A Scalable Architecture for Simplifying Full-Range Scientific Data Analysis

A Thesis Presented for

The Doctor of Philosophy

Degree

The University of Tennessee, Knoxville

Wesley James Kendall

December 2011

For my grandmother

Acknowledgements

There are many people that have been pivotal in my life, and without them I may have never had the opportunity to pursue a Ph.D. One person in this regard is my advisor, Jian Huang, who inspired me as an undergraduate to pursue Graduate School. Along with helping fund my graduate studies, Jian has helped me grow as a critical thinker in ways that I could have never imagined.

I have met many influential people along the way who have helped me in my pursuits of knowledge. Tom Peterka and Rob Ross are two of these people that have been excellent mentors. The two summers I spent at Argonne National Laboratory with Tom, Rob, and even for a short time with Han-Wei Shen, provided the basis for almost all of the research present in my dissertation. Along with them, my summer at Google spent with Jelena Pjesivac-Grbovic was also key to motivating much of my work.

Along with my main mentors, there are several other colleagues who have shaped my experience as a student. Forrest Hoffman, Richard Mills, David Erickson, and Melissa Allen have been central collaborators in many of the application examples throughout this work. I also could not have grown as a student without the help of other students, namely Markus Glatter, Rob Sisneros, Joshua New, and Jingyuan Wang.

This work was supported by funding provided through the Institute of Ultra-Scale Visualization (www.ultravis.org) under the auspices of the Scientific Discovery through Advanced Computing (SciDAC) program within the U.S. Department of Energy (DOE).

This work is also supported in part by the National Science Foundation (NSF) grant CNS-0437508 and a DOE Early Career PI grant awarded to Dr. Jian Huang (No. DEFG02-04ER25610). This research used resources of the National Center for Computational Science (NCCS) at ORNL, which is managed by UT-Battelle, LLC., for DOE under Contract No. DE-AC05-00OR22725. I have also graciously been granted access to resources from the Argonne Leardership Computing Facility.

Last but not least, my family and friends have always been very supportive of me. They have let me do what my heart tells me to do, and they have let me do it without question. For that, I will always be thankful.

Abstract

According to a recent exascale roadmap report, analysis will be the limiting factor in gaining insight from exascale data [Dongarra, J. et. al 2010]. Analysis problems that must operate on the full range of a dataset are among the most difficult. Some of the primary challenges in this regard come from disk access, data managment, and programmability of analysis tasks on exascale architectures. In this dissertation, I have provided an architectural approach that simplifies and scales data analysis on supercomputing architectures while masking parallel intricacies to the user. My architecture has three primary general contributions: 1) a novel design pattern and implmentation for reading multi-file and variable datasets, 2) the integration of querying and sorting as a way to simplify data-parallel analysis tasks, and 3) a new parallel programming model and system for efficiently scaling domain-traversal tasks.

The design of my architecture has allowed studies in several application areas that were not previously possible. Some of these include large-scale satellite data and ocean flow analysis. The major driving example is of internal-model variability assessments of flow behavior in the GEOS-5 atmospheric modeling dataset. This application issued over 40 million particle traces for model comparison (the largest parallel flow tracing experiment to date), and my system was able to scale execution up to 65,536 processes on an IBM BlueGene/P system.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

The advent of supercomputing and computational science has allowed the study of problems of extraordinary magnitude. Scientists can now set physics into action with supercomputers that are capable of over eight petaflops. Output from these simulations are voluminous in disk space and complex in dimensionality. The analysis and visualization of this data oftentimes must take place at similar scales.

Among the most challenging of these tasks is full-range analysis. A full-range analysis task is a procedure which requires accessing up to the entirety of a scientific dataset. Consider the problem of drought analysis, which requires a global search of temporal shifts in vegetation. Also, consider flow tracing or volume rendering, which integrates particle and light trajectories throughout the dataset. These are some of the most common examples of full-range analysis.

My end goal in this dissertation is to develop an architecture that facilitates the process of performing full-range analysis tasks at scale while ultimately providing interactivity to the user.

1.1 The Challenges of Full-Range Analysis

My goal is met with many challenges, created mostly by the the sheer size and complexity of modern scientific datasets. It is common for datasets to span multiple files, contain

multiple variables, be stored across multiple timesteps and on different grids. This growth in data size and complexity is exacerbated by the fact that more and more scientists employ ensemble runs to evaluate stability of their modeling results.

In result, the scientific data analysis pipeline is faced with the following technical hurdles.

1. *Disk Access* - Disk access is the slowest component of modern parallel supercomputers, causing large-data analysis programs to be dominated by time spent reading data. Along with this, the physical layout of scientific datasets often does not match the disk access patterns of processes. This places a burden on application programmers, requiring them to either utilize low-level advanced I/O techniques or possibly suffer orders-of-magnitude worse performance.

2. *Data Organization* - Managing parallel data structures, data decomposition, and data movement is a challenging and error-prone task. Since load imbalance can play a crucial role in analysis, it is often necessary to implement advanced parallel data structures and data movement techniques.

3. *Programmability* - Programmability of I/O, data organization, and general full-range analysis is still very difficult even for seemingly simple tasks (such as statistical techniques or global reductions). This is largely true for I/O. Until our work, there were still are no libraries or routines that natively support parallel access to multiple files at once - a common need that arises when working with scientific datasets. Furthermore, only until very recently have other parallel libraries even supported general analysis tasks, such as parallel volumetric decomposition, neighborhood communications, and parallel sorting.

1.2 The Need for an Architectural Approach

Tackling all of the aforementioned challenges at once requires an architectural approach. As Google has shown with their design of MapReduce [Dean and Ghemawat 2004], infrastructures can greatly mask programming complexity, provide unprecendented scalability, and also deliver a general analysis model. In fact, the MapReduce model has handled almost 85% of Google's large data analysis needs [Dean and Ghemawat 2008].

MapReduce has provided the most impact in the area of programmer productivity. This is one aspect that cannot be overlooked when facilitating the development of large-scale analysis applications. As I will show, the ability to enhance programmer productivity by masking parallel I/O, partitioning, and data management code is greatly enhanced by a systematic approach.

An architecture that can mask these details is still fundamentally new to many scientific applications; however, some initial progress has already utilized the power of the MapReduce programming model in scientific applications [Tu et al. 2008, Hoefler et al. 2009, Plimpton and Devine 2011, Stuart and Owens 2011]. While MapReduce has proved successful in industy and even some scientific applications, I will argue in this dissertation that it lacks the generality to be applicable in many common scientific analysis tasks. One of the primary examples in this regard is the quantitative assessment of flow properties. Another example stems from the interactivity needed when performing query-based analysis – a characteristic not provided by the typical batch processing model of MapReduce implementations.

1.3 An End-to-End Solution for Full-Range Analysis

I have addressed the problem of full-range data anlysis and the aforementioned challenges in a general and comprehensive manner. My architecture provides new simplified design patterns and novel system techniques for achieving high scalability in the following three key areas.

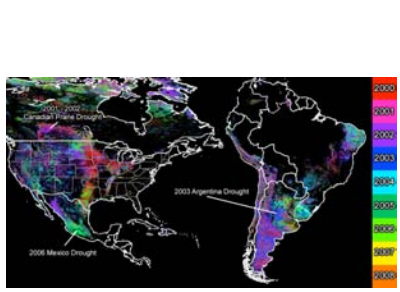
1. *Parallel I/O* - Disk input / output is the main bottleneck in large data analysis. I have designed an I/O solution that is general for many full-range anlaysis tasks [Kendall et al. 2011a]. By using only two functions, my I/O solution elegantly utilizes a wide range of parallel I/O libraries on native application datasets that may span many files and variables. I have also shown superior I/O scalability with this method when compared to traditional single file I/O techniques.

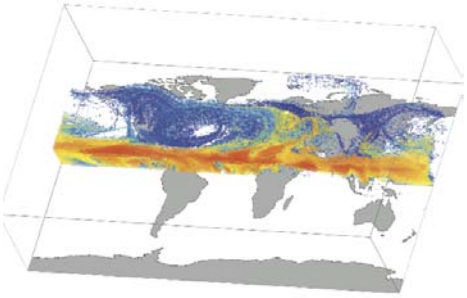
2. *Local Domain and Data-Parallel Analysis* - I have designed a novel solution for many local domain analysis tasks (such as temporal or spatial neighborhood analysis) [Kendall et al. 2009]. The solution combines querying and sorting under a simple design pattern with two functions. It masks details of static load balancing, I/O, and data shuffling.

3. *Global Domain and Task-Parallel Analysis* - Many full-range analysis tasks, such as flow analysis and ray tracing, require task parallelism and potential access to the entire domain from each task. I have created a novel system for these applications that utilizes a simple programming model inspired by MapReduce [Kendall et al. 2011b]. This system is the first ever to combine domain traversal and data reduction under a programming model. It also masks runtime data distribution and load balancing to the user.

An overview of the general architectural design is in Figure 1.1. At the bottom of the figure is a representation of a scientific dataset in netCDF format spread across multiple files and stored on a parallel file system. The Block I/O Layer (BIL) can be used to aid in maximizing read bandwidth for these datasets without the need to understand the details of organizing efficient disk access.

Above BIL are two higher-level components: the Scalable Query Interface (SQI) and DStep. These components use BIL for high performance I/O. They also abstract several other complexities such as parallel data movement, partitioning, and load balancing. They are kept as separate components because they aim to optimize two different types of parallel





Simplifies data-parallel analysis, such as examining voxel-based temporal trends

Simplifies task-parallel applications, such as flow analysis

Scalable Query Interface
- SQI_Query()
- SQI_Sort()

DStep
- DStep()
- Reduce()

Data organization, load balancing, and data movement

Facilitates dataset access and masks I/O complexities

Block I/O Layer
- BIL_Add_block_{raw, nc, hdf}()
- BIL_Read()

Parallel File System

Scientific Dataset

1.nc 2.nc 3.nc 4.nc 5.nc

Figure 1.1: An overview my architectural approach to full-range analysis. The bottom component, the Block I/O Layer, masks parallel I/O intricacies across scientific datasets. On top of that, the Scable Query Interface hides data partitioning and other parallel complexities for data-parallel analysis tasks. Another component for task-parallel applications is DStep, which also transparently schedules work and performs data movement in a scalable manner. All of these components are abstracted by simple design patterns or parallel programming models for their specific goals.

analysis applications: those that obtain parallel speed-up through operations on separate elements of data (data parallel), and those that obtain parallel speed-up by assigning tasks to different processes (task parallel).

As shown in Figure 1.1, users are provided with a set of simple interfaces to leverage these components in high- or low-level application manners. For example, if a user wishes to perform drought analysis (covered in Section 6.1), they could make use of SQI for its data-parallel capabilities. On the other hand, flow analysis (covered in Section 6.3) is a more task-parallel problem that would be readily handled by DStep. In both of these cases, BIL would be used transparently by the architecture. However, if the user wanted to optimize read performance (such as in OSUFlow in Section 3.2), BIL is able to be used as a separate module.

I designed the interfaces of these modules in such a way to give the user a restricted but general model of programming. As I will discuss throughout this dissertation, the design choices were purposefully made in order to utilize new and advanced parallel algorithms (such as two-phase multi-file I/O (Section 3.1), modified parallel sample sorting (Section 4.4), and two-tiered asynchronous communication (Section 5.4)).

1.4 Impact on Computational Science

The generality of my solution has enabled several studies to take place in different application areas - studies which previously were unfeasible because of large data demands. Along with this, I have achieved interactivity that is crucial to the scientific discovery process.

For example, I have allowed scientists to interactively choose multivariate thresholds to define and extract features of interest. The complex problem space that needed to be searched in regards to discovering drought and winter-length [Kendall et al. 2009] or examining eddies and major ocean currents (Section 6.2) could only be performed with interactivity at large scale.

Another key application area I have helped is the area of internal model variability studies [Kendall et al. 2011b], where previous attempts at studying the variability of the models was restricted to downsampled versions of the dataset in the time or elevation range [Galbally et al. 2010]. My architecture helped scientists compare quantitative windfield measurements and effects of Northern / Southern Hemisphere interaction on the entire eight-model run. Using up to 64 K cores on a BlueGene/P machine, my system was able to scale advection of over 40 million particles in parallel, which is also larger than the previous largest particle tracing experiement (125 thousand particles [Peterka et al. 2011b]).

In the following, I discuss the novel components and design of my architecture for full-range analysis.

Chapter 2

Background

My research is related to many areas of large data visualization. In this background section, I aim to divide the full-range analysis pipeline into separate components and discuss the related work in detail. I also discuss other related competing systems. Before diving into these categories, I first provide information about the computing resources and datasets used in this work. I also provide background information of the driving applications in this dissertation.

2.1 Datasets and Computing Resources Used in This Work

For convenience, I overview the datasets that are used in my experiments/applications and the computing resources that are used.

Datasets:

GEOS-5: The GEOS-5 general climate model (GCM) uses a flux-form semi-Lagrangian finite-volume dynamical core with floating vertical coordinates developed by [Lin 2004]. The GCM computes the dynamical tendencies of vorticity, divergence, surface pressure, and a variety of selected trace constituents. The spatial resolution of the model is a $1^\circ \times 1.25^\circ$ lat-lon grid with 72 vertical pressure layers that transition from terrain-following near the surface to pure pressure levels above

180 hPa. The top vertical boundary is at 0.01 hPa (near 80 km). At the ocean surface, temperature and sea ice distributions are specified using a global data set, and the Hadley Center sea surface temperatures match the calendar dates of the output.

An eight-member ensemble of simulations using a free-running model, each initialized with meteorology from different days in January, was performed in order to examine the effect of internal-model variability on simulated trace gas distributions. Annual CO_2 flux values distributed both hourly and monthly are input from the Carnegie Ames Stanford Approach (CASA) datasets for the years 2000 and 2001 in each of the eight model runs. In total, the eight-model daily dataset consists of 5,840 timesteps saved in separate daily netCDF files. Each file has roughly 35 floating-point variables, totaling to ≈2.3 TB of data.

MODIS: Part of NASA's MODIS [*] satellite database is used in my experiments. The dataset used consists of a 500 meter resolution sampling of North and South America, creating a 31,200 by 21,600 grid. The dataset is continuously updated, and 417 timesteps of 8 day intervals from February 2000 to February 2009 are used. Two short integer variables are saved: NDVI and NDWI. NDVI is the normalized difference vegetation index and is a useful indicator of vegetation content. NDWI is the normalized difference water index and is a good indicator of moisture content of soil. The entire dataset totals 1.1 TB.

POP: The Parallel Ocean Program (POP) is a high-resolution eddy-resolving ocean circulation model [Maltrud and McClean 2005]. POP was started using observational analysis, and the general circulation is well represented. To allow inclusion of the Arctic Ocean, it employs a displaced tripole grid. The grid is 2.5D and has 40 layers of u and v velocity components at a 3,600 by 2,400 resolution that spans monthly from February 2001 to September 2003. Along with the u and v components, POP generates salinity and temperature variables. The dataset totals 165 GB.

[*] http://modis.gsfc.nasa.gov

Jet: The Jet dataset is computed from a Navier-Stokes jet propulsion simulation that has u, v, and w floating point velocity variables saved in tuples on a 256^3 grid across 2,000 timesteps in separate raw binary files (375 GB total).

Nek5000: The Nek5000 dataset is generated by a large-scale Computational Fluid Dynamics (CFD) solver. Our dataset consists of one timestep on a 2048^3 grid consisting of three floating-point velocity variables. It totals to $\approx$100 GB.

Computing Resources:

Intrepid: A BlueGene/P supercomputer at the Argonne National Laboratory. Intrepid contains 40,960 nodes consisting of quad-core 850 MHz IBM PowerPC processors and 2 GB of memory. Intrepid's computing resources are organized into node cards (32 nodes per card), which are then organized into racks (32 node cards per rack). Intrepid consists of 40 racks.

Intrepid has three different types of high-speed networks which are utilized by the underlying MPI implementation. The first network is a 3D torus network, which features six outgoing connects per each node to others on a 3D grid. This network is useful for point-to-point communications. The second network is a tree network which is useful for collective operations such as gathering, scattering, and reducing data elements. The third network is a global barrier and interrupt network. This network is specifically inteded to allow all processes to synchronize with extremely low latency (1.6 microseconds).

Intrepid contains a GPFS parallel file system. For each set of 64 nodes (a *pset*), there is one additional node that is dedicated to forwarding I/O requests from those processes to separate storage servers. In total, 640 I/O nodes are connected through 10 Gigabit Ethernet ports to the storage system, which consists of a diameter-5 Myricom Myri-10G switch complex. Studies from [Lang et al. 2009a] show peak obtained read bandwidth rates of up to 48 GB/s using all of the I/O nodes.

Jobs on Intrepid can run in three different modes. The first mode is SMP mode, which utilizes one MPI task per node and allows it to have access to all 2 GB of memory. In this mode of execution, MPI processes are allowed to spawn up to three more threads for shared memory access. The second mode, dual mode, allows execution of two MPI tasks which have access to 1 GB of memory each. Although the memory is technically not distributed, the programmer must treat the memory as distributed between the processes. Similar to SMP mode, the nodes may spawn one additional thread each. The last node, virtual node mode, assigns one MPI task to each core and grants them access to 512 MB of memory each.

JaguarXT4: A Cray XT4 machine located at the Oak Ridge National Laboratory. JaguarXT4 consists of 7,832 quad-core 2.1 GHz AMD Opteron processors with 8 GB of memory. The system totals to 31,328 cores with 62 TB of main memory. The nodes are connected in a 3D torus topology.

The parallel file system on Jaguar is the Lustre file system [†]. Instead of having dedicated nodes for forwarding I/O such as Intrepid, Jaguar runs Lustre clients on the compute nodes. These nodes first contact a Lustre Metadata Server (MDS), which is responsible for providing information about how the file is stored across storage targets. The Lustre clients then interact with these storage servers over the Cray torus network. The JaguarXT4 machine contained a partition of 144 separate storage targets during experiments. Results from [Fahey et al. 2008] showed a maximum peak obtained read bandwidth of 42 GB/s.

Sonoran: A fat SMP node with an Ubuntu Linux installation. Sonoran contains two quad core Intel Xeon processors clocked at 2.1 GHz. The file system is a serial Linux file system with two Barracuda 7200.11 SATA hardrives that each have a theorectical peak of 375 MB/s bandwidth.

[†]http://www.lustre.org

2.2 Driving Applications

My work was motivated to address general scientific application needs at large scale. Because of this, the driving applications are the testimony to the application needs that my solution addresses. Here, I discuss the three driving applications that have been enabled by my approach. I overview the importance of these applications and related smaller-scale studies of the problems done by previous researchers.

2.2.1 Discovering Multivariate Climatic Trends in Observational Data

The first driving application is to study observational data in NASA's MODIS database to discover multivariate climatic trends. Over the years, many researchers have sucessfully utilized observational data and correlated measurements with real world events. The daunting task in this regard is that many real world events (such as drought) have key multivariate thresholds that define an event. For example, the right combination of vegetation and water indicies at certain thresholds can be an indicator that drought has occurred [Gu et al. 2007, Wang et al. 2008, Liu and Wu 2008]. Studying combinations of these thesholds can help provide more accurate understandings of how observational data relates to real world occurrences.

The difficulty in observational data such as MODIS is not only the combinatorial space that could potentially be explored, but also the overwhelming dataset size for even a study of just the Western Hemisphere. The dataset used in this application consists of a 500-meter sampling of North and South America, creating a 31,200 by 21,600 grid. The dataset is continuously updated, and we used 417 timesteps of 8 day intervals from February 2000 to February 2009. MODIS data is stored in various wavelength bands which may be used to compute other variables. By computing two variables that are related to the application and storing them as short integers, the entire dataset totals to $\approx$1.1 TB. With a dataset of this magnitude, we have two primary goals: to provide visualization and analysis methods for cases when many of the timesteps of the dataset need to be loaded, and to deliver usability and near-interactive functionality specifically for application scientists.

The visualization aspect of this task is very demanding for several reasons. First, extracting isocontours with an inherent temporal component is rather new in the field. Drought, for instance, is not a single timestep event and requires "abnormally low rainfal" to have lasted for a period of time. Although tracking contours over time has already been solved [Silver and Wang 1997], visualization typically still treats contour extraction statically in the spatial domain [Lorensen and Cline 1987]. We instead seek for spatial locations in a contour that fit two criteria: thresholds for scalar variables, and the number of continuous timesteps that meet the first criteria. Advanced data structures are necessary to allow for such high dimensional searches. Unfortunately, the most likely methods and data structures, particularly those used in query-driven visualization, incur preprocessing overheads on the order of several hours for datasets of 100 GB [Glatter et al. 2006, Gosink et al. 2008, Stockinger et al. 2005b].

Second, current data analysis tools are not efficiently integrated with system level tools such as parallel I/O. This causes a severe bottleneck in efforts to shorten the end-to-end latency as outlined by the vision of in-situ analysis and visualization [Ma et al. 2007]. In addition, few current data analysis tools have the scalability to leverage modern systems, such as the Cray XT4 and the IBM Blue Gene/P. Not being able to fully leverage systems of that caliber, especially the next generation storage and parallel I/O systems, imposes yet another bottleneck in obtaining the full potential of data analysis tools in production scientific use.

Third, typical user work flow includes an often neglected but very expensive component. That is, to get the datasets from an application-native format into a format that is the most amenable to the parallel data analysis or visualization. To minimize the end-to-end latency, this step must be studied in depth. In this application, we specifically focused on one of the most common cases of our collaborators, where individual timesteps are initially stored in separate netCDF files.

To the best of our knowledge, there were no previous studies of climatic events that were interactively performed at this scale before the publishing of this application study [Kendall

et al. 2009]. Many of the studies [Gu et al. 2007, Wang et al. 2008, Liu and Wu 2008], which we cover in detail in Section 6.1, utilized a smaller or subsampled dataset.

Very recent attempts [Mills et al. 2011, Kumar et al. 2011], however, have used the full-range MODIS dataset and supercomputing resources for studying mountain-pine beetle infestation. [Kumar et al. 2011] showed various accelerations to the k-means clustering algorithm that significantly increased performed of the algorithm at large scale. We are unaware of any recent attempts that aim to interactively analyze the data. While we are aware of other open source GIS packages such as GRASS [GRASS Development Team 2008] and climate analysis packages such as NCL [‡], they currently have serial restrictions which make them infeasible to interactively study the aforementioned problems.

2.2.2 Assessing Geometric Features in Large-Scale Flow Data

Geometric representations of flow fields provide a measure to obtain and extract informative features from a dataset. The analysis of flow fields in the Parallel Ocean Program (POP) is one application that greatly benefits from this. Eddies, along with currents and other natural flow features, may be described geometrically with quantitative thresholds. One example is describing an eddy or current with the angle of turn or residence of the flow field. Similar to the previous MODIS application, the largeness of the data paired with the unbounded amount of thresholds used to describe features makes interactivity very challenging.

POP produces a 2.5-dimensional grid and has 40 layers of u and v velocity components at a 3,600 by 2,400 resolution. The dataset was simulated in monthly intervals from February 2001 to September 2003. Along with the u and v components, POP generates salinity and temperature variables. The dataset is 165 GB in total. While 165 GB is not large in comparison to other modern datasets, the complexity of the data and the anlaysis makes interactive exploration very challenging.

[‡]http://www.ncl.ucar.edu

3D unsteady flow fields contain a vast space of geometric, temporal, spatial, and derived scalar characteristics that can be useful for understanding the underlying nature of the flow. Exploring these feature spaces interactively is essential for developing accurate understanding of the phenomena of interest. In addition to this, the ability to succinctly express these features plays a major role in the exploratory process. Various research from information and scientific visualization have uniquely addressed these challenges [Doleisch et al. 2003, Wei et al. 2010, Shi et al. 2007] in a variety of application domains [Doleisch et al. 2004, Kehrer et al. 2008].

Although scalar-based feature exploration has been widely studied, there has only been limited research [Shi et al. 2007] regarding the effectiveness of using geometric features to explore 3D unsteady vector fields. In addition to this, no quantitative evidence has been shown to claim interactivity on large datasets. Two overarching problems in flow visualization are the cause of this. As noted by [McLoughlin et al. 2009], these include the "sheer volume of data that may be generated from complex simulations" and the "very little work in seeding of integral curves in 3D, unsteady flow fields."

These issues have inspired us to solve this problem by leveraging large data visualization techniques. Our approach relies on recent success in parallel particle tracing [Yu et al. 2007, Pugmire et al. 2009, Peterka et al. 2011b] paired with an architecture for interactively extracting features. This has provided us with the ability to explore neighborhood-based geometric attributes not previously studied due to their data-intensive demands. As we will show in our driving application of ocean modeling (Section 6.2), these neighborhood-based attributes are general enough to not only capture turbulent cold-core eddies, but to also capture more laminar features such as the major ocean currents. The generality that is offered by these features is powerful, primarily because of the degree of freedom that is allowed in the neighborhood size that defines them. Interactively tuning this parameter is critical for the user to uncover precisely what defines the feature of interest – a factor that is enabled by a scalable system such as ours.

Also germane to our work are some important commonalities with recent systems, including SimVis [Doleisch et al. 2003] and other mainstream applications like ParaView [Moreland et al. 2007] and VisIt [Childs et al. 2006]. Many of these systems have components that rely on range queries as the most basic level of data abstraction [Doleisch et al. 2003, Kendall et al. 2009, Stockinger et al. 2005a]. Users of these systems also appreciate the intuitiveness offered by describing features using high-level mini programming languages [Doleisch et al. 2003, McCormick et al. 2004, Glatter et al. 2008] or algebraic formulas (e.g. "expressions" in VisIt and the "calculator" interface in ParaView). We have designed our system on top of range queries and have also integrated procedural methods for expressively describing features to complement these approaches. This generality makes our system applicable to a wide variety of other tools.

In this application, we primarily focus on the low-level aspects that are required for interactivity such as preprocessing, parallel I/O, and scalability. This focus has resulted in minimal feature extraction time, a characteristic that is necessary for users to quickly and precisely uncover complex features with no *a priori* definitions. We show the efficacy of our approach by exploring a large and widely studied global ocean modeling simulation (Section 6.2). We believe the scientifically understood nature and public availability of this dataset provide a fresh benchmark for the flow visualization community and a step forward in feature-based flow visualization of large datasets.

2.2.3 GEOS5 Internal Model Variability Studies

The final driving application is terascale atmospheric data analysis. Accurate modeling of the atmosphere is critical to the understanding of global and regional climate: past, present and future. Likewise, determining the relative contributions of the various sources and sinks of atmospheric CO_2 in different regions is critical to understanding the global carbon budget. As global circulation models move to higher spatial and temporal resolution and are capable of incorporating detailed ground-based and satellite observations as initial conditions for future predictions, custom tools for analysis will be required.

A variety of analysis tools are currently available such as the NCAR Command Language (NCL) [§], Interactive Data Language (IDL) suites [¶], and general scientific visualization tools such as VisIt [‖]. Tools that both capture flow lines and provide meteorological statistical analysis of particle trajectories at high resolution, however, are limited, both in availability and capability. Most analyses rely on point-to-point comparisons [Erickson et al. 2007, Ott et al. 2010], or some type of model output reduction such as Empirical Orthogonal Functions (EOF) [Hannachi et al. 2007] or various types of sampling of the data output [Galbally et al. 2010] because of limited computational power.

For example, in a very recent 2010 assessment of the effects of biomass burning in Indonesia, [Ott et al. 2010] ran two ten-member ensemble simulations, each ensemble with aerosol input data from a different source. The means of the ensembles were then calculated along with each member's difference from its ensemble mean (and the *Student's t-test* performed) in order to evaluate the significance of the global and regional change of a given variable as a result of the change in aerosol concentration. Results from this study were presented as difference plots using standard atmospheric visualization tools. The authors indicated that a limitation of the study was the inability to identify potential climate teleconnections and impacts outside of the immediate source region.

In Section 6.3, we directly evaulate CO_2 and windfield teleconnections in GEOS-5 atmospheric modeling data using DStep. Our analytical method addresses CO_2 teleconnections directly since flow among regions is examined first in four dimensions, and probability distributions are computed from the results. The simplicity of our approach paired with the scalable back end has allowed us to write succinct and expressive custom data analysis applications using DStep. As a result, atmospheric scientists have a new way to evaluate the longitudinal dependence of inter-hemispheric transport; for example, to support CO_2 source apportionment according to movement patterns of specific CO_2

[§] http://www.ncl.ucar.edu
[¶] http://www.ittvis.com/idl
[‖] https://wci.llnl.gov/codes/visit

molecules between the Northern and Southern Hemispheres. Furthermore, they also have innovative methods for assessing internal-model variability.

2.3 Components of Full-Range Analysis

My systematic approach to full-range analysis ties together many different components in a way to simplify the process of developing scalable applications. I overview the components of my approach in the following.

2.3.1 Feature Characterization and Extraction

A powerful feature characterization component is crucial for the usefulness of a full-range analysis system. Many times users simply do not know or understand the thresholds or characteristics of variables that define a feature of interest. The extraction of features through time has been one of the most popular topics in this regard. One example is the usage of machine learning to identify the best transfer function for feature in a volume while it progresses through time [Tzeng and Ma 2005]. Other research [Silver and Wang 1998, Ji et al. 2003] uses automated methods to detect time-varying parameters such as the proper isosurface value or time-varying events such as bifurcation.

Instead of using a black box or automated approach, my work has shifted more focus on allowing users to identify key parameters that define a feature of interest. In this regard, it has overlap with other works that define features in a procedural manner. Some examples include [Glatter et al. 2008], who used a regular-expression based language to issue queries and extract temporal events. [Woodring and Shen 2006] used tree-based procedural expressions with logical operators for comparative visualization purposes. Another procedural example is SimVis [Doleisch et al. 2003], which has a Feature Definition Language (FDL) for applying set and fuzzy set operations to ranges of data. The FDL has the capabilities to describe phenomena of interest by adding features with various characteristics that can have logical operations applied to them. Case studies with SimVis

show its applicability to flow fields [Doleisch et al. 2004, Kehrer et al. 2008], however, it only operates on scalar quantities of the flow fields and does not have the ability to operate on geometric quantities.

Other less quantitative examples of feature characterization and extraction are prevalent in the field, although I have yet to adopt them in my work. For geometric processing of flow fields, [Wei et al. 2010] used a sketch-based interface for classification and visualization. With this approach, users can sketch a pattern of interest and then relevant fieldlines matching the pattern will be returned. Furthermore, associated clusters of fieldlines with similar qualities of the sketched line may also be examined. Our approach is quite similar to the authors' in that we use geometric properties of fieldlines when searching for user queries. Visually-oriented approaches such as [Wei et al. 2010], however, lack in the quantitative ability to describe more complex geometric properties, such as fieldlines that turn at most a certain degree or swirl for at least a certain percentage of their length.

2.3.2 Parallel I/O

The continuing growth of data generated by scientific simulations creates challenges for full-range analysis applications, especially those that need to perform reductions on the dataset. For these types of operations, parallel I/O is indispensable.

Significant advances have improved the usability and portability of parallel I/O across high-performance systems. I/O interfaces built on top of the MPI-2 [Gropp et al. 1998] standard, such as ROMIO [Thakur et al. 1999], have delivered methods to enhance distributed access of datasets. One primary advanced I/O method that is available in MPI-2, *collective I/O*, is especially useful for postprocessing applications. Collective I/O can take individual requests from PEs and then aggregate requests into disk accesses that more closely match how the underlying data is stored. In my proposed system, I directly leverage this optimization and discuss how it can be utilized more efficiently for some types of scientific datasets.

I/O has recently been gaining more attention in visualization research. [Ma et al. 2003] first showed how overlapping I/O with rendering could significantly reduce interframe delay in the parallel rendering of large-scale earthquake simulations. [Yu et al. 2004b] extended this by presenting I/O solutions for a parallel visualization pipeline. [Yu et al. 2004a] also presented two parallel I/O methods for the visualization of time-varying volume data in a high-performance computing environment.

[Peterka et al. 2008] have recently performed extensive work on large-scale parallel volume rendering of astrophysics data on the IBM Blue Gene/P. Their method of visualization focused on efficiently using parallel I/O to increase frame rates. By using a system like the Blue Gene/P, the authors showed that a simulation-caliber resource is a valuable alternative to a graphics cluster for visualization, especially when I/O is the bottleneck.

It is common for scientists to store datasets across multiple files that contain sepaate timesteps or variables. The study of parallel I/O across multiple files has had little attention. [Memik et al. 2006] coined the term "multicollective I/O" to describe collective parallel access to multiple files. In their work, the authors examined reducing I/O and data transfer times in collective methods to multiple files. They showed this problem was NP-complete when the files were arbitrary sizes and provided various solutions to the problem. Some of their primary contributions were analyzing greedy heuristics that aimed to minimize data transfer over the network. In my I/O work, I have utilized a similar greedy heuristic approach for performing multiple file reading.

Although I do not specifically focus on the area of writing multiple files, there are two works in this area that have shown significant advancements. [Gao et al. 2009] showed how writing multiple files could help reduce lock contention in Parallel netCDF. [Lofstead et al. 2010] also utilize multiple file output to manage I/O variability during a large simulation.

2.3.3 Parallel Scientific Data Querying

Query-driven visualization [Glatter et al. 2006; 2008, Gosink et al. 2007; 2008, Stockinger et al. 2005b] has become a popular research topic. Query-driven methods rely on translating a user interest into a compound Boolean range query. Regardless of the underlying querying mechanism, such as a distributed M-ary search tree [Glatter et al. 2006], bitmap indexing [Stockinger et al. 2005b] or bin-hashing [Gosink et al. 2008], query-driven visualization has been shown to accelerate contour extraction in large datasets. It is also naturally extended to multivariate datasets. The methods for query-driven visualization can be split into two categories: tree-based and index-based methods.

In indexed-based methods [Stockinger et al. 2006; 2005b, Rübel et al. 2008], bitmap indexing is used to accelerate the search phase. To build the index, each record r with v distinct attribute values generates v bitmaps with r values each. This allows for bitwise logical operations to be performed on range queries, making multivariate range queries simple linear combinations of single-valued queries [Stockinger et al. 2005b]. In tree-based methods, data is sorted and indexed with tree structures such as B-trees or M-ary search tree structures [Glatter et al. 2006].

It has been shown that tree-based methods suffer from the "Curse of Dimensionality", where adding more dimensions results in an exponential growth in storage and processing requirements [Stockinger et al. 2005b]. Although this is true for a naive tree method, [Glatter et al. 2006] showed how an M-ary search tree structure could be used to reduce the storage overhead to less than 1% of the dataset size and also showed that the overhead is not dependent on the dataset size. This factor is attractive to our method of querying since we load the data in main memory. It was also shown in [Glatter et al. 2006] that queries could be load balanced by distributing data on the granularity of single voxels. Although parallel methods of index-based query-driven visualization have been established [Stockinger et al. 2006, Rübel et al. 2008], these do not focus on the issue of load-balanced data extraction. In order for many PEs to scalably perform analyses after querying, whether these types of analyses operate on single voxels, time series, or geometry, load balance is crucial. It

is because of this reason why [Glatter et al. 2006] is used as the basis for query-driven visualization.

2.3.4 Parallel Sorting

Parallel sorting is a major component of some large data analysis tasks. Because of the results in the seminal work by [Blelloch et al. 1998], we chose to use the parallel sample sort algorithm.

The parallel sample sort algorithm randomly samples the dataset choosing samples in such a way to linearly split the entire dataset. The processes then bin their data based on the splitter samples and sort their bins, creating a globally sorted list. Parallel sorting is an inherently network-intensive process, and this algorithm sums up all of its network communication in one MPI_Alltoall call to bin the dataset. To locally sort the bins, we used the quick sort algorithm.

To the best of my knowledge, no research has assessed the implications of this algorithm over one thousand processes. I have developed advancements to this algorithm to aid in its scalability to thousands of processes.

2.3.5 Parallel Particle Tracing

Along with limited capability of large-scale processing tools in atmospheric science, our motivation for a new analysis model stemmed from the recent efforts in parallelizing flow tracing techniques. Scalable parallelization of flow analysis methods remains a challenging and open research problem. The most widely-used flow analysis technique is the tracing of tangential fieldlines to the velocity field. Steady-state fieldlines are the solution to the ordinary differential equation

$$\frac{d\vec{x}}{ds} = \vec{v}(\vec{x}(s)) \, ; \vec{x}(0) = (x_0, y_0, z_0), \tag{2.1}$$

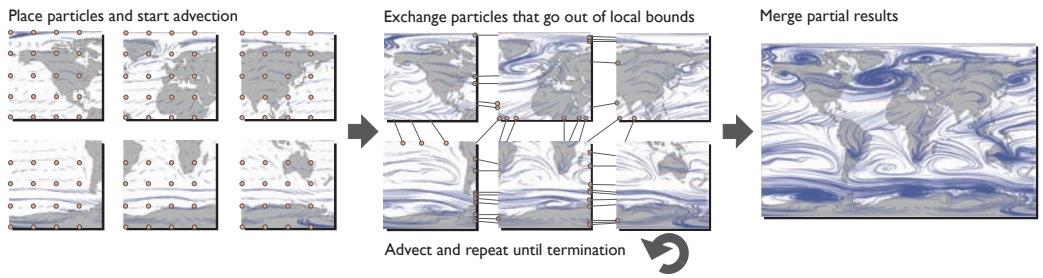


Figure 2.1: One example of domain traversal is fieldline tracing, shown here using six distributed-memory processes. The procedure initializes particles in the subdomains, which are then advected through the flow field. Particles are exchanged when going out of bounds, and all partial fieldlines are merged when particles finish advection.

where $x(s)$ is a 3D position in space (x,y,z) as a function of s, the parameterized distance along the streamline, and v is the steady-state velocity contained in the time-independent data set. Equation 2.1 is solved by using higher-order numerical integration techniques, such as fourth-order Runge-Kutta. For time-varying fieldlines, the integration progresses through space and time.

An example of distributed fieldline tracing is in Figure 2.1, which uses six processes that each own separate parts of the domain. Processes first initialize particles and they begin particle advection through their subdomain. When particles go out of local bounds, they must be exchanged to the owners of the proper subdomain. This continues until particles either exit the global domain or are terminated. The fieldline traces can then be merged and visualized.

Efficient distributed-memory parallelization is difficult because of the communication requirements and task-parallel nature of the problem. The problem has received much recent attention. [Yu et al. 2007] demonstrated visualization of pathlets, or short pathlines, across 256 Cray XT cores. Time-varying data were treated as a single 4D unified dataset, and a static prepartitioning was performed to decompose the domain into regions that approximate the flow directions. The preprocessing was expensive, however, less than one second of rendering required approximately 15 minutes to build the decomposition.

[Pugmire et al. 2009] took a different approach, opting to avoid the cost of preprocessing altogether. They chose a combination of static decomposition and out-of-core

data loading, directed by multiple master processes that monitor load balance. They demonstrated results on up to 512 Cray XT cores, on problem sizes of approximately 20 K particles. Data sizes were approximately 500 M structured grid cells, and the flow was steady. The authors have recently extended the work to operate on multi-core architectures [Camp et al. 2011].

[Peterka et al. 2011b] avoided the bottleneck of directing computation and instead used static and dynamic geometric partitioning strategies for achieving desirable load balance. The authors showed that simple static round-robin partitioning schemes outperformed dynamic partitioning schemes in many cases because of the extra data movement overhead. They showed scalability results up to 32 K Blue Gene/P cores on steady and time-varying datasets on problem sizes of approximately 120 K particles.

The most recent work by [Nouanesengsy et al. 2011] created a solution for maximizing the load balance during streamline generation. In this work, they used the flow field to generate a *flow graph*. Once the flow graph was generated, they could then model the problem of data distribution as an optimization problem. Given the starting location of seeds, the authors could estimate the workload per block (by taking in the average amount of advection steps and other parameters) and then assign blocks accordingly. In some cases, blocks are replicated across processes for better parallel workload distribution. The authors show enhanced scalability on 256 K evenly-spaced particle traces up to 4 K BlueGene/P processes. They showed that in some cases, uneven seeding of streamlines may result in the worst case of using a static partitioning algorithm.

For my work, it is critical to allow generality and simplicity to the user. That is why I chose to use no preprocessing steps and instead design my architecture to exploit asychronous communication while using static partitioning. I show results of using this method against [Peterka et al. 2011b] in Section 5.5 and also application results in Section 6.3.

2.4 Systematic Approaches to Large Data Analysis

Many approaches have aimed to systematically simplify the analysis of large datasets. These approaches often involve domain-specific languages or work towards optimizing one major piece of the large data analysis pipeline. Related examples to my work are discussed in the following section.

2.4.1 In Situ, Coprocessing, and Postprocessing

There are three primary approaches to the analysis of datasets on supercomputing resources. Described in [Ma et al. 2007], these approaches are in-situ analysis, coprocessing, and postprocessing. In-situ processing requires the integration of analysis and visualization into the simulation. This requires merging code with the data structures of the simulation. The authors in [Yu et al. 2010] describe the integration of volume rendering and image compositing techniques with data structures from a large-scale combustion simulation. While the authors had to perform some extra boundary exchanges of data (for gradient computation during rendering), this additional overhead was small considering the valuable time saved by only writing out simulation information when deemed necessary. One of the disadvantes of the approach is that it is hard to generalize in-situ routines to other scientific simulations.

Coprocessing has been a more popular alternative than in-situ because of its generality. There are two primary works which facilitate the coprocessing of data. They do this by providing a layer of transparency between the simulation data structures and analysis applications. One example is the Adaptable I/O System (ADIOS) [Lofstead et al. 2009]. ADIOS provides a lightweight layer for simulations to write out multivariable datasets in structured and unstructured manners. The ADIOS library writes out data in a custom *BP* format which can transparently utilize parallel file systems more efficiently. The BP format can easily be read in by the application, thus facilitating the restart of simulations. ADIOS also has capabilities to send simulation data to a staging area, where analysis can be performed on the fly.

Another coprocessing system called GLEAN [Vishwanath et al. 2011] goes to a higher level than ADIOS. GLEAN sits on the actual I/O servers of a simulation architecture and intercepts data coming to storage. The data can then be forwarded directly to an analysis cluster and visualized in real time. GLEAN offers the advantage that no code has to be changed in a simulation, but it also means that GLEAN must handle specific data output formats (such as FLASH AMR in their example). The authors showed that even using the analysis architecture as a separate asynchronous I/O and staging system can provide orders-of-magnitude improvement over traditional parallel I/O libraries. Efforts have recently been made to more effectively integrate coprocessing components such as GLEAN and ADIOS into widely-used visualization packages [Fabian et al. 2011].

In-situ and coprocessing have been viable alternatives to avoid unnecessary I/O in the analysis pipeline. Unfortunately, some routines require accessing a scientific dataset that may require mulitple runs for completion. This is primarily true in the case of time-varying data analysis. For these situtations, postprocessing is the only appropriate option for analysis.

Postprocessing may involve building a bitmap index of a dataset [Stockinger et al. 2005b], deriving a flow graph [Nouanesengsy et al. 2011] or topological representation of a dataset [Bremer et al. 2011], or performing visualization directly on the dataset [Peterka et al. 2009; 2011b]. For large-scale simulation data, parallelism is critical, yet scientists are still often forced to write low-level MPI code for data distribution, I/O, and data movement. The DIY toolkit [Peterka et al. 2011a] provides higher-level routines for the aforementioned parallel programming difficulties. DIY provides scientists with a means to focus on the analysis components of their application and spend less time performing error-prone data organization tasks. My work is related to DIY in many ways. First of all, my sorting and I/O modules used by my architecture are in DIY. Second, I am also trying to assist the difficulties that arise when managing large data. The main difference with my work is that I am going one step higher by completely hiding parallel data organization details from the user.

2.4.2 Simplified Parallel Data Processing Infrastructures

Although the aforementioned DIY library can aid in postprocessing, there are other examples of systems that aim to simplify data processing by transparently handling all parallel complexities. The areas that are most related to my approach include various visualization systems and MapReduce techniques.

Visualization Systems

Several visualization systems have been developed which transparently mask parallel execution. One defining example in this regard is Scout [McCormick et al. 2004; 2007]. Scout provides the user with a restricted programming interface that helps reveal the parallel nature of GPU programming while also hiding any nuances that are not necessary for visualization. Scout has bult-in visualization operations such as transfer functions, and it also allows for explicit parallel routines such as reductions and scans. Since these types of operations are not immediately available in popular graphics APIs like the OpenGL Shading Lanaguage [**], it provides a much simpler alternative for visualization practitioners.

Scout focuses primarily on data-parallel visualization problems; however, many visualization applications are not immediately data parallel. This is because most are modeled after the *visualization pipeline* and the design of the Visualization Toolkit (VTK) [Schroeder et al. 1996]. In the visualization pipeline, routines are modeled as filter components with inputs and outputs. These filters can be combined by connecting the outputs and inputs together. Parallel execution of the visualization pipeline can be limited in some circumstances, with one primary example being the time-dependent processing of data [Biddiscombe et al. 2007]. A recent framework, DAX [Moreland et al. 2011], has aimed to provide a more efficient parallel computation model than the standard visualization pipeline. Similar to my dissertation work, DAX masks parallel execution to the user and allows them to write *worklets*. The worklets are computational kernels that

[**] http://www.opengl.org/documentation/glsl

provide workers with access to surrounding elements of data. Depending on the amount of surrounding data a worklet needs to access, the DAX runtime can schedule these kernel functions in an efficient manner. The authors showed the usage of the framework for various visualization pipelines. They also provided initial parallel results on a GPU along with the same results of a serial VTK implementation. The authors note that one of the primary challenges of their framwork is optimizing problems which require large inter-worklet communication (such as streamline generation). My dissertation work directly focuses on this problem in Section 5.

MapReduce

It is becoming more popular to solve large-data processing problems with simple serial programming interfaces. The main example in this regard is Google's MapReduce [Dean and Ghemawat 2004], which provides a simple programming framework for data-parallel tasks. I briefly outline the execution of a MapReduce program in Figure 2.2. Users implement a map() and reduce() function. The map() function takes an arbitrary input and outputs a list of intermediate [key, value] pairs. The reduce() function accepts a key and a list of values associated with the key. Reducers typically merge the values, emitting one or zero outputs per key. Output values can then be read by another MapReduce application, or by the same application (i.e. an iterative MapReduce).

While the programming interface is restricted, MapReduce provides a powerful abstraction that alleviates programming burdens by handling the details of data partitioning, I/O, and data shuffling. The power offered to users by this abstraction has advocated new approaches at solving large-scale problems in industrial settings [Dean and Ghemawat 2008]. There are also systems that have implemented MapReduce on top of MPI [Hoefler et al. 2009, Plimpton and Devine 2011] as well as multi-GPU architectures [Stuart and Owens 2011].

The profound success of MapReduce in industry has inspired its use in scientific settings. [Tu et al. 2008] designed *HiMach*, a Molecular Dynamics trajectory analysis

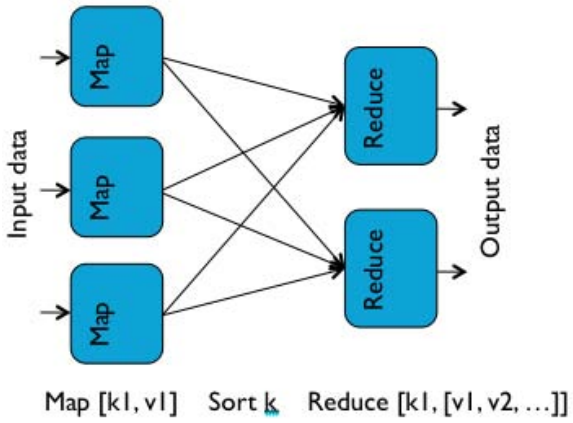


Figure 2.2: An illustration of the execution of the MapReduce programming model.

framework built on top of MapReduce. The authors extended the original MapReduce model to support multiple reduction phases for various time-varying analysis tasks, and they showed scalability up to 512 cores on a Linux cluster.

Although MapReduce is amenable to data-parallel tasks, there are many examples of visualization and analysis routines that have task-parallel components (such as ray tracing and flow tracing). My dissertation work focuses on using designs similar to MapReduce for data-parallel problems in science (such as drought analysis in Section 6.1) and task-parallel problems (such as flow analysis in Section 6.3).

Chapter 3

Managing I/O Complexity at Large Scale

Accompanying architectural shifts over the years, the primary limiting factor in scalability of large scale visualization applications has shifted from computation to I/O [Childs et al. 2010, Peterka et al. 2009]. I/O systems are typically the slowest component of high performance computing architectures. Along with this, the disk access patterns of analysis applications often do not match the physical layout of data on disk, creating an even greater barrier for achieving high performance.

I/O can place an expensive burden on parallel visualization and analysis practitioners. The example in Figure 3.1 is one illustration of this dilemma. In this scenario, a simulation generates data on an IBM BlueGene/P architecture. The simulation computes the physics of a time-varying phenomenon and saves a three-dimensional rectilinear volume at each time step. Many parallel I/O libraries are available to the simulation for storage, with some examples including PnetCDF (for the network Common Data Form), HDF5 (for the Hierarchical Data Format), and MPI-I/O for other custom formats.

While there are many visualization options, such as pathline tracing for vector fields or volume rendering for scalar fields, a parallel visualization approach must first partition the domain across processing elements (PEs). Block-based partitioning is one of the most

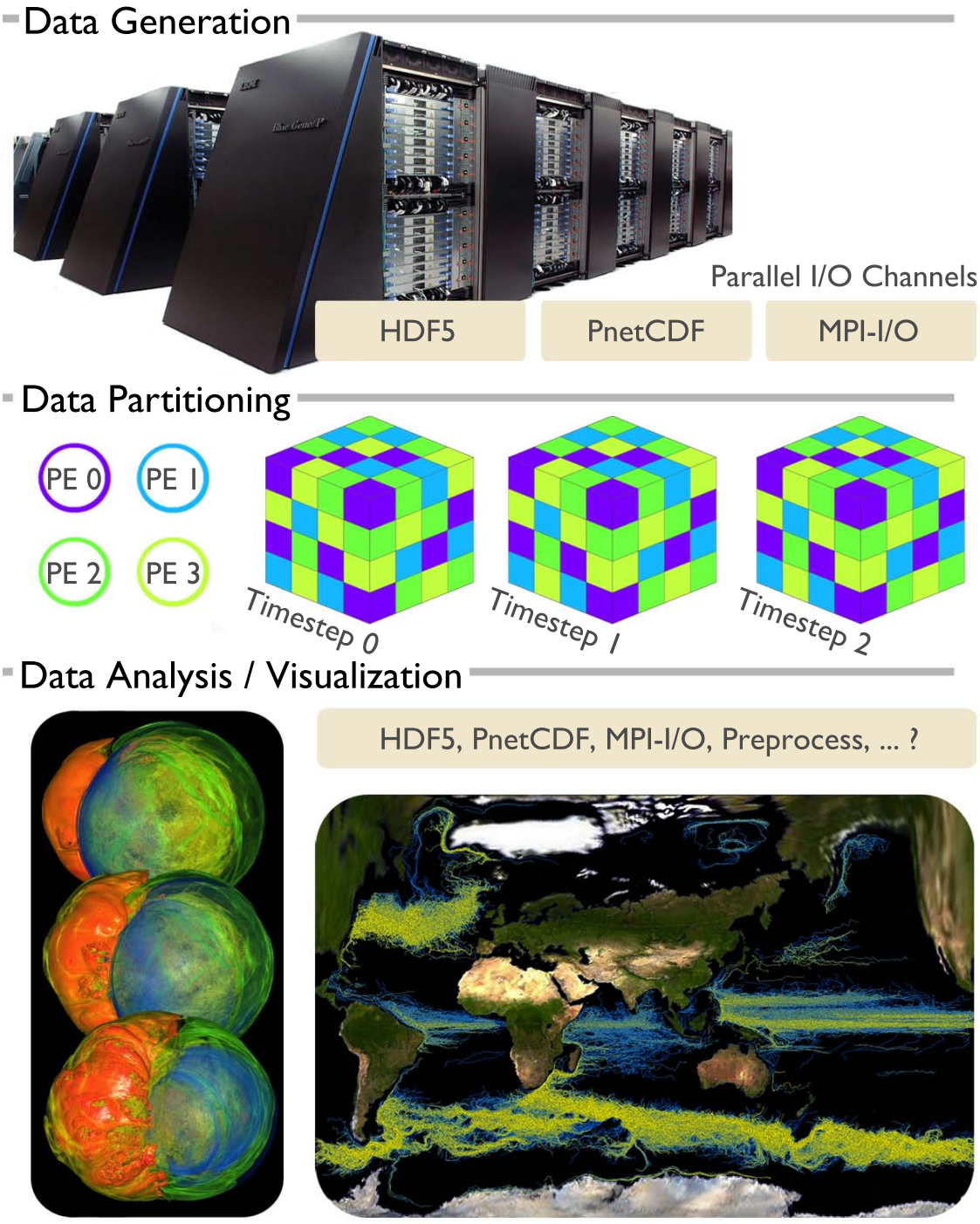


Figure 3.1: Example of a typical simulation and visualization scenario that illustrates some of the primary steps, including data generation, data partitioning, and the resulting visualization. In this example, blocks are distributed and colored by their assignment to four processing elements. The two visualizations show common time-varying techniques, which are pathline tracing and volume rendering.

popular choices for rectilinear grids. In our example, PEs are each assigned multiple blocks for more efficient workload balancing.

The partitioning strategy, which is important for scaling computation, conflicts with physical data storage. If PEs were to issue separate I/O requests for blocks, the many disk seeks and reads will likely result in poor performance. Reading and distributing the dataset from one PE is often the first step taken to avoid this consequence, however, this does not effectively utilize standard parallel file system architectures. Transforming the dataset into a more I/O-efficient format is also another common step. While there has been success in using multi-resolution or compressed out-of-core [Lindstrom and Isenburg 2006] formats, many of these techniques are optimized for serial file systems. Only very recently has parallel access been studied for multi-resolution formats [Kumar et al. 2010]. Furthermore, metadata from higher-level formats which is needed for scientific analysis can easily be lost during this transformation.

The most practical approach uses the same parallel I/O library that the simulation used to write out the data. This approach, however, is still not ideal because of the many possible simulation formats and the difficulties in tuning and understanding low-level details about the parallel I/O APIs. For example, efficiently reading the block pattern in Figure 3.1 requires significant knowledge about MPI Datatypes, the newer non-blocking interface in PnetCDF, or the hyperslab functionality in HDF5. Furthermore, the semantics of these APIs restrict I/O operations to a single file at a time. As we will show in Section 3.2, this can lead to a major underutilization of the available I/O bandwidth for multi-file datasets.

These complexities prompt many challenges for parallel visualization practitioners. Do all researchers and developers have to be parallel I/O experts to create applications that are scalable and portable across scientific formats? Production applications like Visit and ParaView have over one hundred different file readers in use. Will others that desire the same level of ubiquity in their parallel applications also have to pay this much attention to I/O? We believe that there is a need for more generalized parallel I/O solutions that scale across scientific data formats and storage conventions.

3.1 A Design Pattern for Full-Range Parallel I/O

We believe that a more generalized I/O design should center around a partitioning strategy instead of a file format. Rather than having to deal with many formats and API complexities, applications should have access to a simple I/O layer optimized for their partitioning strategy that abstracts file formats and even other intricacies like multi-file dataset storage.

A block-based I/O layer is one solution to this problem. Block-based partitioning, such as the example shown in Figure 3.1, is not only popular in many parallel visualization strategies, but also prevalent in other applications like parallel matrix analysis. To illustrate how such a layer would operate, we have designed and implemented a prototype software, known as the Block I/O Layer (BIL). In the BIL interface, PEs specify a collection of blocks that they individually intend to access, then they collectively operate on the global collection. The interface is designed to simply have two functions:

- *BIL_Add_block_*{*file format*} – Takes the starts and sizes of a block along with the variable and file name. PEs call it for as many blocks as they need, whether they span multiple files or variables. Currently it operates on raw, netCDF, and HDF formats.

- *BIL_*{*Read, Write*} – Takes no arguments. The blocks that were added are either read in or written from the user-supplied buffers.

The implementation is illustrated in Figure 3.2, which shows a simple example of four PEs reading a block-based pattern spanning two files. The PEs first add the desired blocks and then call *BIL_Read*. The requested blocks, which start out as noncontiguous storage accesses for each PE, are aggregated and scheduled into large contiguous accesses. Reading then occurs in parallel and data are exchanged back to the original requesting PEs.

Although the semantics of the underlying parallel I/O APIs would normally restrict users to operate on single files at a time, this design allows the implementation to collectively perform I/O across multiple files. Furthermore, the implementation can use advanced features of I/O libraries when necessary and can be configured for different file

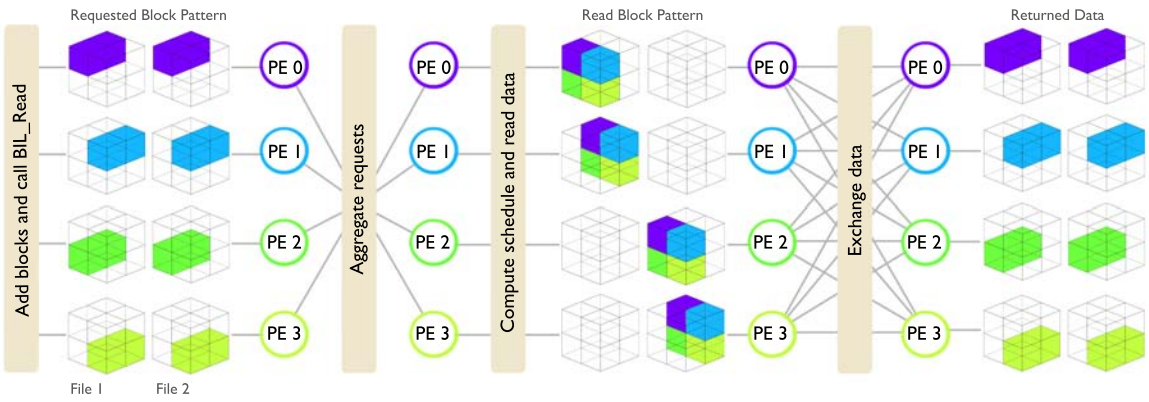


Figure 3.2: An example of how our I/O implementation performs reading of requested blocks. This illustration uses four PEs that each request two blocks that are in separate files. The procedure uses a two-phase I/O technique to aggregate requests, schedule and perform large contiguous reads, and then exchange the data back to the requesting PEs.

systems. For example, we are able to detect when the individual reads of each PE are less than the file system's striping size. When this occurs, we have found that it is generally best to use collective I/O strategies or simply perform I/O from a smaller subset of PEs.

BIL's communication is also built upon advanced MPI mechanisms. For exchanging of data, we use collective communication routines to take advantage of the underlying MPI implementation, which is able to efficiently utilize certain network topologies and architectures. Exchanging data usually takes less than 10% of the overall time, as communication bandwidths are typically orders of magnitude larger than storage bandwidths.

3.2 Flow Tracing I/O Benchmarking Test

We have integrated BIL into OSUFlow, a particle tracing library originally developed by the Ohio State University in 2005 and recently parallelized. The application partitions the domain into four-dimensional blocks (time blocks) and assigns them round-robin to each of the PEs (similar to the illustration in Figure 3.1). For an extensive explanation, we refer the reader to [Peterka et al. 2011b].

OSUFlow has the ability to load time blocks that span multiple files, primarily because scientists often store one file per time step. Its original implementation used parallel I/O libraries to collectively read one file at a time until blocks were completely read. Although this implementation used the I/O libraries in their intended manners, it would still often lead to mediocre performance results.

We compared the original I/O methods with BIL on the Intrepid machine. The comparison used the POP and Jet datasets. The machine and datasets are described in detail in Section 2.1. Bandwidth results appear in Figure 3.3. The top line represents IOR [*], a popular bandwidth benchmark for parallel I/O systems, while the others represent the total bandwidths achieved by the original method vs. BIL. The differences are significant at large scale. At 16 K PEs, we observed a factor of 5 improvement for the POP dataset and a factor of 45 improvement for the Jet dataset. Both BIL results were able to maintain bandwidth rates that were very close to the peak IOR rates. For the Jet dataset, BIL obtained roughly 30 GB/s at 16 K PEs and reduced I/O time from 9 minutes to 12 seconds. At such large PE counts, the amount of data accessed by any given PE when accessing one file at a time is too small to attain any substantial bandwidth; the capability in BIL to concurrently schedule reads to multiple files makes a difference.

3.3 MODIS I/O Benchmarking Test

We tested the bandwidth rates for the Block I/O Layer on varying scales with the MODIS dataset on a different architecture. The machine used was JaguarXT4. An overview of this architecture and parallel file system is described in Section 2.1. In contrast to the previous dataset which was spread across thousands of files, the MODIS dataset is saved across hundreds of much larger files. For a complete overview of the MODIS dataset, see Section 2.1.

The previous section discussed BIL in comparison to IOR benchmarks on the architecture. In this experiment, we used previous IOR benchmark results from other research

[*] http://www.cs.sandia.gov/Scalable_IO/ior.html

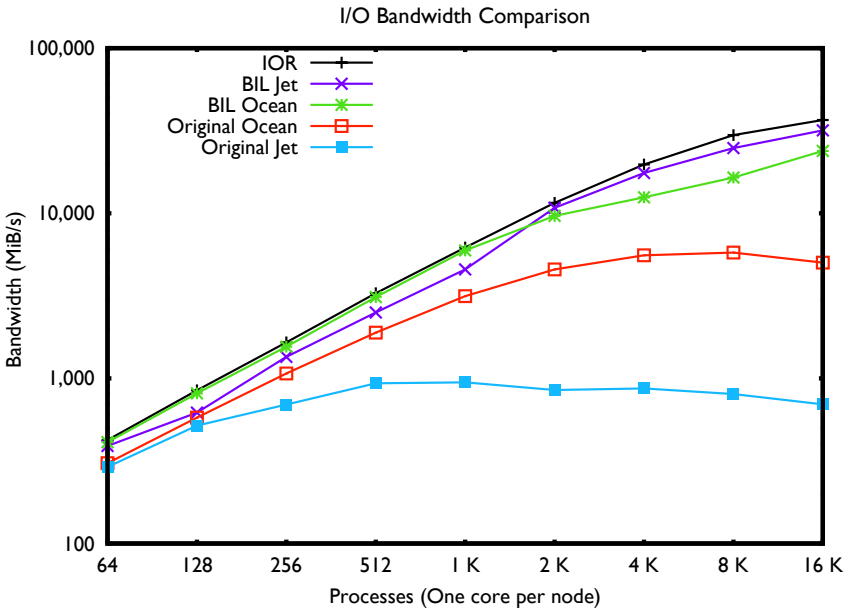


Figure 3.3: Bandwidth results (log-log scale) of our parallel I/O method versus the original parallel I/O method in OSUFlow. All tests were conducted using one core per node (to maximize the amount of I/O nodes used) on Intrepid with two different datasets. The top line represents the IOR benchmark. The original method was using the newer non-blocking Parallel netCDF routines for the POP ocean dataset and collective MPI-I/O for the Jet dataset. The original procedure, however, was restricted to collectively reading one file at a time, leaving much of the available bandwidth unused for these multi-file datasets.

on JaguarXT4 as a comparison. The previous results are discussed in [Yu et al. 2008], where the authors used the IOR benchmark on JaguarXT4 to show how the bandwidth scales when using varying amounts of OSTs to read in data. We use the bandwidth results obtained from using 144 OSTs in [Yu et al. 2008] as a comparison since we also used all 144 OSTs of the JaguarXT4 parallel file system in this test.

Bandwidth results for reading the entire 1.1 TB dataset are shown in Figure 3.4. When timing the results, the files were first opened and then timing was started after an MPI_Barrier. After all the I/O was complete, another MPI_Barrier was called and timing stopped. Timing results were averaged across five separate runs at each process count, and error bars are shown for minimum and maximum bandwidth results. If the aggregate memory of the nodes was insufficient, we instructed BIL to simply discard data during its scheduling and reading of files.

We achieved up to an average of ≈28 GB/s on 4 K cores, roughly 75% of the 42 GB/s benchmark comparison. The bandwidth bottomed out around 2 K cores, and performance degradation was seen when scaling to 8 K and 16 K cores. This same trend was observed in [Yu et al. 2008] on a separate benchmark. This is likely caused by I/O requests oversaturating the network, but further testing is required to confirm this.

When scaling to 4K cores, we were able to minimize the I/O time (on average) on the entire 1.1 TB MODIS dataset to ≈37 seconds. The results show that using collective I/O combined with BIL's approach to saturate the underlying I/O system with requests can achieve significant bandwidth rates, and data in application-native formats can be read in practical amounts of time for an application setting.

3.4 Impact of BIL in Full-Range Analysis Architectures

The design and implementation of BIL has provided a simplified way to perform scalable parallel I/O on the full range of scientific datasets. This trend has held true for our experiements on GPFS (the Intrepid machine) and Lustre (the JaguarXT4 machine), two of the most popular parallel file systems used in scientific data analysis.

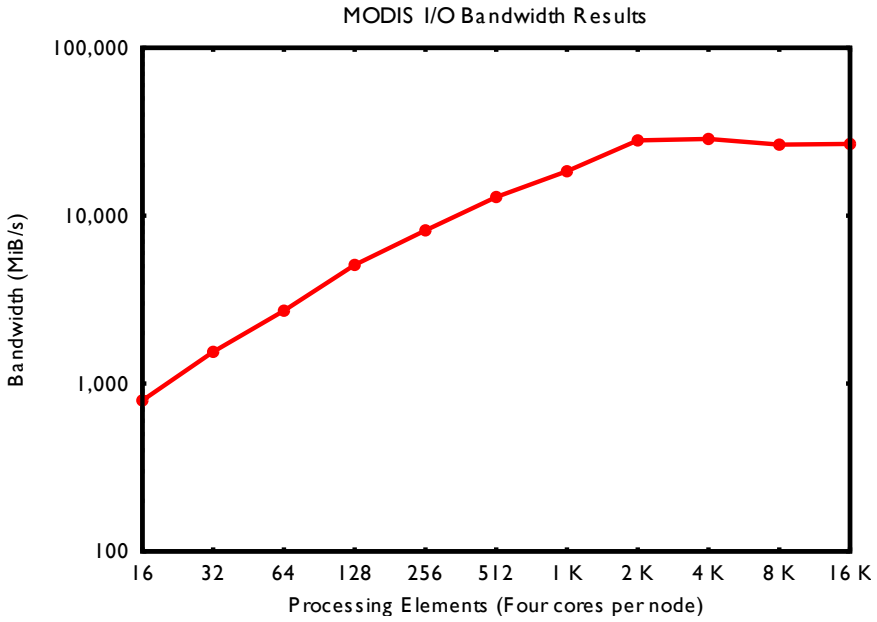


Figure 3.4: I/O bandwidth results (log-log scale) for utilizing the Block I/O Layer on the JaguarXT4 machine with the 1.1 TB MODIS dataset. Results are averaged across five runs.

By encapsulating advanced multi-file I/O routines under a design pattern, BIL has allowed my parallel architecture to achieve substantial I/O bandwidth rates in many application examples. In most cases, BIL significantly cuts down on length of code when compared to handling the many different cases for scientific data storage formats and partitioning patterns. It also transparently provides superior scalability when compared to using typical single file approaches to performing I/O.

In this dissertation, BIL is used as the underlying I/O component for each of the higher level components of my architecture (Sections 4 and 5).

Chapter 4

Simplifying Local Domain and Data-Parallel Analysis

To understand the underlying structures and relationships of variables in datasets through space and time, it is often necessary to analyze local features throughout the full spatiotemporal extent of a dataset. Local features can be described as features which only require access to a small static region surrounding a point in a dataset. The most common example of this is the analysis of a temporal trend of a value at a certain point in a dataset. Another example is the analysis of gradients of a local area.

The need for scalable methods to perform such full-range analysis tasks is growing as scientific simulations produce datasets ranging to the terascale and beyond. This need is particularly acute as visualization applications, especially those that handle large-scale time-varying data, are increasingly being dominated by I/O, data shuffling, and load imbalance to a degree that impedes their practical use.

In this section, my collaborators an I developed a system known as the Scalable Query Interface (SQI). SQI closely integrates techniques in parallel I/O with concepts from parallel query-driven visualization. Paired with scalable load balancing and large-scale parallel sorting, this system has contributed a novel method in performing and simplifying numerous types of local analysis in scientific datasets.

SQI has also been elegantly wrapped under a succinct design pattern, which allows for minimal parallel programming overhead. In the following, I detail the components of the system, the design under which they operate, and provide microbenchmarks of various components. An application example of this system is provided in Section 6 with more timing results.

4.1 A Scalable System for Querying and Feature Extraction

Figure 4.1 illustrates the overall architecture of SQI. The illustration starts from the beginning of data creation by either a simulation or data collection instrument. When data is initially produced, it is stored in a format native to the application, and it is typically striped across the parallel file system for higher access bandwidth. SQI utilizes BIL for its I/O. Our system reads and distributes data in parallel to prepare it into a queriable form with maximal runtime load balance. The relevant data is then queried and spatial or temporal analysis may occur by sorting the data in the proper ordering.

SQI abstracts many complex parallel details away from the user and provides an elegant interface for data-parallel filtering and analysis. Similar to BIL, SQI is based around two functions: SQI_Query() and SQI_Sort(). SQI_Query() is passed ranges of data of interest to the user. For example, the user may be interested in vegetation indices between a certain range that only exist within a certain spatial location. When SQI_Query() is called, a buffer of matching elements is returned. SQI_Sort() can then arrange data elements in a temporal or spatial sequence that is amenable to the task at hand.

Queries are used as the basis for data extraction. The reason for this is twofold. First, query-driven methods rely on translating a user interest into a compound Boolean range query. In doing so, it offers a natural extension of feature extraction capabilities in multivariate data. Second, regardless of the underlying methods, such as a distributed

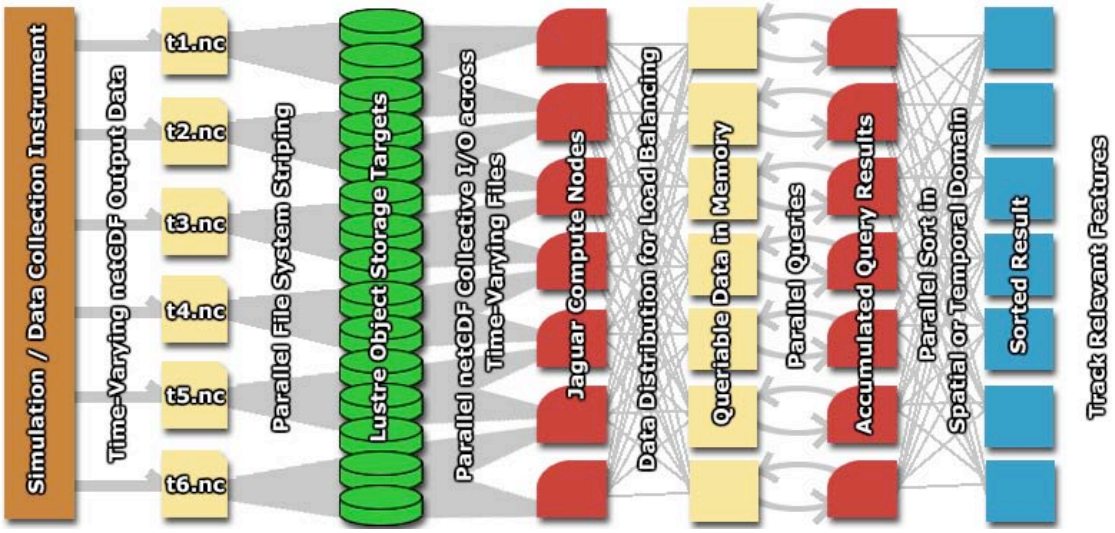


Figure 4.1: An overview of the components of the Scalable Query Interface.

M-ary search tree [Glatter et al. 2006], bitmap indexing [Stockinger et al. 2005b], or bin-hashing, query-driven visualization has been shown to accelerate contour extraction in large datasets.

As mentioned earlier, sorting is used to group data together for analysis. This design decision is quite similar to the design of MapReduce. In MapReduce, analysis occurs on objects that are grouped together by keys. In contrast with MapReduce, a parallel sort in any order is more general for certain analysis tasks that have no unique keys. One example is sorting values in viewing-depth order for rendering purposes.

The components of the system are described in the following, along with microbenchmarks of their performance. Since BIL was already covered, the I/O component of SQI is not discussed.

4.2 Load-Balanced Parallel Querying

As mentioned in Section 2, [Glatter et al. 2006] showed how an M-ary search tree structure could be used to accelerate querying of scientific data. The authors also showed that queries could be load balanced by distributing data on the granularity of single voxels.

The methods in [Glatter et al. 2006], however, were only scaled up to 40 servers, studied only one approach of load balancing, and used a server-client method that was inherently bottlenecked by one client collecting queried data over a network. The following discusses our methods to extend these concepts on large systems and to further improve the scalability and timing of our query-driven data extraction.

For fastest querying and sorting rates, it would be ideal to have equal amounts of returned data on every process for all the queries issued. Although it is impossible to know which queries will be issued, it has been shown that distributing the data in a spatially preserving manner across all the processes results in near-optimal load balance regardless of the query [Glatter et al. 2006]. However, distributing data in [Glatter et al. 2006] involved costly computation of Hilbert indices while doing a sort of the entire dataset.

It is then a question of trade-off between the overhead to distribute data after I/O versus the overhead of a large number of runtime queries. To find a practical optimum in this trade-off, we studied the design in [Glatter et al. 2006] plus four alternative designs, each incurring contrasting overheads of distribution and runtime querying. The starting point of all designs is the same: each node has read in contiguous segments of data from disparate netCDF files. In the following, we describe the load balancing schemes. We refer to a data point being distributed as an *item*, which is the multivariate tuple on each (x, y, t) location in the MODIS dataset.

Hilbert-Order: The original design in [Glatter et al. 2006]. The items are globally sorted by their computed indices along the Hilbert space-filling curve, and then distributed by a round robin assignment to all processes.

Z-Order: The same as Hilbert-Order, except computing indices along the Z space-filling curve.

Round Robin: Each item is distributed by a round robin assignment to all processes.

Random: The items are randomly shuffled locally and then divided into N chunks, where N is the number of processes. Each process P then gathers the P^{th} chunk from each process. This entire method is performed twice.

None: No data distribution.

4.3 Load Balancing Tests and Results

We tested these load balancing schemes to arrive at quantitative conclusions on which ones are best for given applications. We assessed these techniques based on two criteria: the cost associated with distributing the data, and the average time spent querying.

The cost of distributing the data is simply the distribution time. The average time spent issuing a query is calculated by taking the maximum query time of the individual processes, and then averaging it over all the queries. Doing this provides a measurement of how load imbalance in the searching phase will affect the overall parallel querying rates.

We tested the load balancing schemes by performing two separate tests that issued 1,000 randomly generated queries, each with the same random seed. The first test randomly restricted the time and spatial dimensions, and the second test randomly restricted the variable ranges. These tests used a $\approx$100 GB subset of the MODIS dataset of 40 timesteps, and the final results were averaged. The items returned from the individual queries ranged from 0.001% to 20% of the dataset.

The first experiment calculated the overhead times of distributing the data for the load balancing schemes. The Hilbert-Order and the Z-Order schemes required a parallel sort of the dataset, and the parallel sample sorting algorithm discussed in Section 4.4 is used to do this. The timing results are shown in Figure 4.2. The Hilbert-Order and Z-Order schemes scaled linearly. The Random and Round Robin schemes almost scaled linearly, but it is not expected for them to have scaled linearly since they are already under one second at 4K cores. The Hilbert-Order scheme incurred the most overhead because of the time involved in computing and sorting the Hilbert indices. The Z-Order indices were easier to compute, thus the scheme had a smaller overhead than the Hilbert-Order scheme. The Round Robin and Random distributions were almost a factor of 10 faster at all scales when compared to the distributions that require sorting.

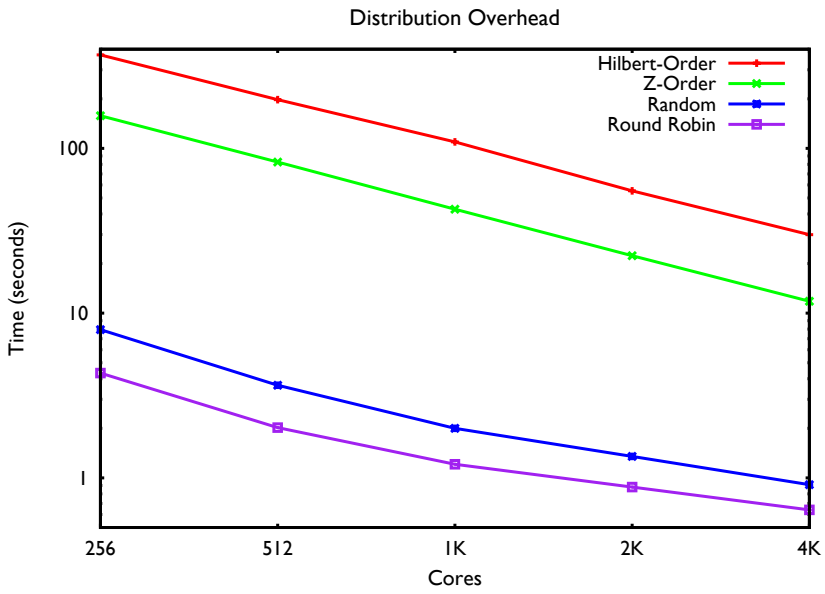


Figure 4.2: The overhead times for the load balancing schemes. Time is shown on logarithmic scale.

To further assess if these distribution costs are outweighed by the benefits of faster querying, we computed the average time per query. The results, displayed in Figure 4.3, showed us that the Hilbert-Order scheme is never the optimal scheme to use, because the overhead time was always the slowest and the resulting querying times were never the fastest at any scale. In all of the cases, the Z-Order distribution gave the fastest query times, and the Random distribution followed closely.

From these results, we conclude that the Random scheme is the best to use for general applications that will not be issuing queries on the order of the thousands. The Z-Order scheme showed the fastest querying rates, but the cost associated with the distribution will only be outweighed by the querying times when very many queries are issued. Even though the Round Robin scheme showed a very small overhead, the resulting time spent querying was costly as the number of queries increased. We believe this is because the Round Robin scheme is highly dependent on the layout of data in memory after the I/O step. As a comparison, applying no load balancing scheme to the dataset resulted in noticeably poorer querying rates.

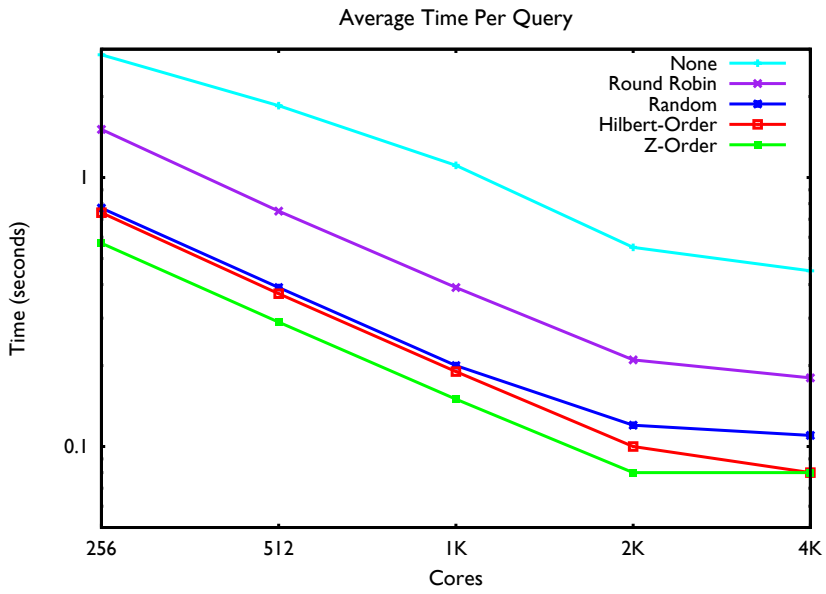


Figure 4.3: The average time per query for the load balancing schemes. Time is shown on logarithmic scale.

Although randomness is not guaranteed to preserve spatial locality as mentioned in [Glatter et al. 2006], the costs of distributing the data along with the resulting performance outweighed the Hilbert-Order scheme that was used in [Glatter et al. 2006]. We chose to use the Random scheme for our architecture because of the significantly small overhead and fast querying rates.

4.4 A Modified Parallel Sample Sorting Algorithm

The authors in [Blelloch et al. 1998] showed that the parallel sample sorting algorithm performed best for large datasets. The reason for this is because of its ability to only use one large data shuffling method in the form of an MPI_Alltoallv call (discussed in Section 2.3.4).

One of the downfalls of the parallel sample sorting algorithm is that the overhead for sorting the relatively small amount of samples increases significantly at large scale. This is due to the fact that the parallel sample sort algorithm must use a less efficient algorithm (e.g. serial or parallel merge sort) to sort the samples. In [Blelloch et al. 1998], the

authors gathered the samples from each process to one process, sorted them serially, and then broadcasted the resulting splitter elements to each process. Although this has small overhead at 1 K processes, the overhead increases at scale. For example, with every process contributing 64 random elements to the sampling at 32 K processes, one process would have to sort over two million sampled elements.

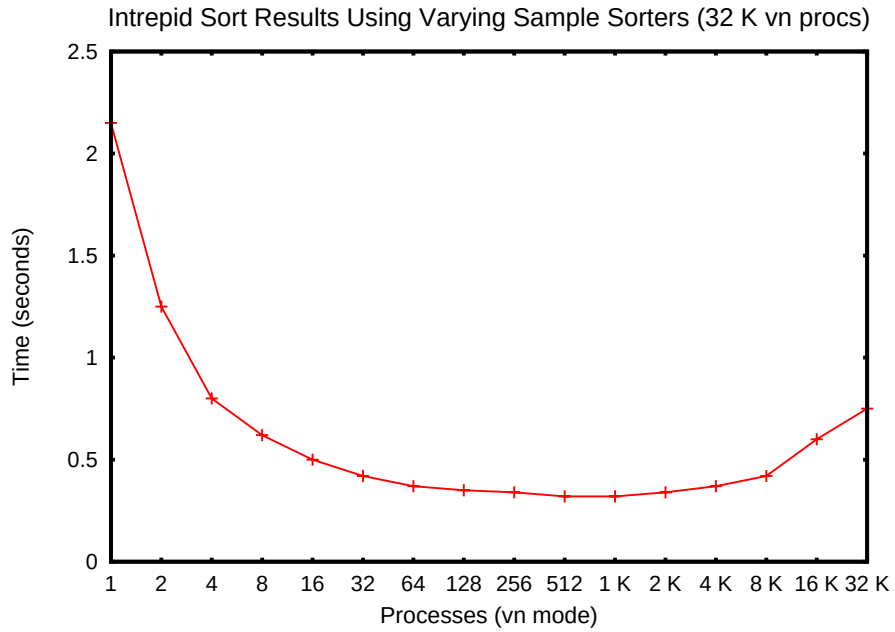
It is then a question of if it is applicable to utilize another parallel sorting algorithm to sort the samples. Parallel merge sort is an efficient alternative with many optimizations that can keep all of the items distributed. One optimization is transferring minimum and maximum items before the merge step to minimize communication.

It is still, however, inefficient to sort 64 elements per process at extremely large scale since the problem size is not large. To address this issue, I used a configurable parameter in my implementation that specifies the subset of sample sorters to use. The samples are gathered to this subset of sorters, an optimized and distributed parallel marge sort is performed, and then the splitter keys are broacasted to all processes. I have found this algorithm to be the most efficient for parallel sample sorting at large scale, and I evaluate its efficiency with varying numbers of sample sorters in the next section.

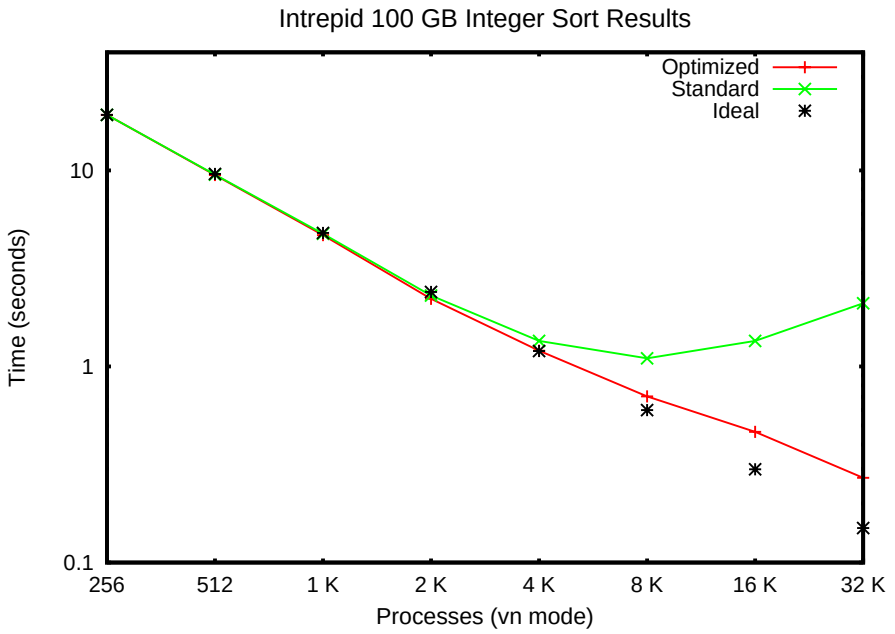
4.5 Parallel Sorting Benchmark

I benchmarked my parallel sorting implementation on Intrepid using synthetic data. In all tests, 10 GB of random integers were generated and distributed evenly across PEs. The results are shown in Figure 4.4.

In my first experiment, I held the number of total processes constant. For each test, I incremented the number of sample sorters from one (which is the classical parallel sample sorting implementation) all the way to the total number of processes. Figure 4.4a shows this experiment with 32 K processes. The inefficiencies of using only one process as well as every process to sort the samples is shown. For the case of 32 K processes, using a range of 256 - 2 K sample sorters provided best results. In fact, in all of the other experiments



(a) Varying Sample Sorters at 32 K PEs



(b) Standard vs. Optimized Approach

Figure 4.4: Results for sorting 10 GB of integers on Intrepid across varying PEs.

at different process counts, I noticed a similar trend. I concluded that using 1 / 32 of the entire process to sort the samples provided optimal results in almost all cases.

I used this parameter for determining the optimal number of sample sorters in my next test, and I compare the overal sorting times to the classic parallel sample sort algorithm. The green line in Figure 4.4b shows the classic implementation. Although this method scales well to 2 K PEs, performance significantly diminishes when going to tens of thousands of PEs. My approach, which distributes samples to a subset of PEs and then performs distributed merge sort, is shown as the red line in Figure 4.4b. This approach scales much better at large PE counts, and continues to have improved performance up to 32 K cores.

4.6 Closing Remarks about SQI

SQI provides a novel method to tie together query-driven visualization techniques with data-parallel analysis components. As I will show in Section 6, SQI has allowed scientists to perform complex full-range analysis on terascale datasets in interactive scientific analysis settings.

Although many different analysis tasks fit SQI's design pattern, there are also different application requirements that do not readily fit into its model. One example is global analysis, in which a process may need access to elements in the entire domain of a dataset. The next section discusses these types of problems in more detail and provides a more flexible parallel programming model for these types of problems.

Chapter 5

Simplifying Global Domain and Task-Parallel Analysis

Domain traversal is the ordered flow of information through a data domain and the associated processing that accompanies it. It is a series of relatively short-range and interleaved communication / computation updates that ultimately results in a quantity computed along a spatially- or time-varying span. When the domain is partitioned among processing elements in a distributed-memory architecture, domain traversal involves a large number of information exchanges among nearby subdomains accompanied by local processing of information prior to, during, and after those exchanges. Examples include computing advection in flow visualization; and global illumination, particle systems, scattering, and multiple scattering in volume visualization.

A capability to flexibly analyze scientific data using parallel domain traversal at scale is much needed but still fundamentally new to many application scientists. For example, in atmospheric science, the planet-wide multi-physics models are becoming very complex. Yet, to properly evaluate the significance of the global and regional change of any given variable, one must be able to identify impacts outside the immediate source region. One example is to quantify transport mechanisms of climate models and associated interactions

for CO_2 emitted into the atmosphere at specific locations in the global carbon cycle. Uncertainty quantification of ensemble runs is another example.

Parallelization of domain traversal techniques across distributed-memory architectures is very challenging in general, especially when the traversal is data dependent. For example, numerical integration in flow advection depends on the result of the previous integration step. Such data dependency can make task parallelism the only option for parallel acceleration; however, at large scale (e.g. tens of thousands of processes), when each particle trace could potentially traverse through the entire domain, one must deal with complexities of managing dynamic parallelism of communication, work assignment, and load balancing.

Here, we provide application scientists with a simplified mode of domain-traversal analysis in a general environment that transparently delivers superior scalability. We call our system *DStep*. In particular, we note DStep's novel ability to abstract and utilize asynchronous communication. Since today's HPC machines commonly offer multiple network connections per node paired with direct memory access (DMA), asynchronous communication is a viable strategy for hiding transfer time. Asynchronous exchanges, however, can easily congest a network at large process counts. We found that efficient buffer management paired with a two-tiered communication strategy enabled DStep to efficiently overlap communication and computation at large scale. Using fieldline tracing as a test of scalability, DStep can efficiently handle over 40 million particles on 65,536 cores, a problem size over two orders of magnitude larger than recent studies in 2009 [Pugmire et al. 2009] and 2011 [Peterka et al. 2011b].

Along with abstracting complicated I/O and communication management, DStep also provides a greatly simplified programming environment that shares similar qualities as those of MapReduce [Dean and Ghemawat 2004] for text processing and Pregel [Malewicz et al. 2010] for graph processing.

The simplicity of our approach paired with the scalable back end has allowed us to write succinct and expressive custom data analysis applications using DStep. As a result, atmospheric scientists have a new way to evaluate the longitudinal dependence of

inter-hemispheric transport; for example, to support CO_2 source apportionment according to movement patterns of specific CO_2 molecules between the Northern and Southern Hemispheres (Section 6.3). Furthermore, they also have innovative methods for assessing internal-model variability. We have experimented with these creative analyses on terascale atmospheric simulation data – an ability not previously reported in the literature. We detail our approach in the following.

5.1 DStep – A Model and API for Simplifying Domain Traversal

Our analysis approach and implementation, *DStep*, is built primarily on two functions: dstep() and reduce(). The dstep() function is passed an arbitrary point in a spatiotemporal domain. Given this point, steppers (those executing the dstep() function) have immediate access to a localized block of the domain which surrounds the point. During the global execution of all steppers (the traversal phase), each stepper has the ability to implicitly communicate with one another by posting generic data to a point in the domain. In contrast to MPI, where processes communicate to others based on rank, this abstraction is more intuitive for domain traversal tasks, and it also allows for flexible integration into a serial programming environment. In other words, DStep programs do not have awareness of other processes.

The reduce() function is identical to that of MapReduce. We found that after the traversal phase, data-parallel operations were important for many of our collaborators' needs. For example, one operation is reducing lat-lon points to compute monthly averages and vertical distribution statistics.

The API of DStep promotes a similar design to that of MapReduce. Users define functions that take arbitrary data as input, and data movement is guided by *emit* functions. Two functions are defined by the user:

dstep(point, block, user_data) – Takes a point tuple from the dataset ($[x,y,z,t]$), the enclosing block subdomain, and user data associated with the point.

reduce(key, user_data[ ]) – Takes a key and list of associated user data.

Three functions are called by the user program:

emit_dstep(point, user_data) – Takes a point tuple belonging to any part of the domain and arbitrary user data. The data is sent to the proper part of the domain, where it may continue traversal.

emit_reduce(key, user_data) – Takes a key and associated user data. All user_data values associated with a key are sent to a reducer.

emit_write(user_data) – Takes arbitrary user data, which is stored to disk.

Using this API, the dstep() function of our fieldline tracing problem shown in Figure 2.1 could be written as:

```
function DSTEP(point, block, user_data)
    if user_data.empty() then
        user_data.key = point                          ▷ Key equals trace start
        user_data.trace_size = 0                       ▷ Initialize trace size
    end if
    Fieldline trace                                    ▷ Initialize partial fieldline trace
    while user_data.trace_size < MAXTRACESIZE do
        trace.append(point)
        point = RK4(point, block)                      ▷ Runge-Kutta
        user_data.trace_size++
        if !block.contains(point) then
            ▷ Post new point when going out of bounds
            emit_dstep(point, user_data)
            break
```

```
        end if
    end while
    emit_reduce(user_data.key, trace)                    ▷ Partial result
end function
```

In this example, fieldlines are computed using fourth-order Runge-Kutta integration. During tracing, steppers emit points to other subdomains when tracing goes out of bounds, and they also emit partial fieldlines for reduce(). The reduce() function would then be responsible for merging partial fieldlines (as shown in Figure 2.1).

The *user_data* variable allows users to pass around arbitrary data for analysis purposes. In fact – although not recommended because of performance reasons – the user could also pass the entire computed fieldline to other steppers instead of reducing partial results.

5.2 DStep Application Instantiation

There are three aspects to instantiating a DStep application, each of which can be controlled programmatically or by XML configuration files. The first is data input. In the spirit of designs such as MapReduce and NoSQL (i.e. avoiding data reorganization before analysis), DStep manages input of native application datasets. This ability has been crucial to our user and application needs, which (in our case) has allowed us to abstract the management of thousands of multi-variable netCDF files. Furthermore, we have observed that our scientists are often only interested in analyzing subsets of their datasets at a given time. Because of this observation, which is common in many settings [Stockinger et al. 2005c], we allow users to specify input as compound range queries. For example, users may specify a range query of $[0 \leq X \leq 100]\&\&[0.2 \leq CO_2 \leq 0.4]$ to filter all of the points that have an X and CO_2 value within the given range. This approach integrates elegantly into our design, and each point matching the user query is simply passed as the point parameter to the dstep() function.

The second aspect is data output. As we will explain later, users simply specify a directory for output, and DStep utilizes a custom high-level format which can easily be read in parallel by another DStep application or parsed serially into other scientific formats.

The final aspect is job configuration. Although we ultimately want to hide partitioning and parallel processing details, we allow users to specify job configurations for obtaining better performance on different architectures. One parameter of the job configuration is the partitioning granularity. Users can specify how many blocks should be assigned to each of the workers, and our system handles round-robin distribution of the blocks. For data replication, we allow users to specify an *elastic* ghost size. Instead of explicitly stating a ghost size for each block, DStep will automatically adjust the ghost size to fit within a specified memory threshold.

5.3 DStep Data Flow

Given our API and the design of application instantiation, we illustrate the entire data flow of DStep in Figure 5.1. A volumetric scientific dataset is partitioned into blocks, which are assigned to steppers. Input to dstep() comes from querying these blocks, and it also comes from other steppers that call emit_dstep(). Data from emit_reduce() are shuffled to reducers. Similar to [Tu et al. 2008], we also allow reducers to perform multiple reduction steps for more sophisticated analysis. Reducers or steppers may store data with emit_write().

The most difficult data flow complexity is introduced by emit_dstep(). Users can induce voluminous and sporadic communication loads with this function call. We have designed a two-tiered communication architecture to manage this intricate data flow. We overview the general implementation surrounding this data flow (the DStep Runtime) in the following, and we provide comprehensive details of our communication strategy.

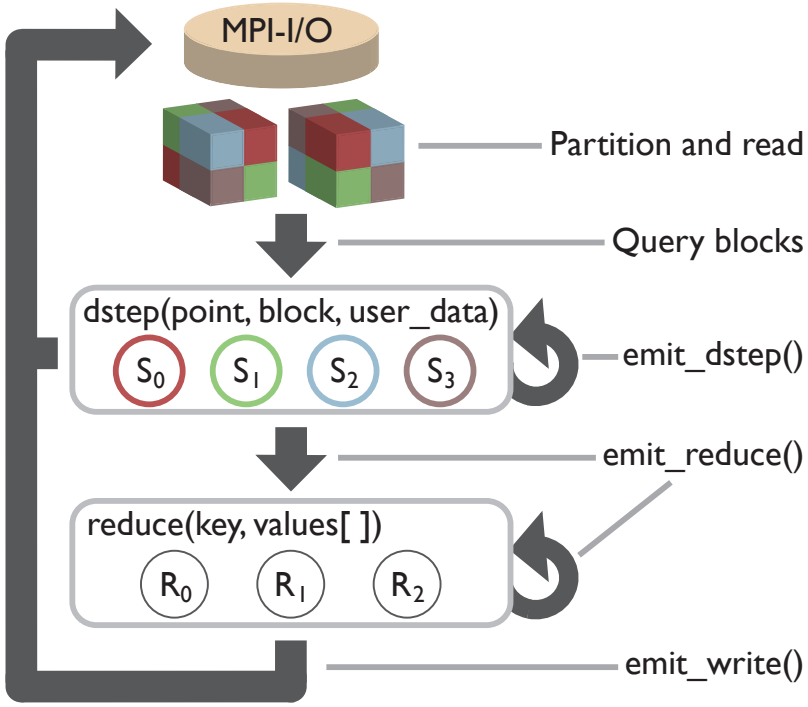


Figure 5.1: Data flow using DStep. Data movement, primarily directed by emit functions, is shown by arrows. Computational functions and associated workers are enclosed in blocks.

5.4 The DStep Runtime

The DStep Runtime is a C++ hybrid threaded/MPI execution system that is designed to perform complex domain traversal tasks on large scientific datasets directly after simulation. In contrast with industrial scenarios that leverage large commodity clusters for batch jobs [Dean and Ghemawat 2004], we designed our implementation to leverage HPC architectures for scientific analysis scenarios.

5.4.1 Software Architecture

A general overview of the DStep software stack is illustrated in Figure 5.2. Components are shown in a bottom-up fashion with open-source tools in gray and our components in other colors.

DStep utilizes up to four different types of workers per MPI process: steppers, reducers, communicators, and writers. The Worker Superclass specifies three virtual functions for work scheduling, which manage the inherited incoming and outgoing work buffers. This management is described in detail in the next section. The general responsibilities of the workers are as follows.

Stepper – Steppers own one or more blocks from a static round-robin distribution of the domain. They are responsible for reading the blocks, processing the user's query on each block, sending the queried input to dstep(), and sending any input from other steppers to dstep().

Reducer – Reducers own a map of keys that have an associated array of values. They are responsible for sending this input to reduce() when the traversal phase has finished.

Communicator – As we will describe later, communication happens in a group-based manner. The sole responsibility of communicators is to act as masters of a worker group, managing worker/application state and routing any long-range messages to other groups.

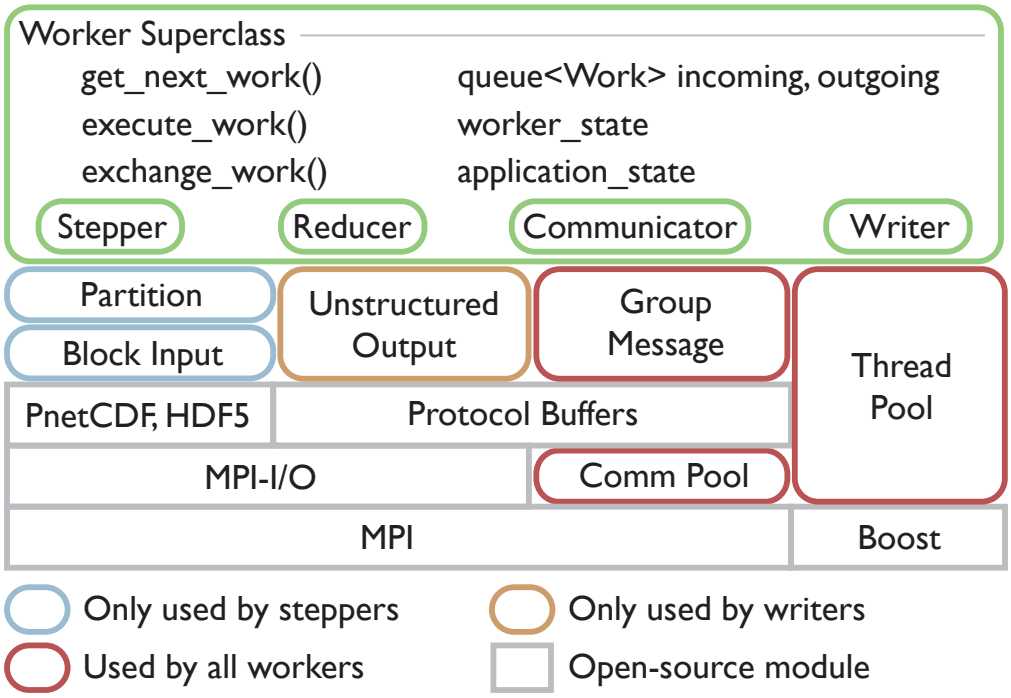


Figure 5.2: The software design of DStep is shown, with open-source modules in gray and custom components in other colors.

Writer – Writers manage data sent to the emit_write() function and use a fixed functionality for data output. We note that we also provide the user to override a write() function for sending output to other channels such as sockets.

5.4.2 Resource and I/O Management

All workers have access to various limited system resources. The first is a group messaging module, which provides an abstraction for asynchronous group-based communication. It is built on top of a communication pool, which manages pending asynchronous communication requests to other group members. If too many requests are pending, i.e. if MPI_Test() returns false for all MPI_Requests in the pool, the module buffers group messages until resources are available. The module also manages the complex unstructured messages from workers, which can include arbitrary user data and state information (such as how much work has been initialized and finished). We use Google's Protocol Buffer

library for packing and unpacking these messages. Protocol Buffers allow users to define a message that can be compiled into a dynamic and serializable C++ class. In contrast to a textual format such as XML, Protocol Buffers pack data into a condensed binary representation.

Another resource is a thread pool, which utilizes the Boost thread library [*]. Workers send work to this thread pool, which manages execution across pre-spawned threads.

We perform I/O with two in-house solutions. For reading data from a block-based partition, we use the Block I/O Layer (BIL) [Kendall et al. 2011a]. BIL provides an abstraction for reading multi-file and multi-variable datasets in various formats (raw, netCDF, and HDF). It does this by allowing processes to add as many blocks from as many files as needed, and then collectively operating on the entire set. In the implementation, BIL aggregates the requests across files, schedules reading from multiple files at once, and then performs a second exchange of data back to the requesting processes. Depending on the nature of the requests, BIL can utilize I/O bandwidth more efficiently than standard single-file collective access. Furthermore, block-based requests which have ghost regions do not suffer from any redundant I/O, and data replication is instead performed in memory. This strategy has allowed us to efficiently operate on native simulation datasets, and we direct the reader to [Kendall et al. 2011a] for a more elaborate explanation of implementation details.

For writing data, we use an unstructured output module that also utilizes Google Protocol Buffers for serializing dynamic user data. Each writer possesses a separate file and header, to which data are appended in chunk sizes equal to file system stripe size. Utilizing multiple file output is beneficial for two primary reasons. First, it can alleviate lock contention issues that arise in parallel file system writes [Gao et al. 2009]. Second, in a similar method to [Lofstead et al. 2010], we found that pinning files to storage targets in a round-robin manner provided an efficient method for managing variability in writes. Since users define their output in Protocol Buffer format, the data can easily be parsed afterwards serially or in parallel.

[*]http://www.boost.org

5.4.3 Two-Tiered Data Management

We introduce a novel two-tiered strategy for managing data flow during execution and exploiting asynchronous communication. The bottom tier of the architecture consists of groups of workers which can communicate asynchronously. The top tier includes processes (i.e. communicators) that are dedicated to routing any inter-group messages. Configurable worker groups paired with buffered work management form the basis of this strategy.

Worker Groups Worker groups define the placement of workers to processing elements, and they also restrict asynchronous exchanges to the defined groups. An example worker group is illustrated in Figure 5.3, which shows a user-specified XML configuration along with its associated worker assignment on quad-core processors. The configuration, which specifies eight workers, is replicated across the total amount of processing elements (in this example, 16 cores).

The primary advantage of worker groups is the ability to perform asynchronous communication without congesting the network. Workers can only post asynchronous messages to others in the same group, and communicators are responsible for routing any inter-group messages. If communicators reside on separate processing elements, routing will ideally occur simultaneously while others perform computation. Furthermore, the communicators can also manage worker state (e.g. the number of initialized and completed work elements) and manage global application state (e.g. determine when the traversal and reduction phases complete).

Worker groups offer several unique advantages for managing large communication loads on HPC architectures. First, the bottleneck of having one master manage worker state is alleviated. Second, HPC architectures often organize processors into racks and cabinets, each of which have larger latencies to one another. Configurable worker groups allow users to localize asynchronous communication, from the shared memory on a node to the nodes of an entire rack, and allow communicators to manage long-range messages.

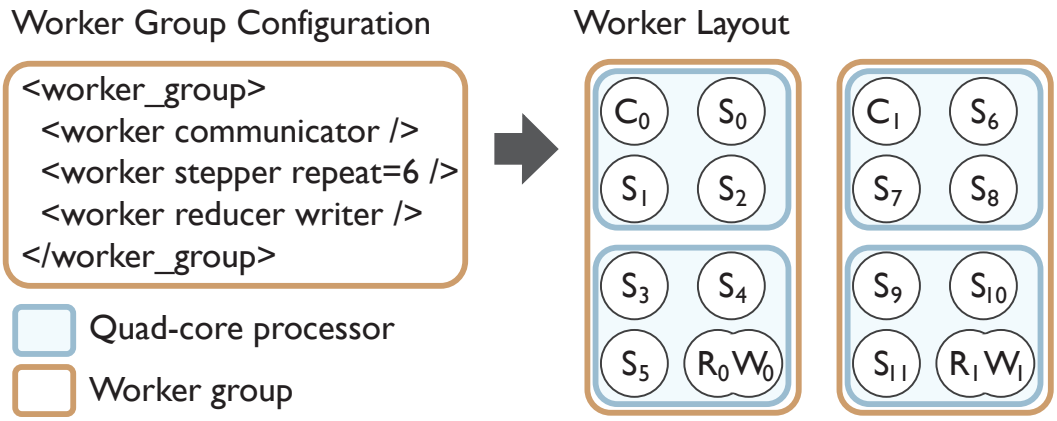


Figure 5.3: Example worker group configuration and layout. The worker group configuration specifies eight different workers, which are replicated across pairs of quad-core processors.

Buffered Work Management Workers buffer work to more efficiently overlap communication with computation. Given ε, which defines the maximum amount of work elements to be executed in a given epoch, the DStep Runtime executes workers as follows:

repeat

for all w in workers **do**

w.get_next_work(ε) ▷ *Epoch start*

w.execute_work()

w.exchange_work() ▷ *Epoch finish*

end for

▷ *Check global application state for termination*

until Runtime::application_finished()

Each worker implements three functions: get_next_work(ε), execute_work(), and exchange_work(). These functions are responsible for managing workers' incoming and outgoing work queues. The actions taken by the workers during these functions are as follows:

get_next_work(ε) – Workers gather up to ε work elements for execution by popping data from their incoming work queues. If steppers have less than ε elements, they process the next $\varepsilon - incoming\_work.size()$ elements of the user-defined query.

execute_work() – Steppers, reducers, and writers pass work elements to user-defined or fixed functions. If any emit functions are called by the user, the DStep Runtime places elements in workers' outgoing work queues. Communicators collectively route any inter-group messages, update global application state, and place routed work elements in their outgoing work queues.

exchange_work() – Based on the destinations of work elements in outgoing work queues, all workers post asynchronous sends to the other appropriate group workers. If resources are not available, i.e. if the communication pool has too many pending requests, workers simply retain work in their outgoing work queues. If any worker has new state information, it is posted to the communicator of the group. Similarly, if the communicator has new application state information, it is posted to all of the group workers. After sends are posted, workers poll and add any incoming messages to their incoming work queue.

Execution is designed such that workers sufficiently overlap communication and computation without overloading the network. Furthermore, ε is chosen to be sufficiently large ($\approx$250) such that enough computation occurs while incoming asynchronous messages are buffered by the network.

We provide a thorough example of two epochs of execution in Figure 5.4. For simplicity, we only use two groups with communicators and steppers, and we show execution of functions in a synchronous manner. We note, however, that synchronization only happens among communicators during execute_work().

The user starts by executing their application with the DStep environment and providing a query as input. In the first epoch, steppers' incoming work queues are filled with ε query results. The incoming work is then sent to dstep(), which calls emit_dstep() for each element in this example. Work elements are placed in steppers' outgoing work queues and

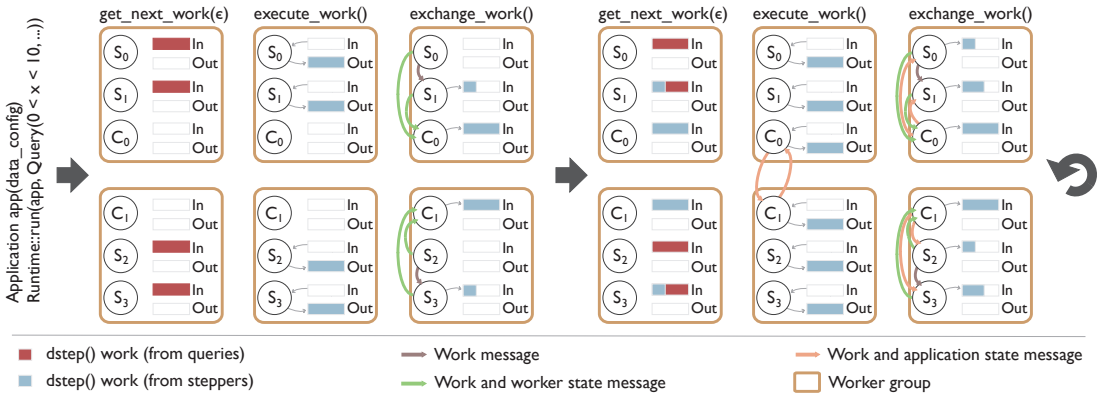


Figure 5.4: Example of two epochs executed by two worker groups that contain steppers and communicators. Steppers initially fill their incoming work queues with points that match the user query, and work progresses through the system. The second execute_work() function shows the primary overlap of communication with computation - communicators are performing long-range communication while steppers are performing work.

then exchanged. In the example, S_0 and S_2 both have work elements that need to be sent to steppers outside the group (which is posted to their respective communicators) and inside the group (which is posted directly to them).

In the second epoch, querying happens in the same manner, with S_1 and S_3 appending queried elements to queues that already include incoming work from others. Steppers perform execution similar to the previous epoch, and communicators exchange work and update the application state. Exchange occurs similar to the previous epoch, except communicators now post any new application state information and inter-group work messages from the previous epoch.

The entire traversal phase completes when the number of initialized dstep() tasks equal the number completed. The reducers, although not illustrated in our example, would then be able to execute work elements that were added to their incoming work queues from emit_reduce() calls. In a related fashion to steppers, reducers also maintain state information since they have the ability to proceed through multiple reduction phases.

In contrast to the synchronous particle tracing strategy presented in [Peterka et al. 2011b] and the strategy that used a single master in [Pugmire et al. 2009], we have found this hybrid and highly asynchronous strategy to be beneficial to our data demands. In

Section 6, we demonstrate the performance of the DStep Runtime in the context of our driving application – terascale atmospheric analysis.

5.5 Parallel Particle Tracing Benchmarking Test

The authors in [Peterka et al. 2011b] provide benchmark results of parallel particle tracing on various test datasets and tuning parameters. The library used in their study was OSUFlow, a parallel particle tracing library originally developed by the Ohio State University. Here, we utilize DStep to perform similar benchmarking tests and also provide comparisons against OSUFlow.

5.5.1 Particle Tracing in OSUFlow

Described in [Peterka et al. 2011b], the particle tracing algorithm in OSUFlow executes by first statically splitting the domain into blocks. Multiple blocks are assigned to processes in a round-robin style for better computational load balance. Similar to DStep, OSUFlow uses the Block I/O Layer for parallel I/O.

OSUFlow first places particles into blocks with a random sampling. The library then advects the particles until they reach block boundaries. Particles that reach block boundaries are packed into a binary array of floats and then synchronously communicated to processes which own neighboring blocks. The algorithm then repeats.

The authors showed that the synchronous nature of the algorithm was inefficient when particles were stuck in vortices. This is due to the fact that when particles are stuck in high amounts of local computation, communication from other processes blocks and subsequently slows down the entire algorithm. By simply terminating particles stuck in vortices, the authors showed that scalability and overall timing improved greatly.

The authors have recently added unpublished optimizations that more efficiently exploit asychronous communication. In their latest approach, they post asynchronous particle messages during nearest-neighborhood exchange. Instead of waiting for all of the particles

to be sent, they only wait for a threshold of particles to be sent before continuing computation. This in turn allows them to "dial down" the amount of synchronization and to continue to perform work while particles are being exchanged.

5.5.2 Nek5000 Benchmark Comparison Against OSUFlow

We have conducted a benchmark test from [Peterka et al. 2011b] using DStep and OSUFlow with the aforementioned optimizations. Below are various testing parameters used, as well as notable differences between the two approaches.

1. The nek5000 dataset (Section 2.1) is used, which is a static 2048^3 grid consisting of three floating-point velocity variables ($\approx$100 GB). Renderings from [Peterka et al. 2011b, Pugmire et al. 2009] show a highly turbulent flow, which makes it a desirable test case for parallel flow tracing.

2. DStep and OSUFlow both use the same partitioning scheme. In this test, eight blocks per process are assigned in a round-robin fashion. Likewise, both approaches use the Block I/O Layer as a one-time read step after partitioning.

3. Since both approaches use a different output format, the timing of writing results are disregarded.

4. The largest disparity between the tests is that OSUFlow and DStep use different integration kernels. OSUFlow uses an adaptive-size fourth-order Runge-Kutta integration kernel that uses trilinear interpolation. Since the kernel is tied deeply into the library and its data structures, it was unfeasible to use the kernel in a DStep application. DStep utilizes a custom fourth-order Runge-Kutta integration kernel that uses a static step-size of one and trilinear interpolation. In both cases, integration is stopped after 1000 integration steps. It is still inconclusive if the resulting geometry is very similar, but comparisons of renderings from [Peterka et al. 2011b] show similar results.

5. Particles from OSUFlow are communicated as raw floating-point arrays which include a header (that indicates how many steps have been taken) along with payload of particles. DStep uses Protocol Buffers as a means to pack the same type of information. Protocol Buffers incur a small overhead of serializing and deserializing messages.

6. DStep uses a configuration of 15 steppers and one communicator per group. The dstep() function is the same as that from Section 5.1, except there is no reduce phase needed for this test since we are ignoring collection and writing of geometry.

7. In both tests, 262,144 particles were traced. DStep starts these particles at even spatial strides while OSUFlow places the seeds randomly at the same density for each block.

8. In the tests, DStep uses the elastic ghost size concept (discussed in Section 5.2) for memory replication.

Using these parameters, a test was conducted on Intrepid (Section 2.1) at varying scales. Since OSUFlow and DStep both use the same partitiong and I/O code, only the total amount of time spent in computation and communication is reported. The results are shown in Table 5.1. The left column shows the total computation / communication time for OSUFlow, the middle column shows the time for DStep without memory replication, and the right column shows the total time for DStep using memory replication.

One noticeable difference between OSUFlow and DStep without memory replication is that DStep begins to show worse scalability when going to 32 K processes. This is likely attributed to the large amount of communication overhead. When replicating memory, DStep keeps more of the computation local and is able to perform better at 32 K; however, it is still slightly slower than 16 K processes.

For OSUFlow, it is difficult to conclude if computation or communication is the bottleneck at smaller process counts. The OSUFlow results in Table 5.1 are about a two-fold peformance improvement than results reported in [Peterka et al. 2011b]. Since the

Processes	OSUFlow	DStep	DStep with Memory Replication
1 K	105.19	20.39	20.37
2 K	82.79	14.36	14.33
4 K	58.00	11.64	11.05
8 K	44.68	10.26	9.29
16 K	39.72	10.94	9.24
32 K	30.98	15.16	10.12

Table 5.1: Total amount of time spent in computation and communication for OSUFlow, DStep, and DStep with memory replication.

only change between these results is the communication algorithm, it is likely that further improvements to the communication algorithm may yield even better results.

One conclusion that can be made from these results is that the communication algorithm of OSUFlow has a better parallel efficiency than that of DStep. However, it must be noted that a slower integration kernel in DStep would likely give it the ability to overlap more communication and computation, which could ultimately provide better parallel efficiency.

Figure 5.5 provides a breakdown of communication and computation times in DStep with and without memory replication. In this figure, the communication times include the time spent waiting for computation to become available. The green line shows that the time spent communicating and waiting becomes much greater at large scale without memory replication. In fact, at 32 K process, the computation time also becomes higher. This can be attributed to the fact that breaking up the computation into smaller pieces results in more overhead in managing queues and serializing/deserializing protocol buffers. At this scale, it seems that the individual computation per process is so small that the overhead starts to become a significant part of the overall time.

With memory replication, the results show that more computation (in blue) is kept local and avoids the overhead of the DStep Runtime. Also, the amount of communication (in purple) is amortized significantly with less processes needing to communication data.

Although the primary purpose of DStep is to provide a simplified interface for large-scale data traversal problems, the results show that the raw performance of DStep is at least comparable, if not more efficient, to a state-of-the-art parallel particle tracing library. One

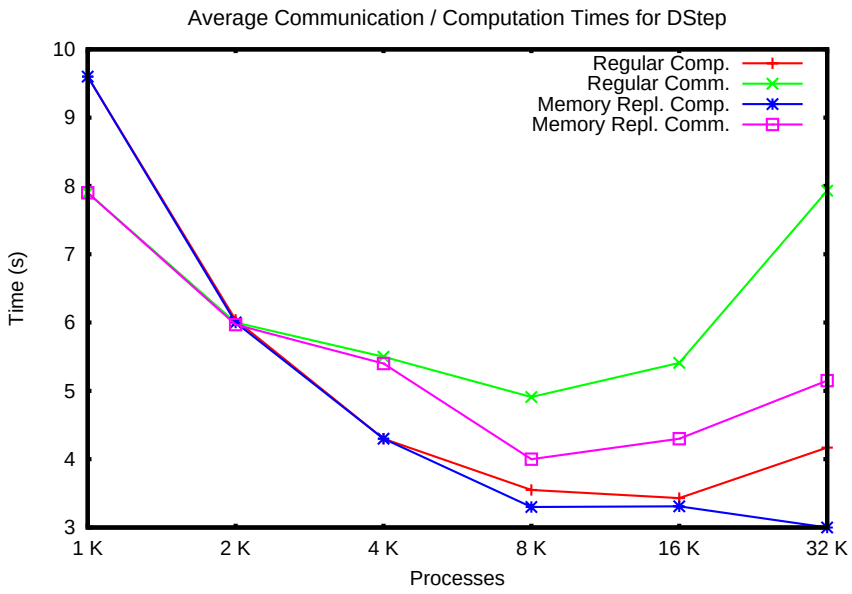


Figure 5.5: Communication and computation times for DStep with no memory replication and with memory replication. Communication times include the time spent waiting for computation to become available.

of the primary advantages is the ability to integrate custom kernels into DStep for a specific dataset, rather than using a general kernal present in a library. More tests will be needed to assess the communication strategies of the two methods, and this is currently out of the scope of this work.

Chapter 6

Enabling Capabilities in Computational Science

6.1 MODIS Analysis

One driving application of my architecture is to study observational data in NASA's MODIS database to discover multivariate climatic trends. The dataset used in our study consists of a 500 meter resolution sampling of North and South America, creating a 31,200 by 21,600 grid. The dataset is continuously updated, and we used 417 timesteps of 8 day intervals from February 2000 to February 2009. MODIS data is stored in various wavelength bands which may be used to compute other variables. By computing two variables that are related to our studies and storing them as short integers, the entire dataset totals to $\approx$1.1 TB. With a dataset of this magnitude, we have two primary goals: to provide visualization and analysis methods for cases when many of the timesteps of the dataset need to be loaded, and to deliver usability and near-interactive functionality specifically for application scientists.

We used our system to discover climatic trends in two variables from the MODIS dataset. Our approach has specific motivations in climate science, but can be applied to

a broad variety of application areas. The two problems we addressed in climate science are drought assessment and time-lag analysis.

6.1.1 Variables

The two variables we used are computed from the satellite bands of the MODIS dataset. The first variable is the Normalize Difference Vegetation Index (NDVI). This variable is computed with the red band (RED) and the near infrared band (NIR) with the following equation:

$$NDVI = \frac{RED - NIR}{RED + NIR} \tag{6.1}$$

NDVI measures the changes in chlorophyll content by the absorption of the visible red band and in spongy mesophyll by the reflected near infrared band. Higher NDVI values usually represent the greenness of the vegetation canopy [Gu et al. 2007].

The other variable we computed is the Normalized Difference Water Index (NDWI). This variable is computed using the red band (RED) and the short wave infrared band (SWIR) with the following equation:

$$NDWI = \frac{RED - SWIR}{RED + SWIR} \tag{6.2}$$

NDWI is a more recent satellite derived index that reflects changes in water content by the short wave infrared band [Gao 1996].

6.1.2 Drought Assessment

Drought is one of the most complicated and least understood of all natural hazards, and much research has gone into using satellite derived data as a means for drought assessment. NDVI and NDWI together have been used to monitor drought in several different studies [Gu et al. 2007, Wang et al. 2008, Liu and Wu 2008]. In particular, NDVI

and NDWI were combined in [Gu et al. 2007] to form a Normalized Difference Drought Index (NDDI) variable. NDDI is computed as:

$$NDDI = \frac{NDVI - NDWI}{NDVI + NDWI} \tag{6.3}$$

It was shown in [Gu et al. 2007] that NDDI is a more sensitive indicator of drought, especially during the summer months in the Central United States. In [Gu et al. 2007], they used NDVI and NDWI values to find drought in the Central Great Plains of the United States and then used the NDDI value to assess the severity of the drought. We used a similar approach and applied this concept to the entire dataset. By doing this, it allowed conclusions to be made if NDDI along with NDWI and NDVI are acceptable measures for drought. It is important to assess these variables to better understand their usefulness in drought monitoring and prediction. Based on these variables, we used five criteria to find periods of drought:

NDWI_thresh: To be considered as a location where drought is occurring, the location must have NDWI below this threshold.
NDVI_thresh: Similarly, it must have NDVI below this threshold to be considered as a location where drought is occurring.

NDDI_range: If the NDDI value is in this range, it is marked as a location of drought. A higher NDDI range indicates more severe drought.

min_time_span: A location marked as a drought must occur for at least this time span within a year to be considered as an extended period of drought.

max_years: If the location meets all the restrictions above, but happens more than a given number of years, that area is discarded as it is considered normal for it to have the other restrictions.

The first three criteria define the multivariate contour in value space. The fourth criterion defines the expanded temporal dimension for the event. This added dimension causes a much increased amount of complexity. The last criterion is a minor component.

Its sole purpose is to filter out regions composed of barren lands, where analyzing drought is not as meaningful.

By using these criteria, we extracted the periods of extended severe or moderate drought. We used our system to query below *NDWI_thresh* and below *NDVI_thresh* on the growing season of the Northern Hemisphere (May - October) and the growing season of the Southern Hemisphere (November - April). After accumulating the queried data over the growing seasons of all the years, a parallel sort was performed in spatial and temporal ordering. We then calculated the time span and number of years that the restriction on *NDDI_range* occurred for each spatial point by stepping through the data on each process. If *min_time_span* and *max_years* was met, the point was colored based on the year that the longest drought occurred. The final image was written in parallel.

We tested our analysis by iteratively stepping through various thresholds of NDVI and NDWI and various ranges of NDDI. Figure 6.1 shows the resulting visualization from setting *NDVI_thresh* = 0.5, *NDWI_thresh* = 0.3, *NDDI_range* = 0.5 – 10, *min_time_span* = 0.3, and *max_years* = 2. In [Gu et al. 2007], the same values for NDWI and NDVI were used to assess drought. The results show various regions where drought was a real-world problem, most notably an abnormal drought in Mexico during 2006. This is one of the examples of how using analysis like this could help find acceptable parameters to use in drought monitoring tools.

6.1.3 Time-Lag Analysis

Along with drought assessment, another widely studied problem in climate science is the time-lag among variable changes. Studying time-lag is important for obtaining better understandings of how variables like NDVI are affected by other conditions. In particular to the drought assessment problem, studying past and present droughts in relation to these conditions could enhance the capability to monitor vegetation and develop better early warning systems [Tadessee et al. 2008].

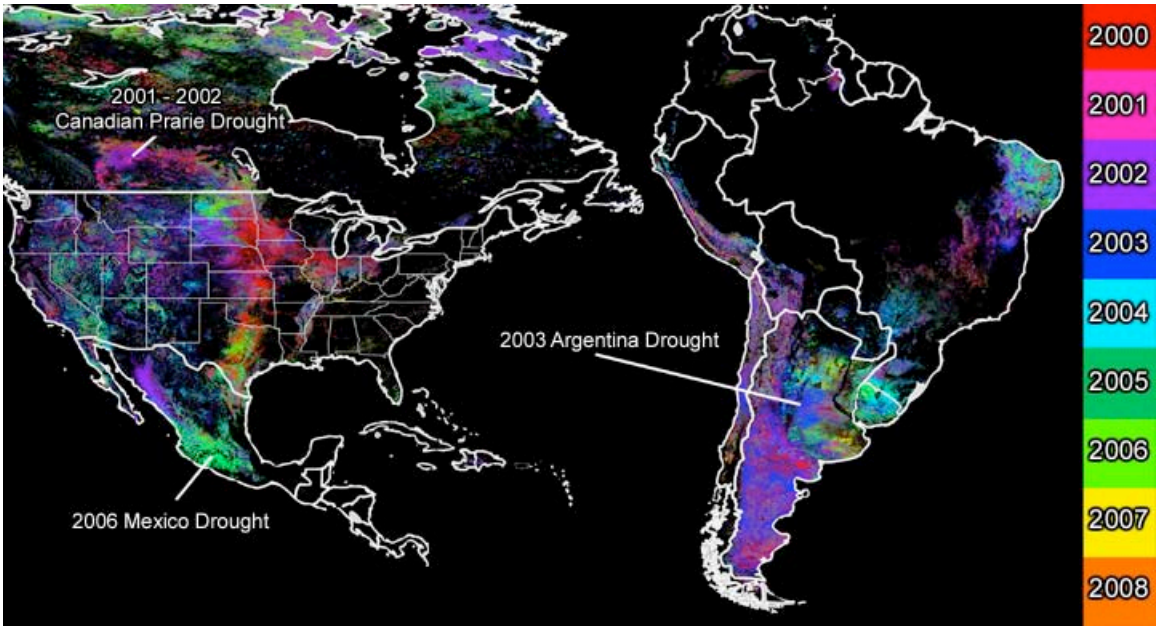


Figure 6.1: Drought analysis of NDVI < 0.5, NDWI < 0.3, and NDDI between 0.5 and 10. These are periods of drought that lasted for at least a month and occurred up to two years for any given region. Several regions are marked where drought has happened, most notably the 2006 Mexico drought. The image was colored based on the year that the longest drought occurred.

We defined the problem of time-lag in the following manner: calculate the time between the first or last occurrence of variable v_0 in range r_0 and the first or last occurrence of variable v_1 in range r_1. We specifically focused on the problem of time-lag between the first snowfall and the first sign of green-up from vegetation. Figure 6.2 shows averaged NDVI and NDWI variables from a region in Saskatchewan Canada to illustrate our problem. NDWI shows abnormally high values when snowfall occurred, and NDVI shows abnormally low values. To calculate the time-lag between first snowfall and vegetation green-up, the times when NDWI reached these abnormally high values and when NDVI first broke out of the low values must be computed.

To solve this, we first queried on ranges r_0 and r_1 for v_0 and v_1 for each year. After querying, the appropriate data was sorted in parallel in spatial and temporal ordering. Each process then computed the time-lag between the first occurrence of v_0 and the first

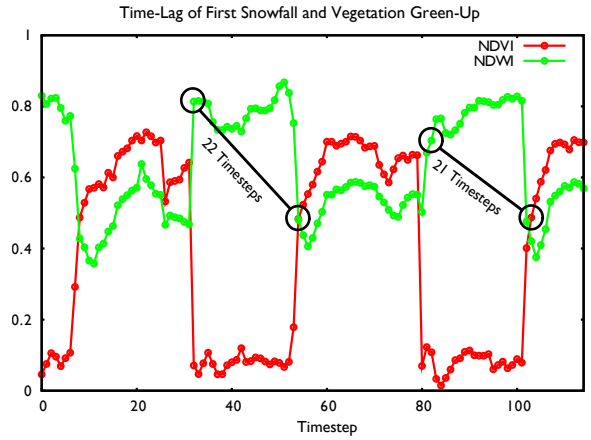


Figure 6.2: Averaged NDVI and NDWI variables from a region in Saskatchewan Canada. Circled points show the first occurrence of NDWI in the 0.7 – 0.9 range and NDVI in the 0.4 – 0.6 range.

occurrence of v_1 in their respective ranges. The resulting image was then colored based on the time-lag and written in parallel.

Figure 6.3 shows the resulting visualization for $0.7 < \text{NDWI} < 0.9$ and $0.4 < \text{NDVI} < 0.6$. The time between the first snowfall and vegetation green-up for various regions in Northern Canada was almost an entire year. Smaller time-lags around the Central and Southeastern United States were found, normally ranging from about one to two months between the first snowfall and the beginning of vegetation green-up.

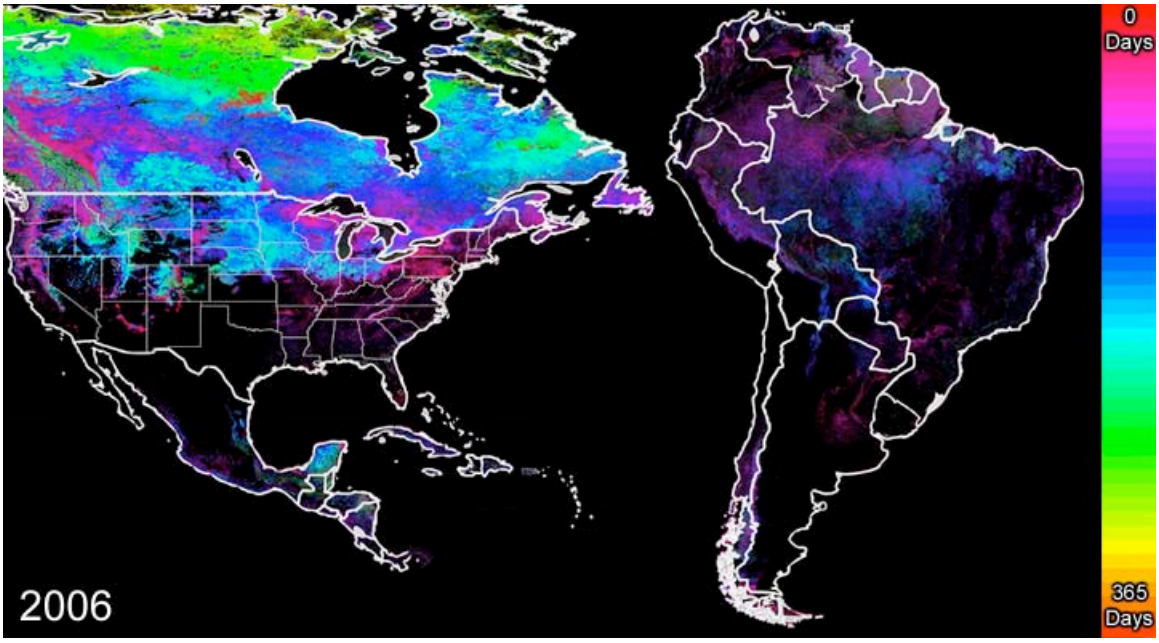


Figure 6.3: Time-lag between the first occurrence of $0.7 < \text{NDWI} < 0.9$ and the first occurrence of $0.4 < \text{NDVI} < 0.6$ in 2006. The color represents the length of the time-lag. Calculating this allowed us to assess the duration from the first snow to the beginning of vegetation green-up.

6.1.4 Application Timing Results

Timing results were gathered from the two separate applications. Detailed results of the drought application are shown in Figure 6.4, and results from the time-lag application are shown in Figure 6.5.

In the drought application, queries were issued on the growing seasons of the Northern and Southern Hemispheres for all growing seasons, resulting in 18 total queries. One parallel sort was issued before analysis, and one final image was written. In Figure 6.4, the aggregate time of all 18 queries is shown, along with timing for the other aspects of the application. With the parameters used to generate Figure 6.1, ≈11 billion relevant items were returned from the queries and sorted before analysis. At 16K cores, the application aggregately queried ≈137.5 billion items per second and sorted ≈2.6 billion items per second.

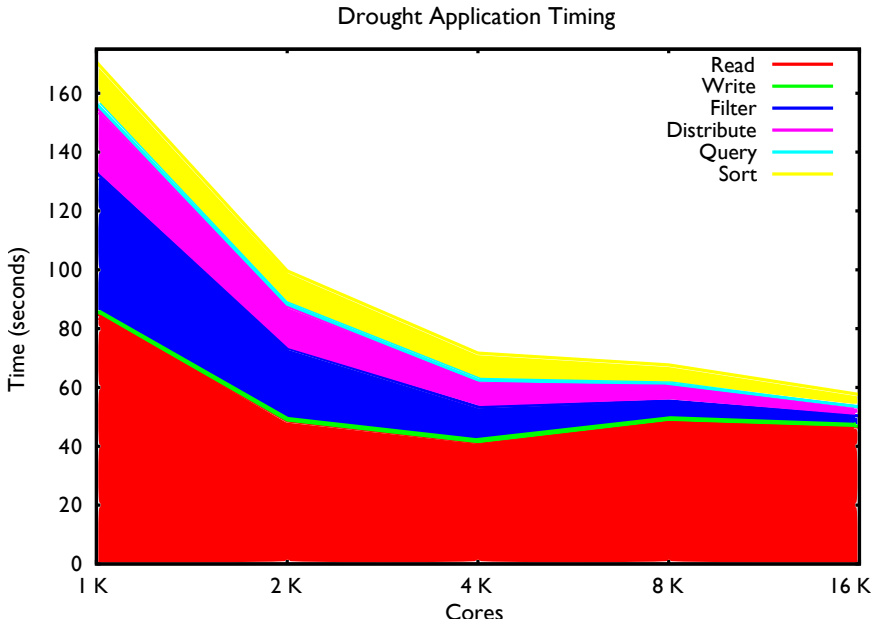


Figure 6.4: Timing results (in seconds) of the drought application.

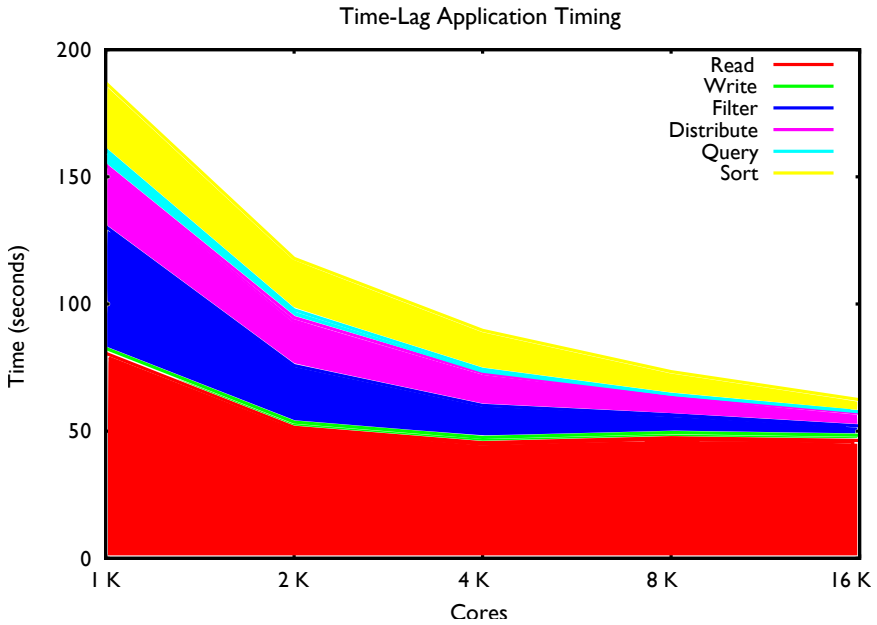


Figure 6.5: Timing results (in seconds) of the time-lag application.

Similarly for the time-lag application, queries were issued for each year on the Northern and Southern Hemispheres with the beginning of each year starting in the Fall. For each year and hemisphere, a query was issued on the appropriate range of NDVI and another was issued on the appropriate range of NDWI to calculate time-lag. Thus, four queries were issued each year. Since we were interested in time-lag on a yearly basis, we gathered the results separately for each year, resulting in 10 sorts being performed before analysis and 10 files being written.

The results in Figure 6.5 show the aggregate times for the querying, sorting, and writing, along with the times for the other parts of the application. With the parameters used to generate Figure 6.3, ≈1.2 billion relevant items were returned, sorted, and analyzed for each year. Since we did this on all 10 years, ≈12 billion relevant items were returned, sorted, and analyzed for the total application. At 16K cores, the application aggregately queried ≈23.5 billion items per second and sorted ≈2.5 billion items per second.

Similar scaling results were observed in the applications. When scaling to 4K cores, I/O bandwidth reached ≈25 GB/s but then tapered off. This is because it was shown in [Yu et al. 2008] that using too many processes to perform I/O will often cause bandwidth degradation. It is beyond the scope of our work right now to address this issue in our reading phase, however, we did gather the final image to a smaller amount of processes in the writing phase. This is why the writing time remained near constant as process counts were scaled.

The aspects of the applications other than I/O showed scalability to 16K cores. The time for filtering the useless points in the dataset turned out to be a significant portion of the application, and more work will be needed to enable faster filtering of useless values. The analysis times were almost negligible. When scaling to 16K cores, both applications showed an end-to-end execution time of nearly one minute.

6.2 Geometric Flow Feature Extraction

The proposed system can also be extended to work with geometric data from flow fields (e.g. traced fieldlines). While many systems have been proposed for scalar analysis of flow

fields [Doleisch 2007], they lack the ability to do quantitative assessment of fieldlines.To integrate geometric flow analysis into my system, I have combined my approach with OSUFlow [Peterka et al. 2011b], a parallel particle tracer. For an extensive explanation about how OSUFlow performs parallel particle tracing, I refer the reader to [Peterka et al. 2011b].

After OSUFlow traces fieldlines, the resulting geometry is stored on disk. My system then operates by reading the fieldline geometry and reading any other scalar values that may be stored with the simulated flow field, such as pressure and density variables. To itemize the dataset and prepare it for querying, associated scalar values are added to the points using trilinear interpolation. Additional quantities related to the geometry of the fieldlines, such as curvature or angle of turn, can be computed and added to the items. After this, the fieldlines are distributed for load balancing and a tree is built on the points of the fieldlines.

Querying is performed on the points of the fieldlines. When a point or percentage of points of a fieldline matches the query, the entire fieldline is returned. This allows querying of fieldlines that exhibit phenomena such as "swirling for 10% of their existence" or "going straight for half of their length and then abruptly turning." As I will show, this ability offers a powerful interface for defining and exploring flow field characteristics.

6.2.1 POP

Here, I show the applicability of my system to flow datasets. I perform geometric analysis of the POP ocean simulation dataset to examine oceanic features like currents and eddies. Before discussing these applications, I outline some of the various features that can be computed on-the-fly with my system and used during analysis.

6.2.2 Derived Geometric Features

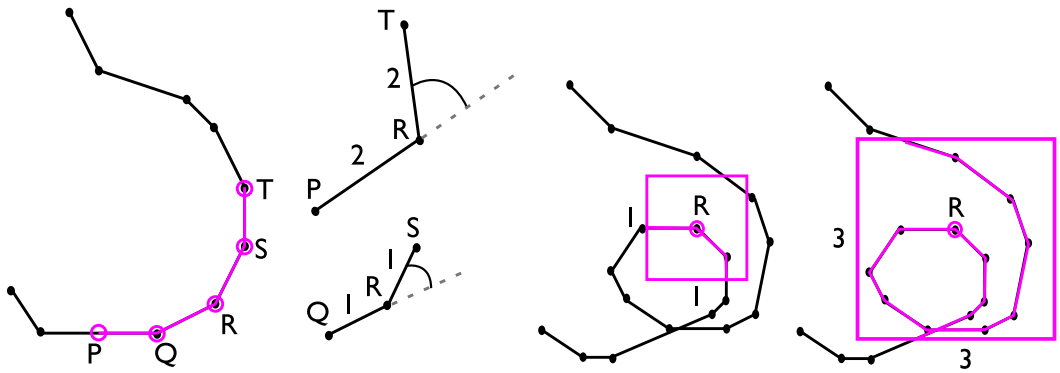
Before analyses of POP begins, derived geometric features are computed to aid in feature specification and extraction. This is similar to computing the derived NDDI variable in the previously explained MODIS drought application. While some studies have examined

global geometric fieldline characteristics like curvature for analysis [Shi et al. 2007], I have researched the viability of using more computationally-intensive neighborhood-based geometric features for flow analysis.

Specifically, the neighborhood-based geometric attributes I examined are angle of turn and residence. As I will show, varying the neighborhood sizes of these attributes is meaningful when examining transient and long-term flow features. My system allows for the precomputation of these features at varying neighborhood sizes for analysis. The unique properties and computation of these features are discussed in the following.

Angle of Turn: My use of angle of turn is patterned after the use of winding angle, which can be computed along the fieldline and used to find swirling patterns such as vortex cores [Sadarjoen et al. 1998]. For finer granularity, I used a variable-length window along the fieldline and computed the angle of turn at each vertex. Figure 6.6a illustrates this. Given a point R and a neighborhood radius of one, I move an arc length of one to the right to get point S and an arc length of one to the left to get point Q. The angle of turn is computed by subtracting the angle formed by $\overrightarrow{RS}$ and $\overrightarrow{RQ}$ from 180°. Similarly, I can use a neighborhood radius of four to compute angle of turn over a longer length. Using combinations of various neighborhood sizes offers intuitive methods when describing fieldlines with swirling characteristics or fieldlines that stay straight or make abrupt transitions.

Residence: Residence describes the residence length and residence time of the fieldline at each of its vertices. A 2D illustration is provided in Figure 6.6b. Given a vertex R, I create a box around it and measure the total arc length of the fieldline from the right and left of R until it exits the box. Residence time is computed by taking the summation of the arc length of every segment inside the box and dividing it by the average velocity magnitude of the two points on each segment. Residence length and residence time are both useful for examining swirling or straight features in relation to velocity of flow.



(a) Angle of Turn **(b)** Residence

Figure 6.6: 2D examples of our neighborhood-based geometric features. Angle of turn is calculated by the angle of vectors formed from the end points at varying arc lengths from a vertex. Residence length is computed by the arc length of a fieldline that resides in a box around a vertex. The residence time can also be computed by dividing this length by the velocity magnitude along the line.

6.2.3 Oceanic Feature Exploration

Before exploration, I traced roughly two million fieldlines to capture the entirety of the POP dataset with the OSUFlow parallel particle tracing library. I stored other various scalar quantities with the computed fieldlines, including vorticity, divergence, velocity magnitude, gradient magnitude, salinity, and temperature. The stored data was roughly 3 GB, a large reduction from the previous vector data size of 160 GB. Once the data was loaded into memory, I computed angle of turn, residence time, and residence length at neighborhood sizes with radii from 1 to 128 in powers of 2. The in-memory footprint of the data was roughly 10 GB.

The first features I explored were the major ocean currents. To do this, I examined the areas of the ocean that exhibited swift and relatively straight movement. Specifically, I used SQI to query for fieldlines that had a low residence time (< 10) with a neighborhood size of 8 for all of their vertices. Figure 6.7 shows a summary visualization of 10,000 randomly sampled fieldlines from this query, and it is colored by shallow areas of the ocean (in blue) to deeper areas (in yellow). Some of the major currents that can be observed in this figure are the Equatorial Currents, which travel almost perfectly horizontal, and the

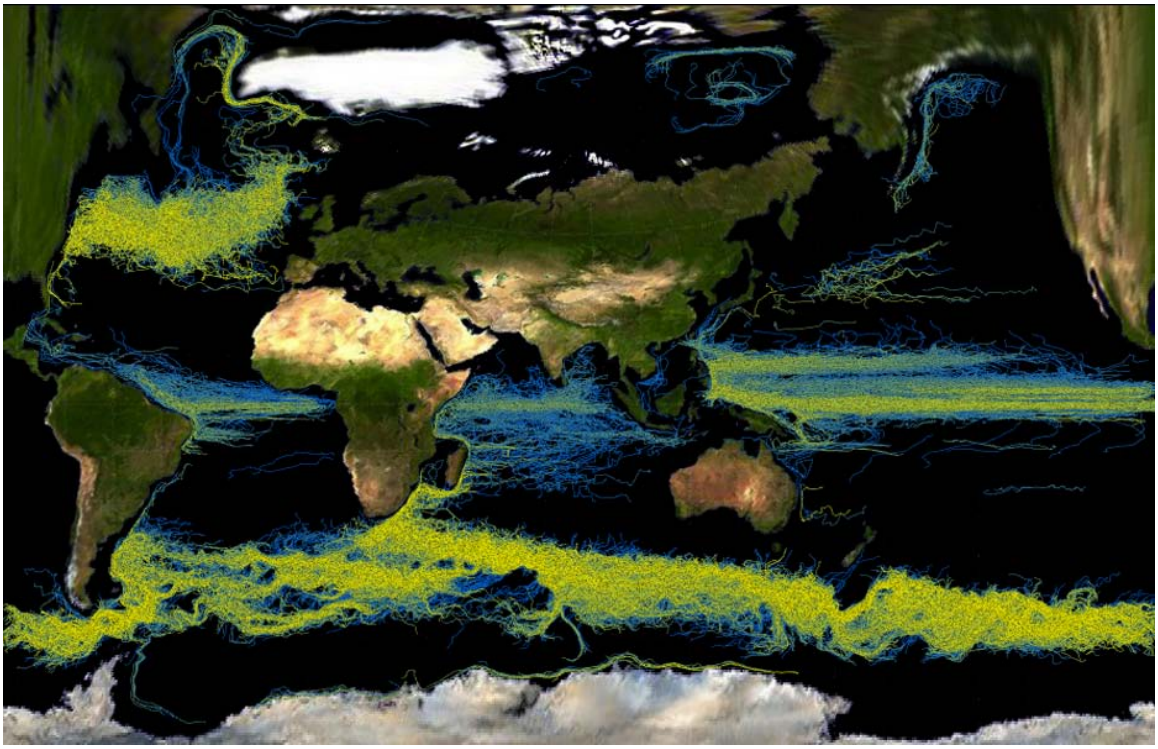


Figure 6.7: A global overview of the major ocean currents in the POP dataset. The currents were extracted by querying for fieldlines that exhibited very low residence time for most of their existence. Many of the major currents, including the Equatorial, Antarctic Circumpolar, and Gulf are easily observed. The color is modulated by yellow for deep to blue for shallow ocean currents.

Antarctic Circumpolar Current, the most dominant current the Southern Hemisphere. The Gulf Stream is also highly recognizable, bordering the continental shelf of the United States and flowing towards Europe. Some other smaller currents include the Alaskan Current and the Labrador Current, which flows between Greenland and Canada. Another small current is the Beaufort Gyre, a shallow, wind-driven current in the Arctic Ocean.

The major currents have considerable effects on various small scale phenomena in the ocean. One of the primary benefits of simulating ocean currents at such high resolution is the ability to resolve smaller high turbulence areas such as eddies. Eddies can range from centimeters to hundreds of kilometers in diameter and can persist for periods of days to many months. I examined various long-term eddies by querying for fieldlines that have high residence length (> 125) in a neighborhood size of 4 for at least 50% of their vertices. 10,000 randomly sampled fieldlines are shown in the global visualization in Figure 6.8, and they are colored by shallow eddies (in blue) to deeper eddies (in yellow). The main areas that exhibit these characteristics are close to the shorelines, where major currents usually flow past and create turbulent activity. High eddy activity is also observed around Madagascar, a phenomenon that has been previously studied [Heywood and Somayajulu 1997].

Eddies have interesting properties that I observed in more detail. At the bottom of Figure 6.8, I zoomed into a portion of the Weddell Sea, an area that has attracted attention to eddy activity [Holland 2001]. The center image at the bottom of Figure 6.8 shows the relatively straight areas of the current, obtained by querying for fieldlines that had a low angle of turn ($< 25°$) in a neighborhood size of 4 for at least 50% of their vertices. The color is modulated by temperature from cold (in blue) to hot (in yellow), and the width of the lines is modulated by salinity. A higher-salinity and warmer current is observed that appears to be driving turbulent activity close to the Antarctic coast. I zoomed in on this area of turbulent activity and again queried for fieldlines that exhibit a low angle of turn, but this time only for 10% of their vertices. The eddies appear to be cold-core eddies, which are classified by having centers that are cooler than the surrounding flow. Cold-core eddies also have the property of being cyclonic, meaning they rotate clockwise in the

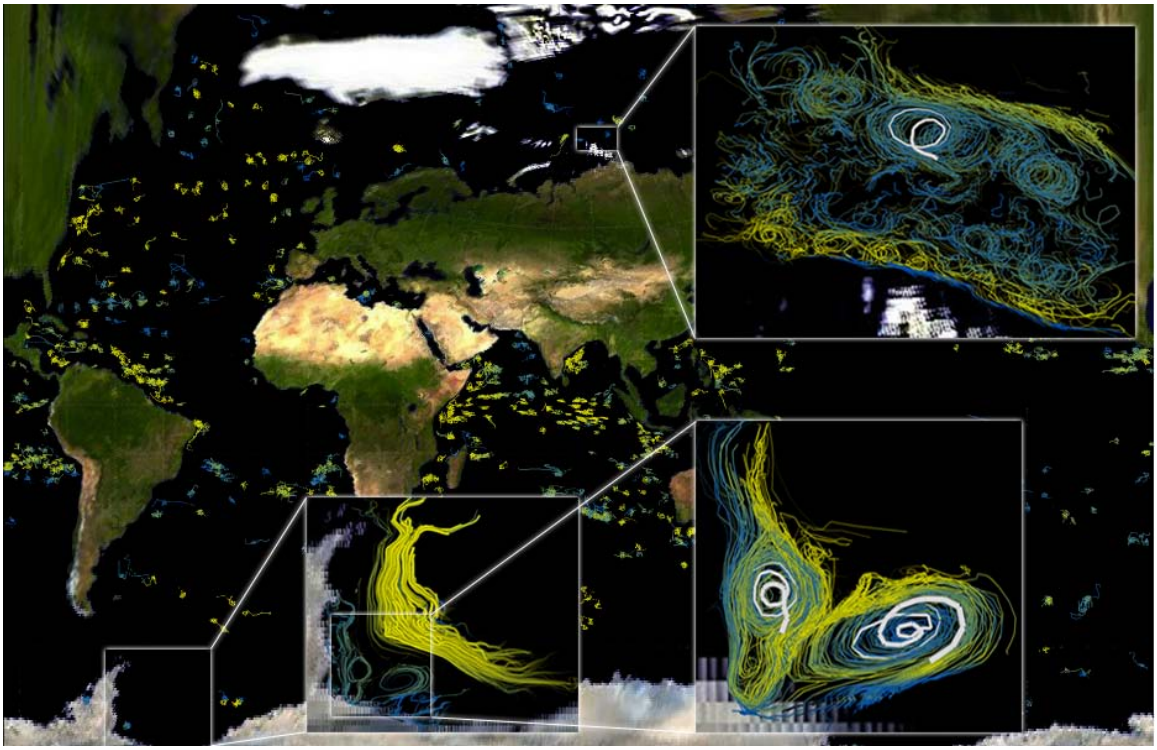


Figure 6.8: A global overview of some of the major ocean eddies in the POP dataset. These were found by querying for fieldlines that exhibited high residence length for the majority of their existence. Some of the areas are magnified and colored to show cold-core eddy activity.

Southern Hemisphere and counterclockwise in the Northern Hemisphere. This was verified by examining the centers of the eddies and querying for fieldlines that have a very high residence length (> 200) with a neighborhood size of 8 for at least 20% of their vertices. The returned fieldlines are colored in white and their widths are modulated by time to show the clockwise rotation of the eddies.

A similar experiment was carried out for the Arctic Ocean, another widely studied area for eddy activity [Manley and Hunkins 1985]. The result, shown at the top of Figure 6.8, shows cold-core eddy activity that results from a higher temperature flow from above. Analogous to the eddies in the Weddell Sea, I extracted a core region of the major eddy to show its cyclonic nature. Since these cold-core eddies are in the Northern Hemisphere, they rotate counterclockwise.

6.2.4 Timing Results

I provide timing results from exploration of the POP dataset at scales from 64 to 1,024 PEs in Table 6.1. The "tracing" and "merging" columns represent the one-time step that occurs to generate the fieldline geometry and merge scalar quantities with it. At 1,024 PEs, OSUFlow was able to trace roughly 2 million fieldlines through the entire ocean dataset in 36 seconds. This number includes the time spent reading the dataset and writing the fieldline geometry. The scalar merging takes about twice as long, primarily because it is reading in additional salinity and temperature quantities from the dataset. Overall, the efficient I/O methods in OSUFlow using BIL obtained a very reasonable preprocessing overhead for even a dataset in the hundreds of gigabytes.

The "startup" and "attributes" timings convey the one-time overhead associated with starting the application. The startup times include I/O overhead and the time it takes to load balance the data and build the necessary querying data structures. The on-the-fly attributes time represents the time it takes to precompute all of the geometric attributes of the fieldlines. Both of these steps showed high scalability. At 1,024 cores, users would be able to recompute an entirely new and extensive feature space in under a second.

PEs	Tracing	Merging	Startup	Attributes	Query	Network	Latency
64	N/A	N/A	31.20	14.38	0.43	0.039	0.47
128	N/A	N/A	15.18	7.31	0.22	0.037	0.26
256	67.68	134.27	7.82	3.67	0.11	0.037	0.15
512	47.89	91.45	4.32	1.84	0.061	0.042	0.10
1 K	36.24	70.56	2.71	0.93	0.034	0.058	0.092

Table 6.1: Average application timing results (in seconds) for the global ocean explorations depicted in Figures 6.7 and 6.8.

The "query", "network", and "latency" times describe the average time it took to query and send the results from the global ocean queries in Figures 6.7 and 6.8. Although not depicted, a remote interface was linked to the system to render the queried geometry. The querying times were highly scalable, obtaining an average time of 0.034 seconds at 1,024 PEs for the largest global ocean queries that I performed. The network transfer times remained nearly equal since both of these experiments restricted the amount of fieldlines to 10,000 and returned roughly 8 MB of geometry. A slight increase, however, in the network time is observed when scaling because of the overhead that is involved when gathering the fieldline geometry to the root PE before sending it to the interface. The relatively constant network times became the bottleneck at larger scales and resulted in less perceived scalability. The overall latency times between submitting a query and obtaining the result, however, were very interactive.

6.3 GEOS5 Internal Model Variability Studies

The biggest testament to the importance of my architecture is the driving application of GEOS-5 internal-model variability analysis. Our initial user need was the ability to analyze teleconnections and identify their difference among the different model runs of the dataset. A teleconnection can generally be described as a significant positive or negative correlation in the fluctuations of a field at widely separated points. We provide a technical use case of our analysis problem and then driving results in inter-hemisphere exchange. We then

provide a performance evaluation of our application and also detail other smaller-scale application examples.

6.3.1 Technical Use Case

The technical requirements of our studies involved quantitatively assessing relationships among the flows from different sources to different destinations. Figure 6.9 illustrates the general problem and approach. Given an unsteady flow field and an initial source of flow (the United States), how can we assess the relationships of the flow with respect to a destination area (China in this example)? While a visualization method such as fieldline rendering can be used (such as in this example), it is difficult for the user to quantitatively assess the relationship. For example, our collaborators were interested in the following analyses:

Time Until Arrival – Starting from the source at various points in time, how long does it take for the flow field to reach the destination?

Residence Time – Once the flow enters the destination, how long does it reside in the area before exiting?

Average CO_2 Until Arrival – What is the concentration of various CO_2 properties along the path to the destination area?

Internal-Model Variability – Given these quantitative analyses, how can they be used in a manner for assessing variability of models with different initial conditions?

Along with these initial challenges, another need from our users was the ability to operate in four dimensions. The GEOS-5 dataset has a time-varying hybrid-sigma pressure grid, with units in meters per second in the horizontal layers and Pascals per second in the vertical direction. Dealing with this grid in physical space involves adjusting for the curvilinear structure of the lat-lon grid and then utilizing another variable in the dataset to determine the pressure thickness at each voxel. Our collaborators were unaware of any

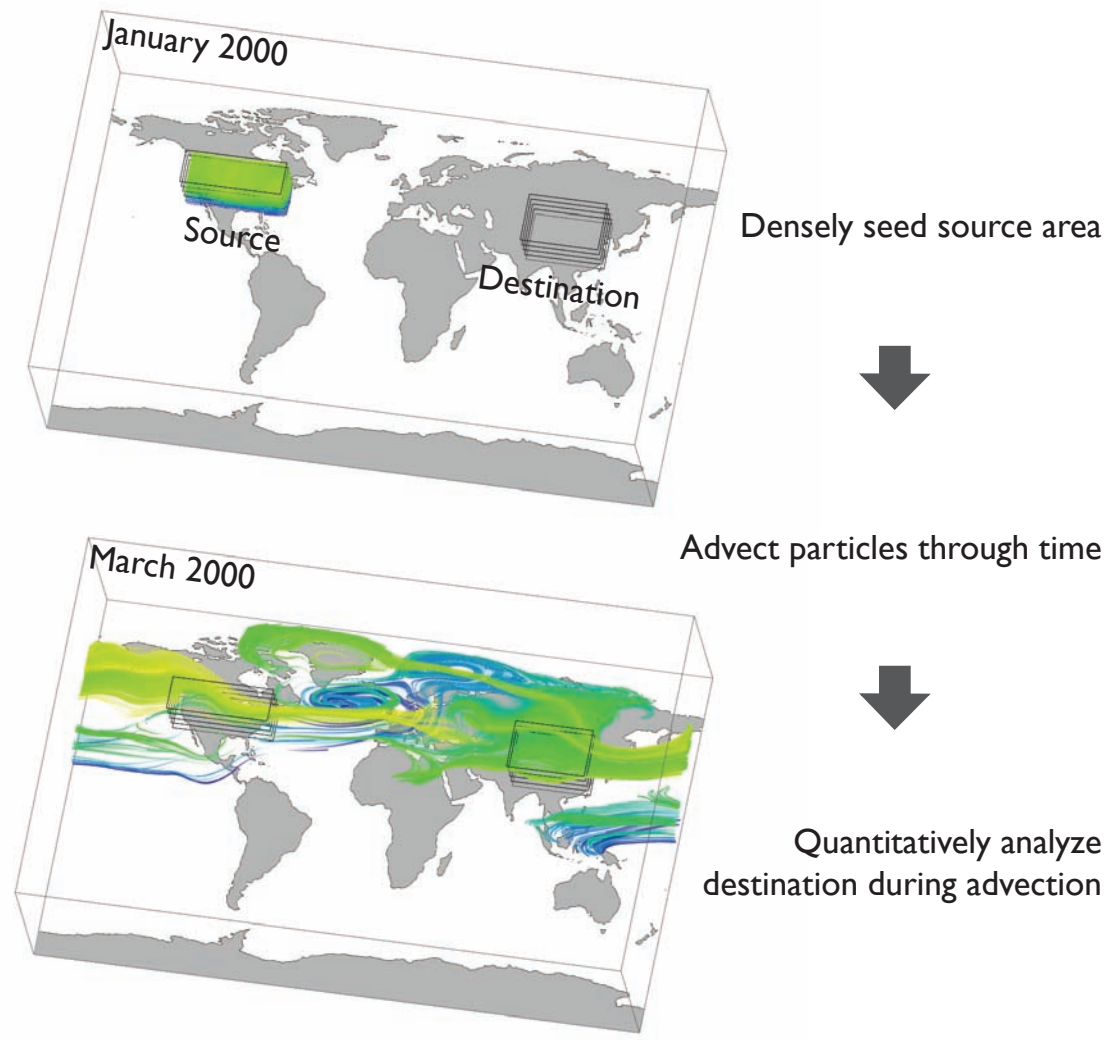


Figure 6.9: Detecting correlations in atmospheric flow from source to destination (teleconnection) is a challenging domain traversal problem. Given an unsteady flow field, what type of quantitative characteristics can be derived from interactions between a given source and destination?

tools that could process the flow of their grid using all four dimensions, and we wrote a custom Runge-Kutta integration kernel for this purpose.

The dstep() function of our application is similar to the example from Section 5, with the exception that particles carry statistics during integration. After each Runge-Kutta step, the particles perform the following function:

```
function UPDATE_PARTICLE(particle)
    if particle.in_destination() then
        if particle.has_already_arrived() then
            particle.residence_time += STEPSIZE
        else
            particle.time_until_arrival = particle.time
            particle.residence_time = 0
        end if
    else
        particle.co2_accumulation += particle.co2
    end if
end function
```

The particles are then reduced based on their starting grid point and the month from which they began tracing. The reduce function then performs point-wise operations of the daily data, averaging it into monthly values. The reduce() function operates in the following manner:

```
function REDUCE(key, particles[ ])
    Result model_results[NUMMODELS]
    for all p in particles[ ] do
        ▷ Compute monthly averages for each model
        model_results[p.model].update_stats(p)
    end for
    for all r in model_results[ ] do
```

emit_write(r) ▷ *Store statistics*

end for

end function

Once finished, the computed statistics may then be rendered and compared with standard point-based techniques.

6.3.2 Studying Inter-Hemisphere Exchange

We used the previously described application to analyze the effects of the flow field from lower levels of the Northern Hemisphere to the lower levels of the Southern Hemisphere. The interaction of the two areas is important since the distributions of heat, moisture, CO_2 and other chemical tracers are critically dependent on exchange between the Northern and Southern Hemispheres. We first used DStep to query for the lower 22 pressure layers of the Northern Hemisphere, and particle tracers were initialized from each queried point. The destination location was set to the lower 22 pressure layers of the Southern Hemisphere.

Since our dataset has a relatively short time span (two years), we focused on small-scale interactions. Specifically, we only saved particles which reached the destination area in under two months. Particles were emitted in five day time intervals for the first year of each model, and each particle was allowed to travel for a year. Before hitting the destination area, particles accumulated CO_2 information at even time samplings and used this to compute the average concentrations along the trace. Once hitting the target destination, particles then accumulated residence time information until exiting the area. If particles exited the area or did not reach it within two months, they were terminated.

We gathered interesting observations using the time until arrival and CO_2 concentrations. Three-dimensional renderings of these characteristics from January and July in two of the ensemble runs are shown in Figure 6.10. The time until arrival starting from January shows interesting properties right along the border of the hemispheres. Between South America and Africa (circle A), one can observe a gap where the particles take up to two months to reach the Southern Hemisphere. In contrast to the surrounding areas, where

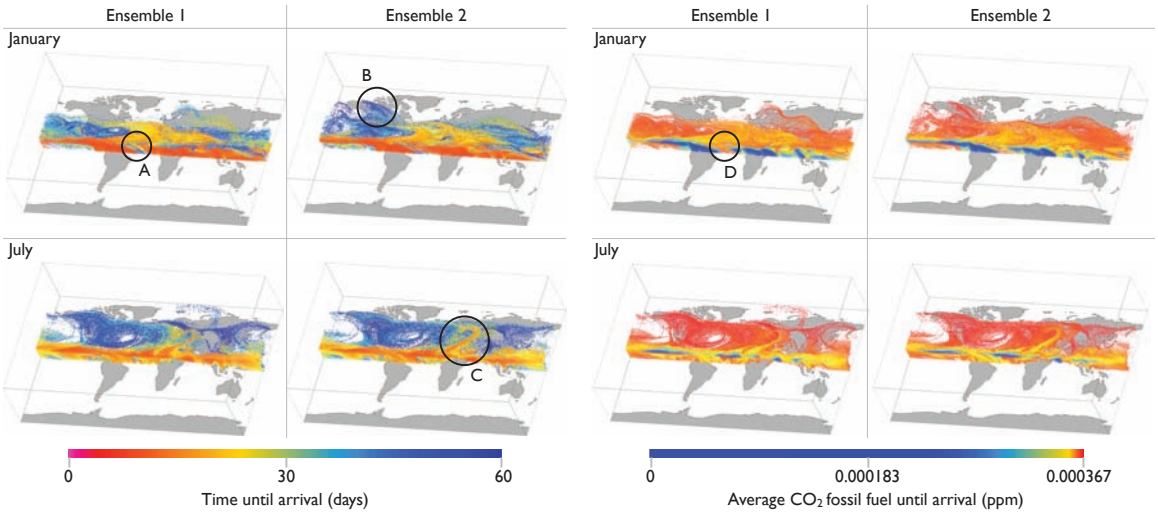


Figure 6.10: Three-dimensional direct volume renderings of the time until arrival and average CO_2 concentrations from January and July in two GEOS-5 ensemble runs. The circled areas are explained in Section 6.3.2.

particles almost immediately reach the Southern Hemisphere, this area is a likely indicator of exchange. Examining the CO_2 concentration at this gap (circle D), one can also observe that the particles traveled through areas with much higher CO_2 concentration.

Time until arrival also shows interesting characteristics in July. In the summer months, the jet stream is located closer to border of the United States and Canada. It appears to be directing many of the particles eastward, which then go into an area of strong downward flow. This downward flow is more apparent in the second ensemble (circle C), resembling the shape of a walking cane. The scientists believed this area could potentially be responsible for much of the interaction that is occurring between the area around the jet stream and the Southern Hemisphere. When observing the CO_2 July rendering, one can observe that many of the main CO_2 emitters potentially carry more CO_2 into the Southern Hemisphere.

One can also arrive at conclusions from visually comparing the ensembles. For example, a structure appears over Canada in January of ensemble two (circle B), but not in ensemble one. For a closer look, we computed the probability that a given lat-lon point over all of the queried vertical layers made it to the Southern Hemisphere within two months.

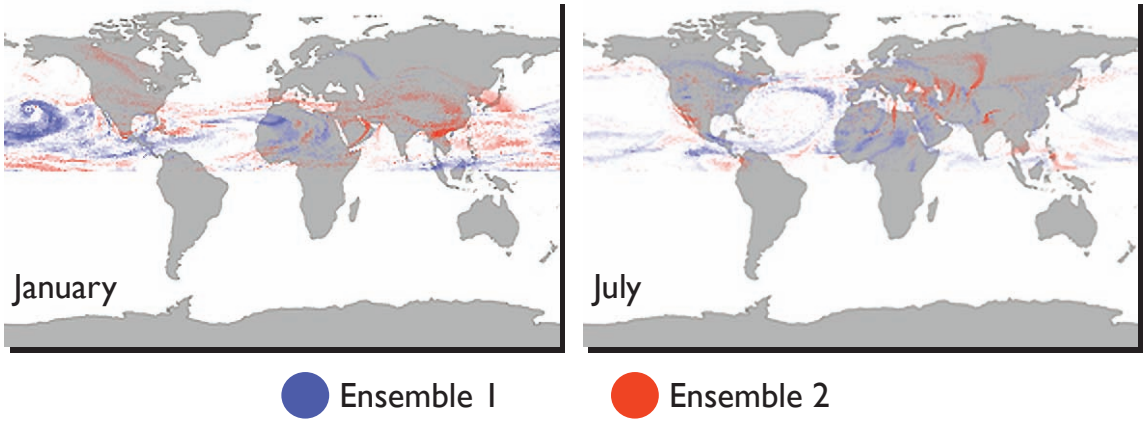


Figure 6.11: Differences between two ensemble runs are revealed by examining their probability distributions in the vertical direction. Color indicates the ensemble that had a higher probability of flow traveling from the Northern to Southern Hemisphere within two months. The opacity of this color is modulated by the absolute difference between the two ensembles, revealing the areas that are different.

We plotted the absolute differences in probability in Figure 6.11. Color represents the model which had higher probability, and the opacity of color is modulated by the absolute difference to highlight differences between the ensembles. One can observe that in January near the Hawaiian area of the Pacific Ocean, particles have a much higher probability of reaching the Southern Hemisphere in ensemble one. As mentioned before, one can also see the structure from ensemble two appearing over Canada in January. In July, the models appear to have similar probabilistic characteristics, which could potentially mean that the ensembles are converging through time.

6.3.3 Inter-Hemisphere Exchange Performance Evaluation

We have evaluated performance of the DStep Runtime in the context of our driving application. Our testing environment is *Intrepid*, an IBM BlueGene/P supercomputer at Argonne National Laboratory. Intrepid contains 40,960 nodes, each containing four cores. We used virtual node mode on Intrepid, which treats each core as a separate process. We also used Intrepid's General Parallel File System (GPFS) for storage and I/O performance results.

We used five variables in the GEOS-5 application. Three variables are horizontal wind and vertical pressure velocities. One variable provides the vertical grid warping, and the last variable is the CO_2 fossil fuel concentration. The dataset is stored in its original netCDF format across 5,840 files. The five variables total to approximately 410 GB.

For inter-hemisphere analysis, the application issued a query for the lower 22 vertical layers of the Northern Hemisphere. We strided the queried results by a factor of two in each spatial dimension and performed this query for the first year of each of the eight ensemble runs in five day intervals. The application initialized approximately 40 million queried particles, which could be integrated up to a year before termination. Many of the particles, however, would be cut off after two months of integration if they had not yet arrived in the Southern Hemisphere.

The tests utilized a worker group of 16 workers that consisted of 15 steppers on separate cores. Reducers, writers, and communicators were placed on the 16th core of each group. Although reducers and writers could potentially interfere with the routing performance of the communicators, this application had small reduction and data output requirements. In total, approximately 400 MB of information were written by the application.

Each worker was configured to own two blocks from the domain and could use up to 128 MB of memory. Because of the memory restrictions, models were processed one at a time at 1 K processes, two at a time at 2 K processes, and so on until all eight models could be processed simultaneously on 8 K processes. Beyond 8 K processes, steppers dynamically incremented the ghost sizes of blocks until their 128 MB limitation was reached. This replicated much of the data, kept more computation local, and ultimately reduced communication requirements.

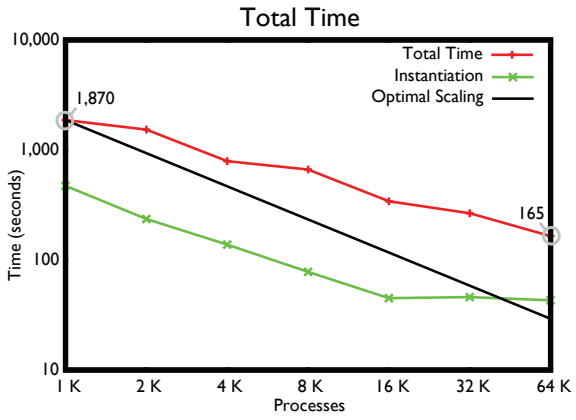
Timing results are shown in Figure 6.12a. The top line shows the entire application time from start to finish. The other line shows the total instantiation time, which includes the time it takes to read the dataset and perform any data replication. In all, the application showed decreased total timing results at every scale, from 1,870 seconds at 1 K processes to 165 seconds at 64 K processes. We observed better parallel efficiency going from 8 K to

64 K (50% efficiency) when compared to 1 K to 8 K (35% efficiency). We attribute this to additional data replication that occurs past 8 K processes.

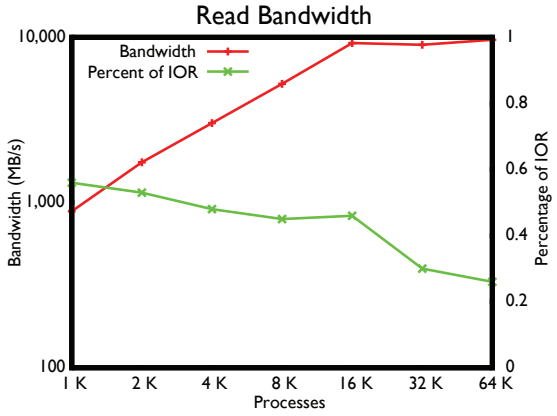
The instantiation times leveled out at 16 K processes, which again is likely attributed to the additional data replication and communication overhead. The reading results, which include data replication as part of the aggregate bandwidth, are shown in Figure 6.12b. The red line shows the bandwidth, which scales up to about 10 GB/s at 16 K cores. We have compared these bandwidth results to the IOR benchmark on Intrepid. IOR provides a baseline benchmark of the maximum obtainable bandwidth. We used similar IOR parameters from the study in [Lang et al. 2009b] for comparisons. For most of the experiements, we were able to obtain 50% of the benchmark. For an application scenario such as ours, which operates on thousands of multivariable netCDF files, consistently obtaining 50% of the benchmark was considered a success.

The asynchronous and dynamic nature of our system makes it difficult to measure component times other than initialization. We have plotted the total bytes transferred over the network in Figure 6.12c to better illustrate some of the communication requirements of this application. At smaller scales (less than 16 K), processes did not have much extra room for dynamic ghost size extension and communicated about 90 GB of particle information in total throughout the job. At 16 K and beyond, processes have much more room to dynamically resize ghost regions of their blocks, which in turn kept more computation local. This is the primary reason why job times continue to scale past 16 K cores.

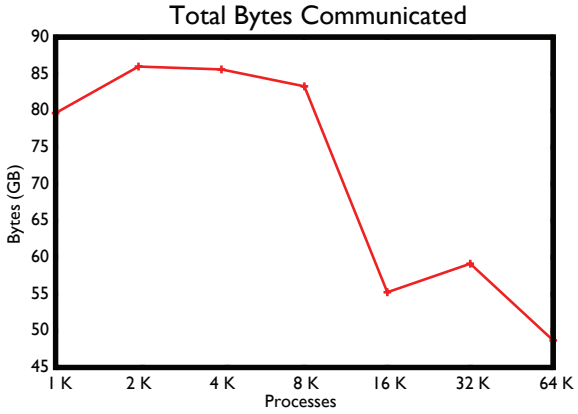
In all, the performance results showed that – even at data sizes of hundreds of GBs, time scales at thousands of days, and advection of tens of millions of particles – the execution of complex domain traversal analysis can be scaled to the largest of today's machines in reasonable times for an interactive scientific analysis setting. To the best of our knowledge, this is the largest particle tracing experiment to date.



(a) Total Time



(b) Read Bandwidth



(c) Bytes Communicated

Figure 6.12: Performance analysis if GEOS-5 Norhtern-Southern Hemisphere interaction application.

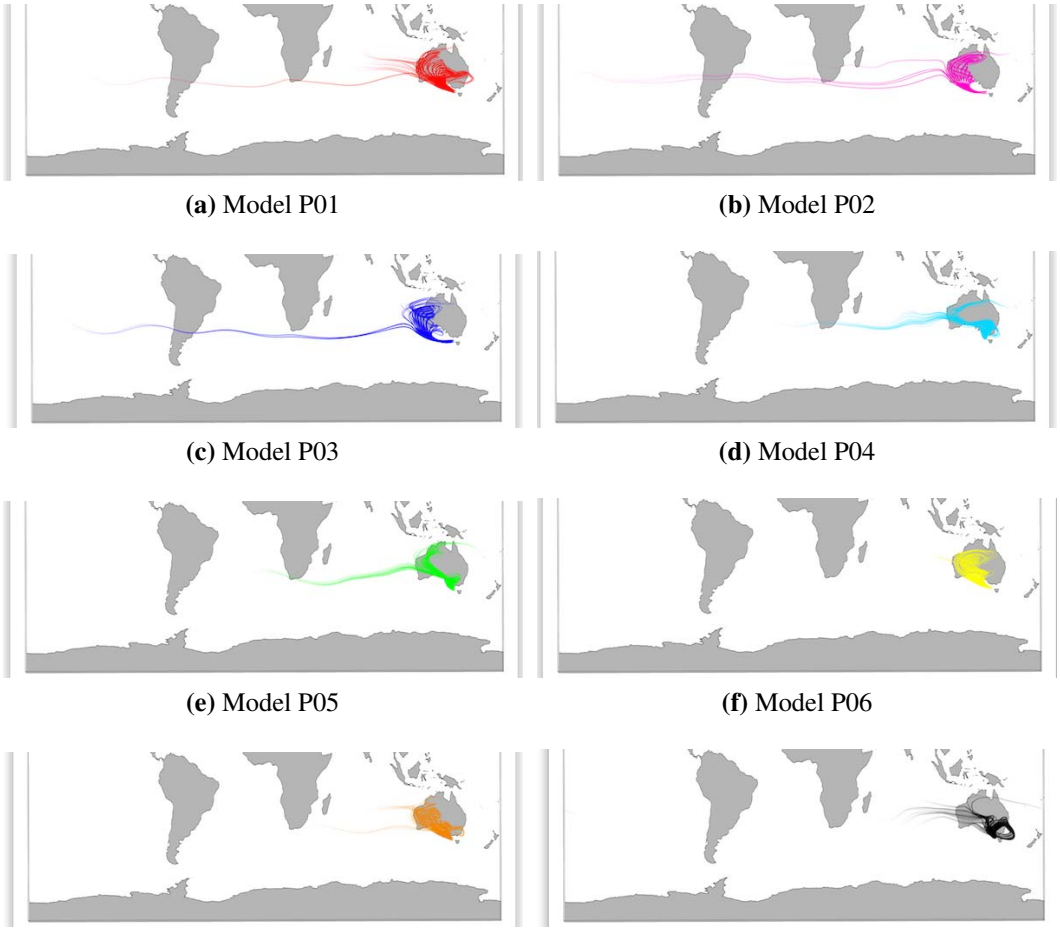
6.3.4 Assessment of Cape Grim Incoming Flow Variability

Although atmospheric models such as GEOS-5 do capture average distributions of CO_2 at a global scale, there are still inaccuracies that need to be studied at smaller local scales in order to assess anomalies of the model. One region of high variability among the GEOS-5 model run is Cape Grim. Our collaborators believed that examing the average directional distributions of incoming flow could provide interesting qualitative and quantitative comparisons of the model runs.

To determine the historical path of a given particle arriving at Cape Grim, integration progresses backward through time. In these analyses, the lower 13 pressure layers (approximately 1013 hPa to 8200hPa) were queried for, and particle tracers initialized from each of those queried points. The destination location was set to the lower 13 pressure layers of the Cape Grim. Monthly GEOS-5 data was used on a smaller desktop computing resource (detailed in Section 6.3.5). Geometry was saved using DStep, and the reduce() function built a histogram of directional distributions for each model.

During collaboration with our atmospheric scientists, a look at the flow patterns of individual ensemble member simulations for May 2001 at one month lead time (Figure 6.13) reveals significant differences in particle pathways into Cape Grim. Figure 6.13 shows streamlines coming primarily from the Atlantic, Africa and southern Asia. P01, P02, P04 and P07 include paths taken through North America. The long range particles that look like long tails in P02-P05 are the result of the fast winds of the upper vertical layers eventually carrying particles downward into the lower layes of Cape Grim.

In addition to the variability among the GEOS-5 model simiulations for Cape Grim for each of the two years simulated, our collaborators also noticed significant variability between years of the models. This result is consistent with the findings of [Conway et al. 1994] who report significant interannual variations in the interhemispheric gradient of atmospheric CO2 in the observation network. Figure 6.14 depicts May 2000 (top) and 2001 (bottom) arrivals into Cape Grim with one month lead time. Streams from blue (P03), purple (P02), red (P01) and cyan (P04) arrive latest, since these come from the largest



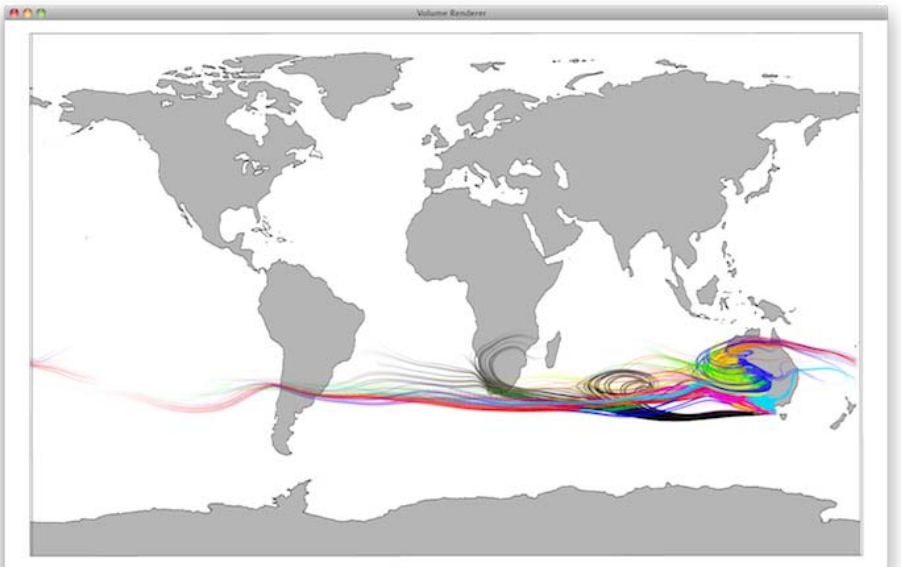
(a) Model P01 **(b)** Model P02

(c) Model P03 **(d)** Model P04

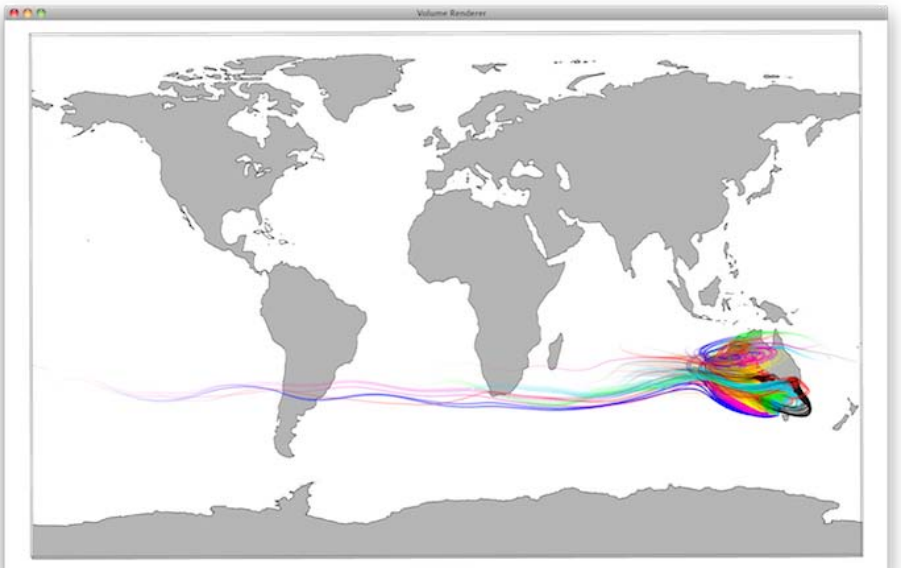
(e) Model P05 **(f)** Model P06

(g) Model P07 **(h)** Model P08

Figure 6.13: Particle pathways into Cape Grim from the eight ensemble runs of monthly GEOS-5 2001 data. The wind trajectories are integrated backwards for one month from May 2001. Comparisons among the models show high variability of incoming flow at this location.



(a) May 2000



(b) May 2001

Figure 6.14: Interannual variability among models in the GEOS-5 runs for years 2000 and 2001. Particles are advected backwards for three months from of the eight models (colored differently).

distances away. In the 2000 simulation, black (P08) is in the horn of Africa one month before arrival, it then travels around a South Pacific vortex before entering Cape Grim. In the 2001 simulation, black (P08) spins in a tight vortex around Tasmania while the other simulations rotate nearer the southern Australian coast. According to our collaborators, these interannual variations may be due to seasonal cyclonic activity in the Indian Ocean or to climatic oscillations such as the El Nino/La Nina cycle. In either case, this variability is seen in only some of the simulations suggesting that meteorological differences from year to year represent only part of the variability in the simulations.

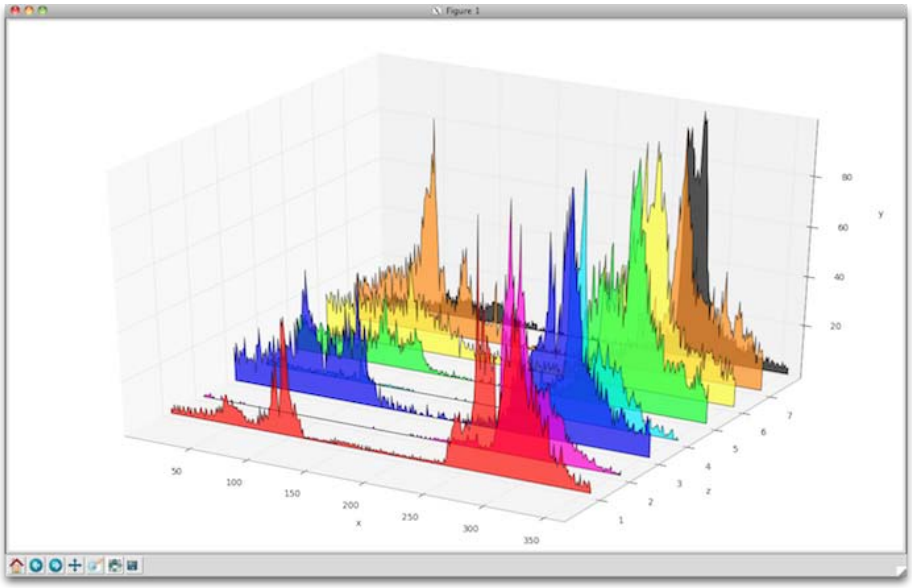
Figure 6.15 contains histograms indicating the direction from which the wind blows one month in advance into Cape Grim for May 2000 and May 2001. The x-axis in each histogram plot represents the direction from which the wind arrives at Cape Grim (in degrees); the z-axis contains one point for each of the model simulations (retaining the color scheme of that of the flow plots); and the y-axis contains the normalized counts of each direction (multiplied by 100 for ease of plotting). In all of the histograms, it can be seen that the wind primarily enters Cape Grim from the west, although in May 2001 there is a more prominent eastern wind component than May 2000. In one case (P07, orange) the eastern wind component dies down.

According to our collaborators, this was the first time that they have been able to quantitatively examine incoming wind distributions in regards to a sink location.

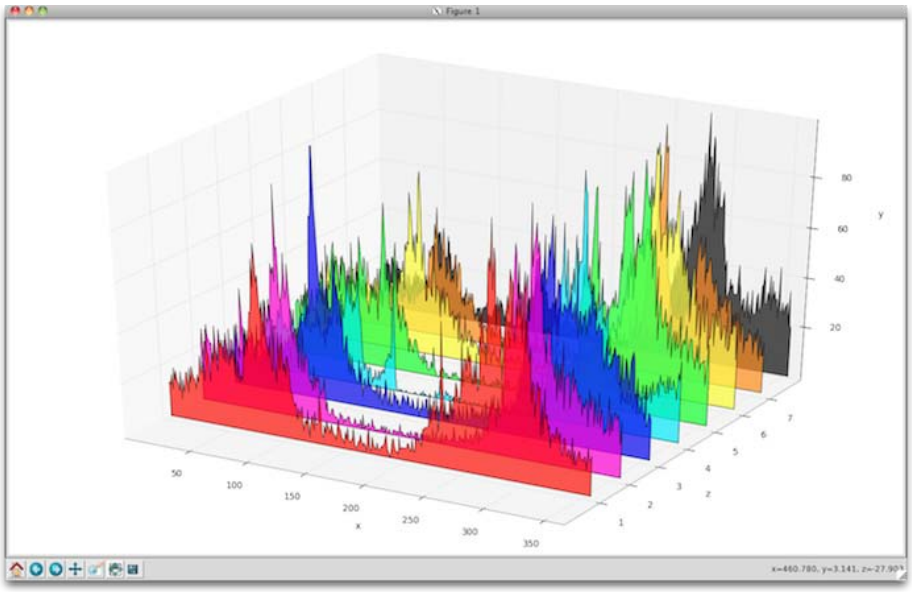
6.3.5 Small-Scale Cape Grim Performance Evaluation

Although DStep's primary use is on supercomputing resources, it also performs well on smaller scale computing resources such as multi-core desktop machines. The Cape Grim variability study shown in the previous subsection is a problem that is small enough to be executed on a desktop.

The desktop resource used in this study is the Sonoran machine (Section 2.1). Tests were conducted using up to all eight cores of the machine. Starting with two cores, DStep used one stepper on one core and placed one writer and one communicator on the second



(a) May 2000



(b) May 2001

Figure 6.15: Distributions of wind direction in the GEOS-5 models. Particles are advected backwards from Cape Grim for three months. The first axis shows the incoming direction in degrees, the second axis represents each model, and the vertical axis shows a normalized count of directions.

Processes	I/O	Comm / Comp	Total
2	50.04	19.86	69.9
3	27.27	11.85	39.27
4	23.37	7.61	30.98
5	20.82	6.44	27.42
6	18.99	5.33	24.34
7	18.22	4.10	22.32
8	19.43	4.10	23.53

Table 6.2: Performance evaluation of Cape Grim application on a Linux Desktop. Timing results are in seconds and componentized into I/O time, communication and computation time, and total time.

core. In all cases when scaling, the number of steppers was increased the the number of communicators and writers stayed at one. The configuration also split the domain into 64 blocks.

Timing results are shown in Table 6.2, which show I/O times, combined communication and computation times, and then the total times of the application. These results are averaged across three separate runs, each which showed negligible deviations among times. When starting out at two processes, one must take into account that only one stepper is being used by DStep, hence one process is performing reading. At three cores, the reading time is almost cut in half primarily because the two I/O buses and disks are being more sufficiently utilized. The reading times increase at smaller amounts when scaling. This can be attributed to more disk saturation by the Block I/O Layer, which is scheduling the reading of as many files at a time as possible. At seven cores, read bandwidth peaks to about 75 MB/s.

Communication and computation times also scale up to seven processes, and they appear to bottom out when going to eight processes. In all, the application scaled up to the fullest extent of the machine, with the exception of the last run which included a small increase in read time.

Chapter 7

Conclusion

As computational science progresses to the exascale, analysis and visualization will be limiting factors in gaining insight from exascale data [Dongarra, J. et. al 2010]. I believe that architectures that operate similar to the MapReduce paradigm will be pivotal in addressing not only the data bandwidth bottleneck, but also the parallel programming bottleneck faced by application scientists. My architecture provides a novel solution for this growing problem by simplifying full-range analysis of large datasets on modern simulation architectures. I summarize some of the major accomplishments of this dissertation and also address limiting factors that may be addressed by future work.

7.1 Success Summary

The success of my architecture can be measured by a variety of quantitative and qualitative metrics. I provided several quantitative comparisons of components of my work with other approaches, which included:

1. Accelerating parallel I/O in the OSUFlow library over the previous single-file I/O approach (Section 3.2)

2. A more scalable parallel sample sorting algorithm over the classic algorithm (Section 4.4)

3. The use of DStep for accelerating parallel particle tracing over a state-of-the-art library (Section 5.5)

The "simplification" of full-range analysis is a characteristic that must be measured in a more qualitative manner. I believe that the following characteristics of my architecture back up my claim of simplification:

1. The two-function design pattern for parallel I/O (Section 3.1) not only allows users to natively perform I/O across multiple files, but it also masks complexities of three different file formats. The ability to do this with such a restricted interface has significantly shortened the I/O code in DIY [Peterka et al. 2011a] and OSUFlow [Peterka et al. 2011b].

2. The two-function design of SQI (Section 4.1) masks a large amount of data organization complexity from the user, resulting in less programmer time with less errors. For example, the entire MODIS drought application C code presented in Section 6.1.2 is less than one hundred lines of code.

3. DStep also has a two-function interface (Section 5.1) that masks significant parallel complexities to the user. Similar to SQI, the interface has allowed large-scale atmospheric analysis applications (Section 6.3.2 and Section 6.3.4) to be written in less than one hundred lines of C++ code. Furthermore, the user needs almost no parallel programming knowledge.

My architecture has led to several successes in computational science. SQI enabled the interactive tuning and visual analysis of parameters which define drought (Section 6.1.2). SQI was also used to interactively assess how geometric properties of flow fields relate to features such as cold-core eddies and major currents in large-scale ocean data (Section 6.2.3). DStep helped atmospheric scientists understand, for the first time, quantitative flow field differences among different GEOS-5 atmospheric models (Section 6.3.2 and Section 6.3.4). DStep was even used as a tool for performing internal model variability studies for a student's Master's Thesis [Allen 2011].

7.2 Potential Future Improvements

There are challenges which my solutions and methods do not directly address. Since I have geared by architecture for large-scale computing resources, I have neglected the regard for addressing out-of-core execution. For example, one type of out-of-core execution that I did not address was streaming data through the nodes during computation. In some cases I have sidestepped this restriction (such as processing individual models at a time in Section 6.3.3), but I have yet to study this issue in more depth. I believe this is one promising area in which my research may be extended. This is mostly true for those who wish to perform similar analyses on smaller computing clusters.

Another example of out-of-core execution that I did not address is in transforming the problem to an in-core problem via data reduction. Some examples in this regard include performing analysis on a wavelet decomposition of the dataset or on a reduced-dimensionality dataset. These routines can often be specific to an application, and they are out of the scope of the more general problem that I am addressing.

Another limit of my work is in the processing of multiresolution and unstructured grids. Currently there has only been limited advancements in the parallel I/O of multiresolution data [Kumar et al. 2010]. Since my focus is heavily on climate, I have not had the need to process these types of datasets. I think this is another interesting area which could extend my research.

Bibliography

Bibliography

Allen, M. (2011). The effects of varying physical parameterizations and initial conditions on tracer transport in the national aeronautics and space administrations goddard earth observation system model, version 5. Master's thesis, The University of Tennessee, Knoxville. 101

Biddiscombe, J., Geveci, B., Martin, K., Morel, K., and Thompson, D. (2007). Time dependent processing in a parallel pipeline architecture. *IEEE Transactions on Visualization and Computer Graphics*, 13. 27

Blelloch, G. E., Leiserson, C. E., Maggs, B. M., Plaxton, C. G., Smith, S., and Zagha, M. (1998). An experimental analysis of parallel sorting algorithms. *Theory of Computing Systems*, 31(2):135–167. 22, 45

Bremer, P.-T., Weber, G. H., Tierny, J., Pascucci, V., Day, M. S., and Bell, J. B. (2011). Interactive exploration and analysis of large-scale simulations using topology-based data segmentation. *IEEE Transactions on Visualization and Computer Graphics*, 17:1307–1324. 26

Camp, D., Garth, C., Childs, H., Pugmire, D., and Joy, K. (2011). Streamline integration using mpi-hybrid parallelism on large multi-core architectures. *tvcg*, 17(11). 24

Childs, H., Duchaineau, M., and Ma, K.-L. (2006). A scalable, hybrid scheme for volume rendering massive data sets. In *Proc. of Eurographics Symp. on Parallel Graphics and Visualization*, pages 153–162. 16

Childs, H., Pugmire, D., Ahern, S., Whitlock, B., Howison, M., Prabhat, Weber, G. H., and Bethel, E. W. (2010). Extreme scaling of production visualization software on diverse architectures. *IEEE Computer Graphics and Applications*, 30:22–31. 30

Conway, T. J., Tans, P. P., Waterman, L. S., Thoning, K. W., Kitzis, D., Masarie, K. A., and Zhang, N. (1994). Evidence for interannual variability of the carbon cycle from the national oceanic and atmospheric administration/climate monitoring and diagnostics laboratory global air sampling network. *Journal of Geophysical Research-Atmospheres*, 99. 94

Dean, J. and Ghemawat, S. (2004). Mapreduce: Simplified data processing on large clusters. In *OSDI '04: Sixth Symposium on Operating System Design and Implementation*. 3, 28, 50, 56

Dean, J. and Ghemawat, S. (2008). Mapreduce: Simplified data processing on large clusters. *Communications of the ACM*, 51:107–113. 3, 28

Doleisch, H. (2007). Simvis: Interactive visual analysis of large and time-dependent 3d simulation data. pages 712 –720. 77

Doleisch, H., Gasser, M., and Hauser, H. (2003). Interactive feature specification for focus+context visualization of complex simulation data. In *VISSYM '03: Proceedings of the symposium on Data visualisation 2003*, pages 239–248, Aire-la-Ville, Switzerland. Eurographics Association. 15, 16, 18

Doleisch, H., Mayer, M., Gasser, M., and Hauser, H. (2004). Case study: Visual analysis of complex, time-dependent simulation results of a diesel exhaust system. In *In Proc. of the 6th Joint IEEE TCVG - EUROGRAPHICS Symposium on Visualization (VisSym 2004*, pages 91–96. 15, 19

Dongarra, J. et. al (2010). IESP Roadmap. *Univ. of Tennessee Dept. of Computer Science Technical Report, ut-cs-10-652*. vi, 100

Erickson, D. J., Mills, R. T., Gregg, J., Blasing, T. J., Hoffman, F. M., Andres, R., Devries, M., Zhu, Z., and Kawa, S. R. (2007). An estimate of monthly global emissions of anthropogenic CO2: The impact on the seasonal cycle of atmospheric CO2. *Journal of Geophysical Research*, 113. 17

Fabian, N., Moreland, K., Thompson, D., Bauer, A., Marion, P., Geveci, B., Rasquin, M., and Jan, K. (2011). The paraview coprocessing library: A scalable, general purpose in situ visualization library. In *Large Data Analysis Symposium (LDAV) at IEEE Visualization*. 26

Fahey, M. R., Larkin, J. M., and Adams, J. (2008). I/O performance on a massively parallel Cray XT3/XT4. In *IPDPS '08: Proceedings of the IEEE International Symposium on Parallel and Distributed Processing*, pages 1–12. 11

Galbally, D., Fidkowski, K., Willcox, K., and Ghattas, O. (2010). Non-linear model reduction for uncertainty quantification in large-scale inverse problems. *International Journal for Numerical Methods in Engineering*, 81(12):1581–1608. 7, 17

Gao, B. (1996). NDWI – A normalized difference water index for remote sensing of vegetation liquid water from space. *Remote Sensing of Environment*, 58(3):257–266. 69

Gao, K., keng Liao, W., Nisar, A., Choudhary, A., Ross, R., and Latham, R. (2009). Using subfiling to improve programming flexibility and performance of parallel shared-file i/o. *International Conference on Parallel Processing*, pages 470–477. 20, 58

Glatter, M., Huang, J., Ahern, S., Daniel, J., and Lu, A. (2008). Visualizing temporal patterns in large multivariate data using textual pattern matching. *IEEE Transactions on Visualization and Computer Graphics*, 14(6):1467–1474. 16, 18, 21

Glatter, M., Mollenhour, C., Huang, J., and Gao, J. (2006). Scalable data servers for large multivariate volume visualization. *IEEE Transactions on Visualization and Computer Graphics*, 12(5):1291–1298. 13, 21, 22, 41, 42, 45

Gosink, L., Anderson, J., Bethel, W., and Joy, K. (2007). Variable interactions in query-driven visualization. *IEEE Transactions on Visualization and Computer Graphics*, 13(6):1400–1407. 21

Gosink, L., Anderson, J. C., Bethel, E. W., and Joy, K. I. (2008). Query-driven visualization of time-varying adaptive mesh refinement data. *IEEE Transactions on Visualization and Computer Graphics*, 14(6). 13, 21

GRASS Development Team (2008). *Geographic Resources Analysis Support System (GRASS GIS) Software*. Open Source Geospatial Foundation. 14

Gropp, W., Huss-Lederman, S., Lumsdaine, A., Lusk, E., Nitzberg, B., Saphir, W., and Snir, M. (1998). *MPI-The Complete Reference: Volume 2 - The MPI Extensions*. MIT Press, Cambridge, MA, USA. 19

Gu, Y., Brown, J. F., Verdin, J. P., and Wardlow, B. (2007). A five-year analysis of MODIS NDVI and NDWI for grassland drought assessment over the central Great Plains of the United States. *Geophysical Research Letters*, 34. 12, 14, 69, 70, 71

Hannachi, A., Jolliffe, I. T., and Stephenson, D. (2007). Empirical orthoghonal functions and related techniques in atmospheric science: A review. *International Journal of Climatology*, 27:1119–1152. 17

Heywood, K. J. and Somayajulu, Y. K. (1997). Eddy activity in the south indian ocean from ers-1 altimetry. In *ERS Symposium on Space at the Service of Our Environment*. 81

Hoefler, T., Lumsdaine, A., and Dongarra, J. (2009). Towards efficient mapreduce using MPI. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface, 16th European PVM/MPI Users' Group Meeting*. Springer. 3, 28

Holland, D. M. (2001). Explaining the weddell polynya-a large ocean eddy shed at maud rise. *Science*, 292:1697–1701. 81

Ji, G., Shen, H.-W., and Wenger, R. (2003). Volume tracking using higher dimensional isosurfacing. In *IEEE Visualization Conference*, page 28. 18

Kehrer, J., Ladstädter, F., Muigg, P., Doleisch, H., Steiner, A., and Hauser, H. (2008). Hypothesis generation in climate research with interactive visual data exploration. *IEEE Trans. on Visualization and Computer Graphics*, 14(6):1579–1586. 15, 19

Kendall, W., Glatter, M., Huang, J., Peterka, T., Latham, R., and Ross, R. (2009). Terascale data organization for discovering multivariate climatic trends. In *SC '09: Proceedings of ACM/IEEE Supercomputing 2009*. 4, 6, 13, 16

Kendall, W., Huang, J., Peterka, T., Latham, R., and Ross, R. (2011a). Visualization viewpoint: Towards a general I/O layer for parallel visualization applications. *IEEE Computer Graphics and Applications*, 31(6). 4, 58

Kendall, W., Wang, J., Allen, M., Peterka, T., Huang, J., and Erickson, D. (2011b). Simplified parallel domain traversal. In *SC '11: Proceedings of ACM/IEEE Supercomputing 2011*. 4, 7

Kumar, J., Mills, R. T., Hoffman, F. M., and Hargrove, W. W. (2011). Parallel k-means clustering for quantitative ecoregion delineation using large data sets. *Procedia CS*, 4:1602–1611. 14

Kumar, S., Pascucci, V., Vishwanath, V., Carns, P., Hereld, M., Latham, R., Peterka, T., Papka, M., and Ross, R. (2010). Towards parallel access of multi-dimensional, multi-resolution scientific data. In *Petascale Data Storage Workshop*, pages 1–5. 32, 102

Lang, S., Carns, P., Latham, R., Ross, R., Harms, K., and Allcock, W. (2009a). I/O performance challenges at leadership scale. In *SC '09: Proceedings of IEEE/ACM Supercomputing*. 10

Lang, S., Carns, P., Latham, R., Ross, R., Harms, K., and Allcock, W. (2009b). I/O performance challenges at leadership scale. In *SC '09: Proceedings of ACM/IEEE Supercomputing 2009*. 92

Lin, S.-J. (2004). A "vertically lagrangian" finite-volume dynamical core for global models. *Monthly Weather Review*, 132:2293–2307. 8

Lindstrom, P. and Isenburg, M. (2006). Fast and efficient compression of floating-point data. *IEEE Trans. on Visualization and Computer Graphics*, 12:1245–1250. 32

Liu, C. and Wu, J. (2008). Crop drought monitoring using MODIS NDDI over mid-territory of China. *IEEE International Geoscience and Remote Sensing Symposium*. 12, 14, 69

Lofstead, J., Zheng, F., Klasky, S., and Schwan, K. (2009). Adaptable, metadata rich IO methods for portable high performance IO. In *IPDPS '09: Proceedings of the IEEE International Symposium on Parallel and Distributed Processing*. 25

Lofstead, J., Zheng, F., Liu, Q., Klasky, S., Oldfield, R., Kordenbrock, T., Schwan, K., and Wolf, M. (2010). Managing variability in the I/O performance of petascale storage systems. In *SC '10: Proceedings of ACM/IEEE Supercomputing 2010*. 20, 58

Lorensen, W. E. and Cline, H. E. (1987). Marching cubes: A high resolution 3d surface construction algorithm. In *Proceedings of ACM SIGGRAPH*, pages 163–169. 13

Ma, K.-L., Stompel, A., Bielak, J., Ghattas, O., and Kim, E. J. (2003). Visualizing large-scale earthquake simulations. In *SC '03: Proceedings of the ACM/IEEE Supercomputing Conference*. 20

Ma, K.-L., Wang, C., Yu, H., and Tikhonova, A. (2007). In situ processing and visualization for ultrascale simulations. *Journal of Physics*, 78. (Proceedings of the SciDAC 2007 Conference). 13, 25

Malewicz, G., Austern, M. H., Bik, A. J., Dehnert, J. C., Horn, I., Leiser, N., and Czajkowski, G. (2010). Pregel: A system for large-scale graph processing. In *Proceedings of the 2010 International Conference on Management of Data*, SIGMOD '10, pages 135–146. 50

Maltrud, M. E. and McClean, J. L. (2005). An eddy resolving global 1/10 ocean simulation. *Ocean Modelling*, 8(1-2):31–54. 9

Manley, T. O. and Hunkins, K. (1985). Mesoscale eddies of the arctic ocean. *JOURNAL OF GEOPHYSICAL Research*, 90:4911–4930. 83

McCormick, P., Inman, J., Ahrens, J., Hansen, C., and Roth, G. (2004). Scout: A hardware-accelerated system for quantitatively driven visualization and analysis. In *Proc. of IEEE Visualization*, pages 171–178. 16, 27

McCormick, P., Inman, J., Ahrens, J., Mohd-Yusof, J., Roth, G., and Cummins, S. (2007). Scout: a data-parallel programming language for graphics processors. *Parallel Comput.*, 33:648–662. 27

McLoughlin, T., Laramee, R. S., Peikert, R., Post, F. H., and Chen, M. (2009). Over two decades of integration-based, geometric flow visualization. In *Eurographics 2009 State of the Art Report*, pages 73–92, Munich, Germany. 15

Memik, G., Kandemir, M. T., Liao, W.-K., and Choudhary, A. (2006). Multicollective I/O: A technique for exploiting inter-file access patterns. *ACM Transactions on Storage*, 2(3):349–369. 20

Mills, R. T., Hoffman, F. M., Kumar, J., and Hargrove, W. W. (2011). Cluster analysis-based approaches for geospatiotemporal data mining of massive data sets for identification of forest threats. *Procedia CS*, 4:1612–1621. 14

Moreland, K., Avila, L., and Fisk, L. A. (2007). Parallel unstructured volume rendering in paraview. In *Proc. of IS&T SPIE Visualization and Data Analysis 2007*. 16

Moreland, K., Ayachit, U., Geveci, B., and Ma, K.-L. (2011). Dax toolkit: A proposed framework for data analysis and visualization at extreme scale. In *Large Data Analysis and Visualization Symposium (LDAV) at IEEE Visualization*. 27

Nouanesengsy, B., Lee, T.-Y. L., and Shen, H.-W. (2011). Load-balanced parallel streamline generation on large scale vector fields. *tvcg*, 17(6). 24, 26

Ott, L., Duncan, B., Pawson, S., Colarco, P., Chin, M., Randles, C., Diehl, T., and Nielsen, E. (2010). Influence of the 2006 Indonesian biomass burning aerosols on tropical dynamics studied with the GEOS-5 AGCM. *Journal of Geophysical Research*, 115. 17

Peterka, T., Ross, R., Kendall, W., Gyulassy, A., Pascucci, V., and Shen, H.-W. (2011a). Scalable parallel building blocks for custom data analysis. In *Large Data Analysis Symposium (LDAV) at IEEE Visualization*. 26, 101

Peterka, T., Ross, R., Nouanesengsey, B., Lee, T.-Y., Shen, H.-W., Kendall, W., and Huang, J. (2011b). A study of parallel particle tracing for steady-state and time-varying flow fields. In *IPDPS '11: Proceedings of the IEEE International Symposium on Parallel and Distributed Processing*. 7, 15, 24, 26, 34, 50, 62, 63, 64, 65, 77, 101

Peterka, T., Yu, H., Ross, R., and Ma, K.-L. (2008). Parallel volume rendering on the IBM Blue Gene/P. In *EGPGV '08: Proceedings of the Eurographics Symposium on Parallel Graphics and Visualization*, pages 73–80. 20

Peterka, T., Yu, H., Ross, R., Ma, K.-L., and Latham, R. (2009). End-to-end study of parallel volume rendering on the IBM Blue Gene/P. In *ICPP '09: Proceedings of the International Conference on Parallel Processing*. 26, 30

Plimpton, S. J. and Devine, K. D. (2011). Mapreduce in mpi for large-scale graph algorithms. 3, 28

Pugmire, D., Childs, H., Garth, C., Ahern, S., and Weber, G. H. (2009). Scalable computation of streamlines on very large datasets. In *SC '09: Proceedings of IEEE / ACM Supercomputing*. 15, 23, 50, 62, 64

Rübel, O., Prabhat, Wu, K., Childs, H., Meredith, J., Geddes, C. G. R., Cormier-Michel, E., Ahern, S., Weber, G. H., Messmer, P., Hagen, H., Hamann, B., and Bethel, E. W.

(2008). High performance multivariate visual data exploration for extremely large data. In *SC '08: Proceedings of the ACM/IEEE Supercomputing Conference*, pages 1–12, Piscataway, NJ, USA. IEEE Press. 21

Sadarjoen, I., F.H.P., Ma, B., Banks, D., and Pagendarm, H. (1998). Selective visualization of vortices in hydrodynamic flows. In *Proc. of IEEE Visualization*. 78

Schroeder, W. J., Martin, K. M., and Lorensen, W. E. (1996). The design and implementation of an object-oriented toolkit for 3d graphics and visualization. 27

Shi, K., Theisel, H., christian Hege, H., and peter Seidel, H. (2007). Path line attributes - an information visualization approach to analyzing the dynamic behavior of 3d timedependent flow fields. In *In Proceedings of TopoInVis 2007*. 15, 78

Silver, D. and Wang, X. (1997). Tracking and visualizing turbulent 3d features. *IEEE Transactions on Visualization and Computer Graphics*, 3(2):129–141. 13

Silver, D. and Wang, X. (1998). Tracking features in unstructured datasets. In *VIS '98: Proceedings of the IEEE Visualization Conference*. 18

Stockinger, K., Bethel, E. W., Campbell, S., Dart, E., and Wu, K. (2006). Detecting distributed scans using high-performance query-driven visualization. In *SC '06: Proceedings of the ACM/IEEE Supercomputing Conference*. 21

Stockinger, K., Shalf, J., Bethel, W., and Wu, K. (2005a). Query driven visualization of large data sets. In *Proc. of IEEE Visualization*, pages 167–174. 16

Stockinger, K., Shalf, J., Wu, K., and Bethel, E. (2005b). Query-driven visualization of large data sets. In *VIS '05: Proceedings of the IEEE Visualization Conference*, pages 167–174. 13, 21, 26, 41

Stockinger, K., Shalf, J., Wu, K., and Bethel, E. W. (2005c). Query-Driven Visualization of Large Data Sets. In *Proceedings of IEEE Visualization 2005*, pages 167–174. IEEE Computer Society Press. LBNL-57511. 53

Stuart, J. and Owens, J. (2011). Multi-GPU MapReduce on GPU clusters. In *IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. 3, 28

Tadessee, T., Wardlow, B. D., and Ryu, J. H. (2008). Identifying time-lag relationships between vegetation condition and climate to produce vegetation outlook maps and monitor drought. *22nd Conference on Hydrology*. 71

Thakur, R., Gropp, W., and Lusk, E. (1999). Data sieving and collective I/O in ROMIO. In *Proceedings of the 7th Symposium on the Frontiers of Massively Parallel Computation*, pages 182–189. 19

Tu, T., Rendleman, C. A., Borhani, D. W., Dror, R. O., Gullingsrud, J., Jensen, M. O., Klepeis, J. L., Maragakis, P., Miller, P., Stafford, K. A., and Shaw, D. E. (2008). A scalable parallel framework for analyzing terascale molecular dynamics simulation trajectories. In *Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, SC '08. 3, 28, 54

Tzeng, F.-Y. and Ma, K.-L. (2005). Intelligent feature extraction and tracking for visualizing large-scale 4d flow simulations. In *SC '05: Proceedings of the ACM/IEEE Supercomputing Conference*. 18

Vishwanath, V., Hereld, M., and Papka, M. (2011). Towards simulation-time data analysis and i/o acceleration on leadership-class systems. In *Large Data Analysis and Visualization Symposium (LDAV) at IEEE Visualization*. 26

Wang, L., Qu, J., and Xiong, X. (2008). A seven year analysis of water related indices for Georgia drought assessment over the 2007 wildfire regions. *IEEE International Geoscience and Remote Sensing Symposium*. 12, 14, 69

Wei, J., Wang, C., Yu, H., and Ma, K.-L. (2010). A sketch-based interface for classifying and visualizing vector fields. In *Proceedings of IEEE Pacific Visualization Symposium*. 15, 19

Woodring, J. and Shen, H.-W. (2006). Multi-variate, time varying, and comparative visualization with contextual cues. *IEEE Trans. on Visualization and Computer Graphics*, 12(5):909–916. 18

Yu, H., Ma, K.-L., and Welling, J. (2004a). I/O strategies for parallel rendering of large time-varying volume data. In *EGPGV '04: Proceedings of the Eurographics Symposium on Parallel Graphics and Visualization*, pages 31–40. 20

Yu, H., Ma, K.-L., and Welling, J. (2004b). A parallel visualization pipeline for terascale earthquake simulations. In *SC '04: Proceedings of the ACM/IEEE Supercomputing Conference*. 20

Yu, H., Wang, C., Grout, R., Chen, J., and Ma, K.-L. (2010). In situ visualization for large-scale combustion simulations. *Computer Graphics and Applications, IEEE*, 30(3):45 –57. 25

Yu, H., Wang, C., and Ma, K.-L. (2007). Parallel hierarchical visualization of large time-varying 3d vector fields. In *SC'07*, pages 1–12. 15, 23

Yu, W., Vetter, J. S., and Oral, S. (2008). Performance characterization and optimization of parallel I/O on the Cray XT. In *IPDPS '08: Proceedings of the IEEE International Symposium on Parallel and Distributed Processing*, pages 1–11. 37, 76

Vita

Wesley Kendall was born in Knoxville, Tennessee on July 19, 1986. He completed his undergraduate career at the University of Tennessee at Knoxville with a Bachelor's degree in Computer Science and a minor in Mathematics in December 2007. He also continued his graduate studies at the University of Tennessee at Knoxville, completing his Master's degree in Computer Science in May 2009. In December 2011, he graduated from The University of Tennessee at Knoxville with the Doctor of Philosophy degree in Computer Science.