Section 5.2, this algorithm performs computations on a per-tetrahedral basis and assembles vertex data when evaluating each tetrahedron.

The first step in the algorithm is to create two temporary tables that will be used for housing intermediate data in the calculation. The first table contains the absolute coordinates for each vertex in the timestep that is above the specified threshold value. The second table provides a place to store the volumes of relevant tetrahedra.

```
--create temporary table for values of interest
CREATE TEMPORARY TABLE TEMPVALSNODE ( nodeID INTEGER NOT NULL,
                                      x DOUBLE NOT NULL ,
                                      y DOUBLE NOT NULL ,
                                      z DOUBLE NOT NULL );

INSERT INTO TEMPVALSNODE SELECT v.nodeID,   n.x+v.x_displacement,n.y+v.y_displacement,n.z+v.z_displacement
        FROM  ValsNode v INNER JOIN NodeId n ON n.nodeID=v.NodeID
        WHERE v.timestep=0 AND v.val1>6;


CREATE TEMPORARY TABLE vols ( vol double not NULL);
```

The second step in the algorithm is to compute the individual volumes for each of the six tetrahedra that make up each element. This operation utilizes inner joins to assemble the coordinates of each tetrahedron vertex and performs the volume calculation inline. Once the volume information has been assembled properly, the total volume can be computed through as standard sum operation.

```
--begin 6X4 joins
--(can also implement as subquery instead of temporary table but not faster in this case to do so)


INSERT INTO vols SELECT
        SUM( ABS( ((v1.x)-(v2.x)) * ((v3.y)-(v2.y)) * ((v4.z)-(v2.z))
            +   ((v1.y)-(v2.y)) * ((v3.z)-(v2.z)) * ((v4.x)-(v2.x))
            +   ((v1.z)-(v2.z)) * ((v3.x)-(v2.x)) * ((v4.y)-(v2.y))
            -   ((v1.z)-(v2.z)) * ((v3.y)-(v2.y)) * ((v4.x)-(v2.x))
            -   ((v1.x)-(v2.x)) * ((v3.z)-(v2.z)) * ((v4.y)-(v2.y))
            -   ((v1.y)-(v2.y)) * ((v3.x)-(v2.x))*  ((v4.z)-(v2.z)) ) )/6.0 as vol
         FROM  ElementID e INNER JOIN tempValsNode v1 ON e.node1 = v1.nodeID
                INNER JOIN tempValsNode v2 ON e.node0 = v2.nodeID
                INNER JOIN tempValsNode v3 ON e.node3 = v3.nodeID
                INNER JOIN tempValsNode v4 ON e.node4 = v4.nodeID;
INSERT INTO vols  SELECT
         SUM( ABS( ((v1.x)-(v2.x)) * ((v3.y)-(v2.y)) * ((v4.z)-(v2.z))
            +   ((v1.y)-(v2.y)) * ((v3.z)-(v2.z)) * ((v4.x)-(v2.x))
            +   ((v1.z)-(v2.z)) * ((v3.x)-(v2.x)) * ((v4.y)-(v2.y))
            -   ((v1.z)-(v2.z)) * ((v3.y)-(v2.y)) * ((v4.x)-(v2.x))
            -   ((v1.x)-(v2.x)) * ((v3.z)-(v2.z)) * ((v4.y)-(v2.y))
            -   ((v1.y)-(v2.y)) * ((v3.x)-(v2.x))*  ((v4.z)-(v2.z)) ) )/6.0 as vol
        FROM  ElementID e INNER JOIN tempValsNode v1 ON e.node7 = v1.nodeID
              INNER JOIN  tempValsNode v2 ON e.node6 = v2.nodeID
              INNER JOIN tempValsNode v3 ON e.node5 = v3.nodeID
              INNER JOIN tempValsNode v4 ON e.node2 = v4.nodeID ;
INSERT INTO vols  SELECT
        SUM( ABS( ((v1.x)-(v2.x)) * ((v3.y)-(v2.y)) * ((v4.z)-(v2.z))
            +   ((v1.y)-(v2.y)) * ((v3.z)-(v2.z)) * ((v4.x)-(v2.x))
            +   ((v1.z)-(v2.z)) * ((v3.x)-(v2.x)) * ((v4.y)-(v2.y))
            -   ((v1.z)-(v2.z)) * ((v3.y)-(v2.y)) * ((v4.x)-(v2.x))
            -   ((v1.x)-(v2.x)) * ((v3.z)-(v2.z)) * ((v4.y)-(v2.y))
            -   ((v1.y)-(v2.y)) * ((v3.x)-(v2.x))*  ((v4.z)-(v2.z)) ) )/6.0 as vol
        FROM  ElementID e INNER JOIN tempValsNode v1 ON e.node3 = v1.nodeID
              INNER JOIN  tempValsNode v2 ON e.node1 = v2.nodeID
              INNER JOIN tempValsNode v3 ON e.node2 = v3.nodeID
              INNER JOIN tempValsNode v4 ON e.node4 = v4.nodeID ;
INSERT INTO vols   SELECT
        SUM( ABS( ((v1.x)-(v2.x)) * ((v3.y)-(v2.y)) * ((v4.z)-(v2.z))
            +   ((v1.y)-(v2.y)) * ((v3.z)-(v2.z)) * ((v4.x)-(v2.x))
            +   ((v1.z)-(v2.z)) * ((v3.x)-(v2.x)) * ((v4.y)-(v2.y))
            -   ((v1.z)-(v2.z)) * ((v3.y)-(v2.y)) * ((v4.x)-(v2.x))
            -   ((v1.x)-(v2.x)) * ((v3.z)-(v2.z)) * ((v4.y)-(v2.y))
            -   ((v1.y)-(v2.y)) * ((v3.x)-(v2.x))*  ((v4.z)-(v2.z)) ) )/6.0 as vol
        FROM  ElementID e INNER JOIN tempValsNode v1 ON e.node1 = v1.nodeID
              INNER JOIN  tempValsNode v2 ON e.node2 = v2.nodeID
              INNER JOIN tempValsNode v3 ON e.node5 = v3.nodeID
              INNER JOIN tempValsNode v4 ON e.node4 = v4.nodeID ;
INSERT INTO vols     SELECT
              SUM( ABS( ((v1.x)-(v2.x)) * ((v3.y)-(v2.y)) * ((v4.z)-(v2.z))
            +   ((v1.y)-(v2.y)) * ((v3.z)-(v2.z)) * ((v4.x)-(v2.x))
            +   ((v1.z)-(v2.z)) * ((v3.x)-(v2.x)) * ((v4.y)-(v2.y))
            -   ((v1.z)-(v2.z)) * ((v3.y)-(v2.y)) * ((v4.x)-(v2.x))
            -   ((v1.x)-(v2.x)) * ((v3.z)-(v2.z)) * ((v4.y)-(v2.y))
            -   ((v1.y)-(v2.y)) * ((v3.x)-(v2.x))*  ((v4.z)-(v2.z)) ) )/6.0 as vol
        FROM  ElementID e INNER JOIN tempValsNode v1 ON e.node2 = v1.nodeID
              INNER JOIN  tempValsNode v2 ON e.node3 = v2.nodeID
              INNER JOIN tempValsNode v3 ON e.node7 = v3.nodeID
              INNER JOIN tempValsNode v4 ON e.node4 = v4.nodeID ;
INSERT INTO vols   SELECT
              SUM( ABS( ((v1.x)-(v2.x)) * ((v3.y)-(v2.y)) * ((v4.z)-(v2.z))
            +   ((v1.y)-(v2.y)) * ((v3.z)-(v2.z)) * ((v4.x)-(v2.x))
            +   ((v1.z)-(v2.z)) * ((v3.x)-(v2.x)) * ((v4.y)-(v2.y))
            -   ((v1.z)-(v2.z)) * ((v3.y)-(v2.y)) * ((v4.x)-(v2.x))
            -   ((v1.x)-(v2.x)) * ((v3.z)-(v2.z)) * ((v4.y)-(v2.y))
            -   ((v1.y)-(v2.y)) * ((v3.x)-(v2.x))*  ((v4.z)-(v2.z)) ) )/6.0 as vol
        FROM  ElementID e INNER JOIN tempValsNode v1 ON e.node2 = v1.nodeID
              INNER JOIN  tempValsNode v2 ON e.node5 = v2.nodeID
              INNER JOIN tempValsNode v3 ON e.node7 = v3.nodeID
              INNER JOIN tempValsNode v4 ON e.node4 = v4.nodeID;


SELECT sum(vol) from vols;
```

A.2 Enterprise Control Language (ECL)

For the LexisNexis, algorithms are defined in the Enterprise Control Language (ECL). This language mixes concepts from SQL with formatting from other structured languages. The first step in the ECL implementation of the threshold volume calculation is to distribute data among different nodes in the cluster to allow computations to be performed in a distributed manner.

```
NodeIdFileTemp := PROJECT(NodeIdInputFile,
                          TRANSFORM(NodeIdRecTemp,
                                    SELF.elementId := (UNSIGNED3)TRUNCATE((REAL)LEFT.NodeId/8);
                                    SELF := LEFT));

NodeIdFile := DISTRIBUTE(NodeIdFileTemp, elementId)
: PERSIST('distVal100M');
```

The second step in this algorithm is to remove data values from the input that are below a specified threshold value.

```
ThreshValsNodeFile := DISTRIBUTE( ValsNodeFile(vel_x  > 6),
                                  ((UNSIGNED3)TRUNCATE((REAL)NodeId/8)) )
: PERSIST('distThr100M');
```

Next, the algorithm updates node coordinate data for a particular frame from relative to absolute positioning.

```
NodeIdRec3 Xform(NodeIdRecTemp L, ThreshValsNodeFile R) := TRANSFORM
  SELF.meetsThresh := IF(R.vel_x != 0, TRUE, FALSE);
  SELF.X := L.X + R.x_dis;
  SELF.Y := L.Y + R.y_dis;
  SELF.Z := L.Z + R.z_dis;
  SELF := L;
END;
```

Element and node entries are then merged together using a JOIN operation.

```
NodeIdFileMerged := JOIN( NodeIdFile,
                          ThreshValsNodeFile,
                          LEFT.NodeId = RIGHT.NodeId,
                          Xform(LEFT, RIGHT),
                          LEFT OUTER,
                          LOCAL)
: PERSIST('join100M');
```

An element's node data values are collected into a single record for a given timestep using the ROLLUP command.

```
NodeIdFileDu := PROJECT(NodeIdFileMerged,
                        TRANSFORM( concatRec,
                                   SELF.elementId := LEFT.elementId;
                                   SELF.nodes := LEFT),
                        LOCAL);

ElementDs1 := ROLLUP(NodeIdFileDu, LEFT.elementId = RIGHT.elementId,
TRANSFORM(concatRec, SELF.nodes := LEFT.nodes + RIGHT.nodes; SELF := LEFT), LOCAL)
: PERSIST('test100M');
```

Finally, the six tetrahedra volumes are computed for each element through a PROJECT operation and then combined using a SUM operation. The results are passed designated through the use of an OUTPUT operation.

```
Vols calcVols(concatRec L) := TRANSFORM
a1 := IF( L.nodes[1].meetsThresh AND L.nodes[2].meetsThresh AND
          L.nodes[4].meetsThresh AND L.nodes[5].meetsThresh
        , ABS((L.nodes[2].X-L.nodes[1].X) * (L.nodes[4].Y-L.nodes[1].Y) * (L.nodes[5].Z-L.nodes[1].Z)+
              (L.nodes[2].Y-L.nodes[1].Y) * (L.nodes[4].Z-L.nodes[1].Z) * (L.nodes[5].X-L.nodes[1].X)+
              (L.nodes[2].Z-L.nodes[1].Z) * (L.nodes[4].X-L.nodes[1].X) * (L.nodes[5].Y-L.nodes[1].Y)-
              (L.nodes[2].Z-L.nodes[1].Z) * (L.nodes[4].Y-L.nodes[1].Y) * (L.nodes[5].X-L.nodes[1].X)-
              (L.nodes[2].X-L.nodes[1].X) * (L.nodes[4].Z-L.nodes[1].Z) * (L.nodes[5].Y-L.nodes[1].Y)-
              (L.nodes[2].Y-L.nodes[1].Y) * (L.nodes[4].X-L.nodes[1].X) * (L.nodes[5].Z-L.nodes[1].Z)  )
        , 0);

a2 := IF( L.nodes[3].meetsThresh AND L.nodes[6].meetsThresh AND
          L.nodes[7].meetsThresh AND L.nodes[8].meetsThresh
        , ABS((L.nodes[8].X-L.nodes[7].X) * (L.nodes[6].Y-L.nodes[7].Y) * (L.nodes[3].Z-L.nodes[7].Z)+
              (L.nodes[8].Y-L.nodes[7].Y) * (L.nodes[6].Z-L.nodes[7].Z) * (L.nodes[3].X-L.nodes[7].X)+
              (L.nodes[8].Z-L.nodes[7].Z) * (L.nodes[6].X-L.nodes[7].X) * (L.nodes[3].Y-L.nodes[7].Y)-
              (L.nodes[8].Z-L.nodes[7].Z) * (L.nodes[6].Y-L.nodes[7].Y) * (L.nodes[3].X-L.nodes[7].X)-
              (L.nodes[8].X-L.nodes[7].X) * (L.nodes[6].Z-L.nodes[7].Z) * (L.nodes[3].Y-L.nodes[7].Y)-
              (L.nodes[8].Y-L.nodes[7].Y) * (L.nodes[6].X-L.nodes[7].X) * (L.nodes[3].Z-L.nodes[7].Z)  )
        , 0);

a3 := IF( L.nodes[2].meetsThresh AND L.nodes[3].meetsThresh AND
          L.nodes[4].meetsThresh AND L.nodes[5].meetsThresh
        , ABS((L.nodes[4].X-L.nodes[2].X) * (L.nodes[3].Y-L.nodes[2].Y) * (L.nodes[5].Z-L.nodes[2].Z)+
              (L.nodes[4].Y-L.nodes[2].Y) * (L.nodes[3].Z-L.nodes[2].Z) * (L.nodes[5].X-L.nodes[2].X)+
              (L.nodes[4].Z-L.nodes[2].Z) * (L.nodes[3].X-L.nodes[2].X) * (L.nodes[5].Y-L.nodes[2].Y)-
              (L.nodes[4].Z-L.nodes[2].Z) * (L.nodes[3].Y-L.nodes[2].Y) * (L.nodes[5].X-L.nodes[2].X)-
              (L.nodes[4].X-L.nodes[2].X) * (L.nodes[3].Z-L.nodes[2].Z) * (L.nodes[5].Y-L.nodes[2].Y)-
              (L.nodes[4].Y-L.nodes[2].Y) * (L.nodes[3].X-L.nodes[2].X) * (L.nodes[5].Z-L.nodes[2].Z)  )
        , 0);

a4 := IF( L.nodes[2].meetsThresh AND L.nodes[3].meetsThresh AND
          L.nodes[5].meetsThresh AND L.nodes[6].meetsThresh
        , ABS((L.nodes[2].X-L.nodes[3].X) * (L.nodes[6].Y-L.nodes[3].Y) * (L.nodes[5].Z-L.nodes[3].Z)+
              (L.nodes[2].Y-L.nodes[3].Y) * (L.nodes[6].Z-L.nodes[3].Z) * (L.nodes[5].X-L.nodes[3].X)+
              (L.nodes[2].Z-L.nodes[3].Z) * (L.nodes[6].X-L.nodes[3].X) * (L.nodes[5].Y-L.nodes[3].Y)-
              (L.nodes[2].Z-L.nodes[3].Z) * (L.nodes[6].Y-L.nodes[3].Y) * (L.nodes[5].X-L.nodes[3].X)-
              (L.nodes[2].X-L.nodes[3].X) * (L.nodes[6].Z-L.nodes[3].Z) * (L.nodes[5].Y-L.nodes[3].Y)-
              (L.nodes[2].Y-L.nodes[3].Y) * (L.nodes[6].X-L.nodes[3].X) * (L.nodes[5].Z-L.nodes[3].Z)  )
        , 0);

a5 := IF( L.nodes[3].meetsThresh AND L.nodes[4].meetsThresh AND
          L.nodes[5].meetsThresh AND L.nodes[8].meetsThresh
        , ABS((L.nodes[3].X-L.nodes[4].X) * (L.nodes[8].Y-L.nodes[4].Y) * (L.nodes[5].Z-L.nodes[4].Z)+
              (L.nodes[3].Y-L.nodes[4].Y) * (L.nodes[8].Z-L.nodes[4].Z) * (L.nodes[5].X-L.nodes[4].X)+
              (L.nodes[3].Z-L.nodes[4].Z) * (L.nodes[8].X-L.nodes[4].X) * (L.nodes[5].Y-L.nodes[4].Y)-
              (L.nodes[3].Z-L.nodes[4].Z) * (L.nodes[8].Y-L.nodes[4].Y) * (L.nodes[5].X-L.nodes[4].X)-
              (L.nodes[3].X-L.nodes[4].X) * (L.nodes[8].Z-L.nodes[4].Z) * (L.nodes[5].Y-L.nodes[4].Y)-
              (L.nodes[3].Y-L.nodes[4].Y) * (L.nodes[8].X-L.nodes[4].X) * (L.nodes[5].Z-L.nodes[4].Z)  )
        , 0);

a6 := IF( L.nodes[3].meetsThresh AND L.nodes[5].meetsThresh AND
          L.nodes[6].meetsThresh AND L.nodes[8].meetsThresh
        , ABS((L.nodes[3].X-L.nodes[6].X) * (L.nodes[8].Y-L.nodes[6].Y) * (L.nodes[5].Z-L.nodes[6].Z)+
              (L.nodes[3].Y-L.nodes[6].Y) * (L.nodes[8].Z-L.nodes[6].Z) * (L.nodes[5].X-L.nodes[6].X)+
              (L.nodes[3].Z-L.nodes[6].Z) * (L.nodes[8].X-L.nodes[6].X) * (L.nodes[5].Y-L.nodes[6].Y)-
              (L.nodes[3].Z-L.nodes[6].Z) * (L.nodes[8].Y-L.nodes[6].Y) * (L.nodes[5].X-L.nodes[6].X)-
              (L.nodes[3].X-L.nodes[6].X) * (L.nodes[8].Z-L.nodes[6].Z) * (L.nodes[5].Y-L.nodes[6].Y)-
              (L.nodes[3].Y-L.nodes[6].Y) * (L.nodes[8].X-L.nodes[6].X) * (L.nodes[5].Z-L.nodes[6].Z)  )
        , 0);

ac := (a1 + a2 + a3 + a4 + a5 + a6) / 6.0;
SELF.v := ac;
END;

volumes := PROJECT(ElementDs1, calcVols(LEFT));
result := sum(volumes, v);
OUTPUT(result);
```

A.3 Hadoop Java MapReduce

The threshold volume calculation can be implemented in Hadoop's MapReduce framework in a straightforward manner. At the top level, a Java class named MeshVolumeNodeThreashold is comprised of three components: (1) a mapper class named VolumeMap for computing the volume of relevant elements, (2) a simple reducer class named VolumeReduce for merging results, and (3) Hadoop configuration methods for specifying how the work should be completed. The following listing provides a top-level view of the application.

```java
package gov.sandia.sicaida.hadoop.mapreduce.volume;

import gov.sandia.sicaida.hadoop.util.DoubleWritableTwoDArrayWritable;

import java.io.IOException;
import java.util.ArrayList;
import java.util.Iterator;
import java.util.List;

import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.conf.Configured;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.DoubleWritable;
import org.apache.hadoop.io.LongWritable;
import org.apache.hadoop.io.Writable;
import org.apache.hadoop.mapred.Counters;
import org.apache.hadoop.mapred.FileInputFormat;
import org.apache.hadoop.mapred.FileOutputFormat;
import org.apache.hadoop.mapred.JobClient;
import org.apache.hadoop.mapred.JobConf;
import org.apache.hadoop.mapred.MapReduceBase;
import org.apache.hadoop.mapred.Mapper;
import org.apache.hadoop.mapred.OutputCollector;
import org.apache.hadoop.mapred.Reducer;
import org.apache.hadoop.mapred.Reporter;
import org.apache.hadoop.mapred.RunningJob;
import org.apache.hadoop.mapred.TextOutputFormat;
import org.apache.hadoop.util.Tool;
import org.apache.hadoop.util.ToolRunner;

public class MeshVolumeNodeThreshold extends Configured implements Tool
{
  public static class VolumeMap extends MapReduceBase implements
        Mapper<LongWritable, DoubleWritableTwoDArrayWritable, LongWritable, DoubleWritable>
  {

      public enum Mesh {  TetsUsed  };
      private LongWritable allTheSameKey = new LongWritable((long) 0);
      private int          nodeThreshold;

      //Methods
      public  void    map(...)                          { ... }
      public  void    configure(...)                    { ... }
      public  boolean checkThreshold(...)               { ... }
      private double  tetVolume(...)                    { ... }
  }

  public static class VolumeReduce extends MapReduceBase implements
        Reducer<LongWritable, DoubleWritable, LongWritable, DoubleWritable>
  {
      private double totalVolume = 0;

      //Methods
          public void reduce(...)   { ... }
  }

  public int          run(...)             throws Exception  { ... }
  public JobConf      confMeshVolume(...)                    { ... }
  public static void  main(...)            throws Exception  { ... }
}
```

The mapper task performs the bulk of the work in this application. After extracting the element's eight xyz coordinates, the mapper examines each of the six internal tetrahedra that form the element. If a tetrahedron's data values are all greater than the threshold, the volume of the tetrahedron is added to the tally.

```
// VolumeMap::map()
public void map(LongWritable key, DoubleWritableTwoDArrayWritable value,
                OutputCollector<LongWritable, DoubleWritable> output,
                Reporter reporter) throws IOException
{
  Writable[][] coord = value.get();
  double Ax = ((DoubleWritable) coord[0][0]).get();
  double Bx = ((DoubleWritable) coord[1][0]).get();
  double Cx = ((DoubleWritable) coord[2][0]).get();
  double Dx = ((DoubleWritable) coord[3][0]).get();
  double Ex = ((DoubleWritable) coord[4][0]).get();
  double Fx = ((DoubleWritable) coord[5][0]).get();
  double Gx = ((DoubleWritable) coord[6][0]).get();
  double Hx = ((DoubleWritable) coord[7][0]).get();

  double Ay = ((DoubleWritable) coord[0][1]).get();
  double By = ((DoubleWritable) coord[1][1]).get();
  double Cy = ((DoubleWritable) coord[2][1]).get();
  double Dy = ((DoubleWritable) coord[3][1]).get();
  double Ey = ((DoubleWritable) coord[4][1]).get();
  double Fy = ((DoubleWritable) coord[5][1]).get();
  double Gy = ((DoubleWritable) coord[6][1]).get();
  double Hy = ((DoubleWritable) coord[7][1]).get();


  double Az = ((DoubleWritable) coord[0][2]).get();
  double Bz = ((DoubleWritable) coord[1][2]).get();
  double Cz = ((DoubleWritable) coord[2][2]).get();
  double Dz = ((DoubleWritable) coord[3][2]).get();
  double Ez = ((DoubleWritable) coord[4][2]).get();
  double Fz = ((DoubleWritable) coord[5][2]).get();
  double Gz = ((DoubleWritable) coord[6][2]).get();
  double Hz = ((DoubleWritable) coord[7][2]).get();

  double tetVector[] = { 0, 0, 0, 0, 0, 0 };

  if (checkThreshold(reporter, coord, 0, 1, 3, 4))
    tetVector[0] = Math.abs(tetVolume(Bx - Ax, By - Ay, Bz - Az,
                                      Dx - Ax, Dy - Ay, Dz - Az,
                                      Ex - Ax, Ey - Ay, Ez - Az));
  if (checkThreshold(reporter, coord, 2, 5, 6, 7))
    tetVector[1] = Math.abs(tetVolume(Hx - Gx, Hy - Gy, Hz - Gz,
                                      Fx - Gx, Fy - Gy, Fz - Gz,
                                      Cx - Gx, Cy - Gy, Cz - Gz));
  if (checkThreshold(reporter, coord, 1, 2, 3, 4))
    tetVector[2] = Math.abs(tetVolume(Dx - Cx, Dy - Cy, Dz - Cz,
                                      Bx - Cx, By - Cy, Bz - Cz,
                                      Ex - Cx, Ey - Cy, Ez - Cz));
  if (checkThreshold(reporter, coord, 1, 2, 4, 5))
    tetVector[3] = Math.abs(tetVolume(Fx - Cx, Fy - Cy, Fz - Cz,
                                      Bx - Cx, By - Cy, Bz - Cz,
                                      Ex - Cx, Ey - Cy, Ez - Cz));
  if (checkThreshold(reporter, coord, 2, 3, 4, 7))
    tetVector[4] = Math.abs(tetVolume(Dx - Cx, Dy - Cy, Dz - Cz,
                                      Hx - Cx, Hy - Cy, Hz - Cz,
                                      Ex - Cx, Ey - Cy, Ez - Cz));
  if (checkThreshold(reporter, coord, 2, 4, 5, 7))
    tetVector[5] = Math.abs(tetVolume(Fx - Cx, Fy - Cy, Fz - Cz,
                                      Hx - Cx, Hy - Cy, Hz - Cz,
                                      Ex - Cx, Ey - Cy, Ez - Cz));

  double elementVolume = 0.0;
  for (double tetVolume : tetVector)
    elementVolume += tetVolume;


  if (elementVolume > 0)
    output.collect(allTheSameKey, new DoubleWritable(elementVolume));
}
```

The configure method provides a way to pass job parameters among components in the framework. In threshold volume application, the only parameter that needs to be passed to the mapper is the cutoff threshold value.

```
// VolumeMap::configure()
public void configure(JobConf job)
{
  // Get threshold
  String nodeThresholdString = job.get("nodeThreshold");
  this.nodeThreshold = Integer.parseInt(nodeThresholdString);
}
```

The checkThreshold() method determines whether a tetrahedron's four data values are all above the threshold.

```
// VolumeMap::checkThreshold()
private boolean checkThreshold(Reporter reporter, Writable[][] nodeData, int... nodesToCheck)
{
  for (int i : nodesToCheck)
  {
    if (((DoubleWritable) nodeData[i][4]).get() <= nodeThreshold)
    {
      return false;
    }
  }

  reporter.incrCounter(Mesh.TetsUsed, 1);

  return true;
}
```

The tetVolume() method computes the volume of a tetrahedron, given its three defining vectors.

```
// VolumeMap::tetVolume()
private double tetVolume(double x1, double y1, double z1,
                         double x2, double y2, double z2,
                         double x3, double y3, double z3)
{
  double volume = (1.0 / 6.0) *
                 ((x1 * y2 * z3 +  y1 * z2 * x3 + z1 * x2 * y3) -
                  (z1 * y2 * x3 +  x1 * z2 * y3 + y1 * x2 * z3)    );
  return volume;
}
```

The reducer for this application is simply the summation of all volumes that were generated by the mappers.

```
// VolumeReduce::reduce()
public void reduce(LongWritable key,
                   Iterator<DoubleWritable> values,
                   OutputCollector<LongWritable, DoubleWritable> output,
                   Reporter reporter) throws IOException
{
  while (values.hasNext())
  {
    totalVolume += values.next().get();
  }

  output.collect(key, new DoubleWritable(totalVolume));
}
```

The run() method for this application allows users to supply job paramters to the framework as well as specify the overall data flow of the job.

```
// MeshVolumeNodeThreshold::run()
public int run(String[] args) throws Exception
{
  StringBuilder summaryInfo = new StringBuilder();
  int numMapTasks = 32;
  int numReduceTasks = 1;

  List<String> other_args = new ArrayList<String>();
  for (int i = 0; i < args.length; ++i)
  {
    try
    {
      if ("-m".equals(args[i]))
      {
        numMapTasks = Integer.parseInt(args[++i]);
      } else if ("-r".equals(args[i]))
      {
        numReduceTasks = Integer.parseInt(args[++i]);
      } else
      {
        other_args.add(args[i]);
      }
    } catch (NumberFormatException except)
    {
      System.out.println("ERROR: Integer expected instead of " + args[i]);
      return printUsage();
    } catch (ArrayIndexOutOfBoundsException except)
    {
      System.out.println("ERROR: Required parameter missing from " + args[i - 1]);
      return printUsage();
    }
  }
  // Make sure there are exactly 2 parameters left.
  if (other_args.size() != 3)
  {
    System.out.println("ERROR: Wrong number of parameters: " + other_args.size() + " instead of 2.");
    return printUsage();
  }

  String staticDataInputPath = other_args.get(0);
  String variableDataInputPath = other_args.get(1);
  String outputPath = other_args.get(2);

  outputPath += System.currentTimeMillis() + "/";

  // Calc Volume Phase
  Long jobStartTime = System.currentTimeMillis();

  JobConf confMeshVolume = confMeshVolume(staticDataInputPath, variableDataInputPath,
                                          outputPath + "volume", numMapTasks, numReduceTasks);
  RunningJob job = JobClient.runJob(confMeshVolume);

  Long jobEndTime = System.currentTimeMillis();

  Counters counters = job.getCounters();
  long tetsUsed = counters.getCounter(MeshVolumeNodeThreshold.VolumeMap.Mesh.TetsUsed);

  summaryInfo.append("Number of tets included in volume: " + tetsUsed + "\n");
  summaryInfo.append("MeshVolume-NodeTheshold(" + variableDataInputPath + "): Time: "
                     + ((jobEndTime - jobStartTime) / 1000.0)   + " sec ("
                     + ((jobEndTime - jobStartTime) / 60000.0)  + " min)\n");
  System.out.println(summaryInfo);

  return 0;
}
```

Hadoop utilizes a job configuration object to pass static information to tasks. The confMeshVolume() method creates a JobConf object for this particular application.

```
// MeshVolumeNodeThreshold::confMeshVolume()
public JobConf confMeshVolume(String staticDataIntputPath, String variableDataIntputPath,
                              String outputPath, int numMaps, int numReduces)
{
  JobConf conf = new JobConf(MeshVolumeNodeThreshold.class);
  conf.setJobName("meshVolume-NodeTheshold");

  conf.set("valsNode", variableDataIntputPath + "ValsNode.csv.bin");

  conf.set("startingTimestep", "0");
  conf.set("endingTimestep", "0");
  conf.set("numNodeIds", "8388608");

  conf.set("nodeThreshold", "6");

  conf.setOutputKeyClass(LongWritable.class);
  conf.setOutputValueClass(DoubleWritable.class);

  conf.setMapperClass(VolumeMap.class);
  conf.setCombinerClass(VolumeReduce.class);
  conf.setReducerClass(VolumeReduce.class);

  conf.setNumMapTasks(numMaps);
  conf.setNumReduceTasks(numReduces);

  conf.setInputFormat(MeshDataPreJoinedStaticInputFormat.class);
  conf.setOutputFormat(TextOutputFormat.class);

  FileInputFormat.setInputPaths(conf, new Path(staticDataIntputPath));
  FileOutputFormat.setOutputPath(conf, new Path(outputPath));

  return conf;
}
```

The main() method for this application creates a new class for the application and passes the data to the framework.

```
public static void main(String[] args) throws Exception
{
  ToolRunner.run(new Configuration(), new MeshVolumeNodeThreshold(), args);
}
```

The printUsage() method displays the parameters for launching a job.

```
static int printUsage()
{
  System.out.println("MeshVolumeNodeThreshold␣" +
                     "[-m␣<maps>]␣[-r␣<reduces>]␣" +
                     "<static␣input>␣<dynamic␣input>␣<output>");
  return -1;
}
```

References

[1] Amazon elastic compute cloud (ec2). `http://www.amazon.com/ec2/`, June 2002.

[2] Dbx powered by XtremeData. Product Brief, 2009.

[3] Lexisnexis data analytics supercomputer. LNSSI White Paper, 2009.

[4] K. Alvin. ASC national code strategy simulation-based complex transformation, 2009.

[5] David G. Andersen, Jason Franklin, Michael Kaminsky, Amar Phanishayee, Lawrence Tan, and Vijay Vasudevan. Fawn: a fast array of wimpy nodes. In Jeanna Neefe Matthews and Thomas E. Anderson, editors, *SOSP*, pages 1–14. ACM, 2009.

[6] A. Bialecki, M. Cafarella, D. Cutting, and O. OMalley. Hadoop: A framework for running applications on large clusters built of commodity hardware. `http://lucene.apache.org/hadoop`, June 2009.

[7] D. Borthakur. The hadoop distributed file system: Architecture and design. `http://lucene.apache.org/`, June 2009.

[8] P. Braam. The lustre storage architecture. `http://www.lustre.org`, 2002.

[9] A. Cedilnik, B. Geveci, K. Moreland, J. Ahrens, and J. Favre. Remote large data visualization in the paraview framework, May 2006.

[10] G. Davidson, K. Boyack, R. Zacharski, S. Helmreich, and J. Cowie. Data-centric computing with the netezza architecture, April 2006.

[11] J. Dean and S. Ghemawat. Mapreduce: Simplified data processing on large clusters, 2004.

[12] S. Ghemawat, H. Gobioff, and S. Leung. The google file system, 2003.

[13] Yunhong Gu and Robert Grossman. Sector and sphere: The design and implementation of a high performance data cloud, June 2009.

[14] J. Laros, L. Ward, R. Klundt, S. Kelly, J. Tomkins, and B. Kellogg. Red storm io performance analysis, September 2007.

[15] W. Lorensen and H. Cline. Marching cubes: A high resolution 3d surface construction algorithm, 1987.

[16] R. Meisner. A platform strategy for the advanced simulation and computing program, 2007.

[17] Roger Rea and Krishna Mamidipaka. Ibm infosphere streams: Enabling complex analytics with ultra-low latencies on data in motion. IBM White Paper, 2009.

[18] W. Tantisiriroj, S. Patil, and G. Gibson. Data-intensive file systems for internet services: A rose by any other name , October 2008.

[19] Andy Yoo and Ian Kaplan. Evaluating use of data flow systems for large graph analysis. In *Proceedings of the 2nd Workshop on Many-Task Computing on Grids and Supercomputers*, 2009.

[20] O. Zienkiewicz, R. Taylor, and J. Zhu. *The finite element method: its basis and fundamentals; 6th ed.* Elsevier, Amsterdam, 2005.

Sandia National Laboratories