Simplified Parallel Domain Traversal

Wesley Kendall[*], Jingyuan Wang[*], Melissa Allen[†], Tom Peterka[‡], Jian Huang[*], and David Erickson[§]
[*] Department of Electrical Engineering and Computer Science, The University of Tennessee, Knoxville
[†] Department of Civil and Environmental Engineering, The University of Tennessee, Knoxville
[‡] Mathematics and Computer Science Division, Argonne National Laboratory
[§] Computational Earth Sciences Group, Oak Ridge National Laboratory

ABSTRACT

Many data-intensive scientific analysis techniques require global domain traversal, which over the years has been a bottleneck for efficient parallelization across distributed-memory architectures. Inspired by MapReduce and other simplified parallel programming approaches, we have designed *DStep*, a flexible system that greatly simplifies efficient parallelization of domain traversal techniques at scale. In order to deliver both simplicity to users as well as scalability on HPC platforms, we introduce a novel two-tiered communication architecture for managing and exploiting asynchronous communication loads. We also integrate our design with advanced parallel I/O techniques that operate directly on native simulation output. We demonstrate DStep by performing teleconnection analysis across ensemble runs of terascale atmospheric CO_2 and climate data, and we show scalability results on up to 65,536 IBM BlueGene/P cores.

Keywords

Data-Intensive Analysis, Parallel Processing, Parallel Particle Tracing, Atmospheric Ensemble Analysis

1. INTRODUCTION

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.

In this work, 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 [23] and 2011 [21].

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 [7] for text processing and Pregel [19] 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

SC11 November 12-18, 2011, Seattle, Washington, USA

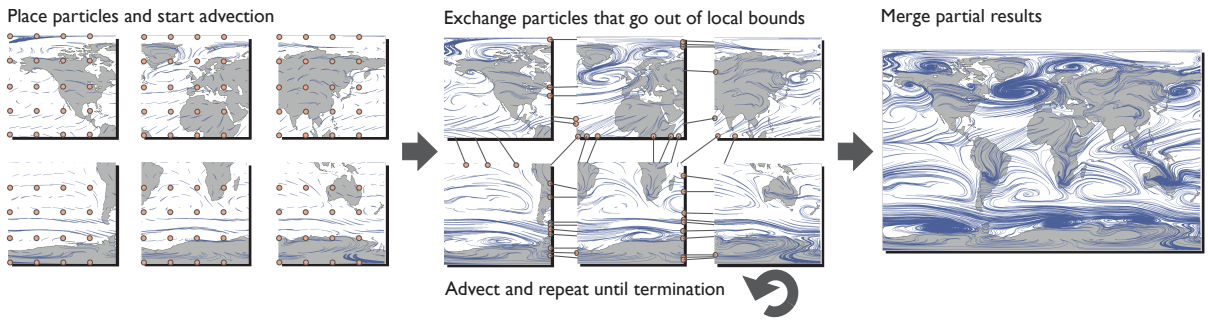


Figure 1: One example of domain traversal is fieldline tracing, shown here using six distributed-memory processes. The procedure initializes particles in the subdomains, which are then advected through the flow field. Particles are exchanged when going out of bounds, and all partial fieldlines are merged when particles finish advection.

example, to support CO_2 source apportionment according to movement patterns of specific CO_2 molecules between the Northern and Southern Hemispheres. 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, and we provide driving application results on an IBM Blue-Gene/P machine.

2. BACKGROUND

Our work encompasses a variety of areas related to large-data processing. We first begin by describing related analysis techniques of our driving application. We then cover related work in parallel flow tracing, which is our defining problem for domain traversal. We also review existing approaches in simplified large-data processing.

2.1 Driving Application

The driving application for our work 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) [4], Interactive Data Language (IDL) suites [2], and general scientific visualization tools such as VisIt [6]. 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 [9, 20], or some type of model output reduction such as Empirical Orthogonal Functions (EOF) [12] or various types of sampling of the data output [10] because of limited computational power.

For example, in a very recent 2010 assessment of the effects of biomass burning in Indonesia, Ott et al. [20] 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. 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.

2.2 Parallel Flow 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 analysis technique is the tracing of tangential fieldlines to the velocity field. Steady-state fieldlines are the solution to the ordinary differential equation

$$\frac{d\vec{x}}{ds} = \vec{v}(\vec{x}(s)) \; ; \; \vec{x}(0) = (x_0, y_0, z_0), \tag{1}$$

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 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 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. [27] 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. [23] 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 a master process that monitors 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.

Peterka et al. [21] avoided the bottleneck of having one master 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.

Because parallel particle tracing is one of the most well-defined domain traversal problems in visualization, we use it as the primary test case. However, beyond this test case, our overall goal is to create a design for general domain traversal problems. The novelty of our work is to improve user effectiveness by allowing them to write succinct and powerful analysis applications, and by transparently acheiving scalability at large scale without the need of detailed understanding of each parallel systems' unique aspects. Hence, a comparison solely about performance against algorithms mentioned in this section is beyond the scope of this work.

2.3 Simplified Large-Scale Data Processing

Many large data processing problems have been solved by allowing users to write serial functional programs which can be executed in parallel. The defining example is Google's MapReduce [7], which provides a simple programming framework for data parallel tasks. 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 [8]. There are also systems that have implemented MapReduce on top of MPI [13, 22] as well as multi-GPU architectures [25].

The profound success of MapReduce in industry has inspired its use in scientific settings. Tu et al. [26] designed *Hi-Mach*, a Molecular Dynamics trajectory analysis 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. Kendall et al. [14] also performed time-varying climatic analysis tasks on over a terabyte of satellite data using an infrastructure similar to MapReduce. The authors showed scalability up to 16 K Cray XT4 cores and total end-to-end execution times under a minute.

Although MapReduce is useful for data-parallel tasks, many problems are not inherently data parallel and are difficult to efficiently parallelize with MapReduce. Graph processing is one class of problems that fit in this category. Malewicz et al. introduced Pregel [19], a programming framework and implementation for processing large-scale graphs. In contrast with MapReduce, a process in Pregel has the ability to communicate to neighbors based on the topology of the graph. Users are required to implement various functions that operate on a per-vertex basis. The authors showed that the model was expressive enough to perform many popular graph algorithms, and to also scale across thousands of cores in a commodity cluster.

Like Pregel, we have found that allowing a restricted form of communication provides a much more flexible model for data traversal. In contrast, DStep and our analysis needs are centered around spatiotemporal scientific datasets. Allowing arbitrary traversal through a domain, while powerful for many tasks, can easily introduce high communication volumes that do not have a structured form such as a graph.

3. DSTEP - SIMPLIFIED PARALLEL 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.

3.1 DStep API

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 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 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.

3.2 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 [24], 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.

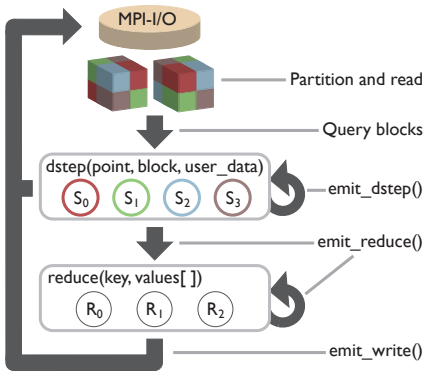


Figure 2: Data flow using DStep. Data movement, primarily directed by emit functions, is shown by arrows. Computational functions and associated workers are enclosed in blocks.

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.

3.3 DStep Data Flow

Given our API and the design of application instantiation, we illustrate the entire data flow of DStep in Figure 2. 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 [26], 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.

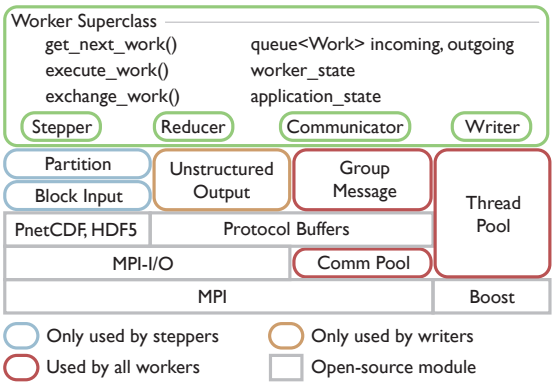


Figure 3: The software design of DStep is shown, with open-source modules in gray and custom components in other colors.

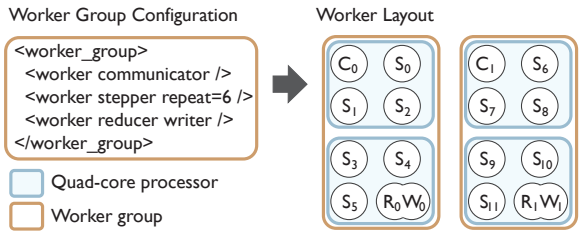


Figure 4: Example worker group configuration and layout. The worker group configuration specifies eight different workers, which are replicated across pairs of quad-core processors.

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 [7], we designed our implementation to leverage HPC architectures for scientific analysis scenarios.

4.1 Software Architecture

A general overview of the DStep software stack is illustrated in Figure 3. 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 subsection. 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.

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.

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 [5] 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 [1]. 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) [15]. 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 [15] 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 [11]. Second, in a similar method to [18], we found that pinning

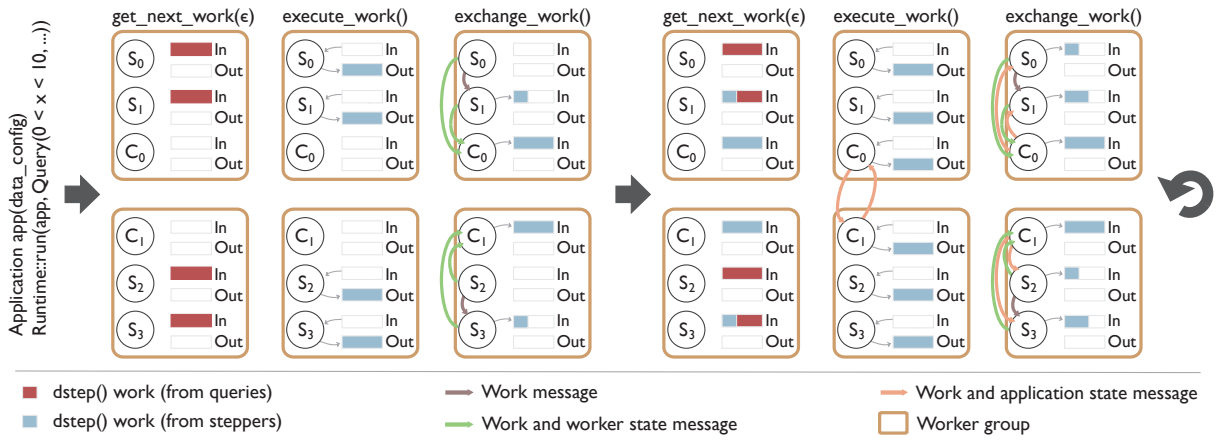


Figure 5: Example of two epochs executed by two worker groups that contain steppers and communicators. Steppers initially fill their incoming work queues with points that match the user query, and work progresses through the system. The second execute_work() function shows the primary overlap of communication with computation - communicators are performing long-range communication while steppers are performing work.

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.

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 4, 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.

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 $\epsilon - incoming\_work.size()$ 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 work-

ers. 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. 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 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 [21] and the strategy that used a single master in [23], we have found this hybrid and highly asynchronous strategy to be beneficial to our data demands. We demonstrate the performance of the DStep Runtime in the context of our driving application – terascale atmospheric analysis.

5. DRIVING APPLICATION RESULTS

Our initial user need was the ability to analyze teleconnections and perform internal-model variability studies. 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 an overview of our dataset, the technical use case of our analysis problem, and then driving results in inter-hemisphere exchange. We then provide a performance evaluation of our application.

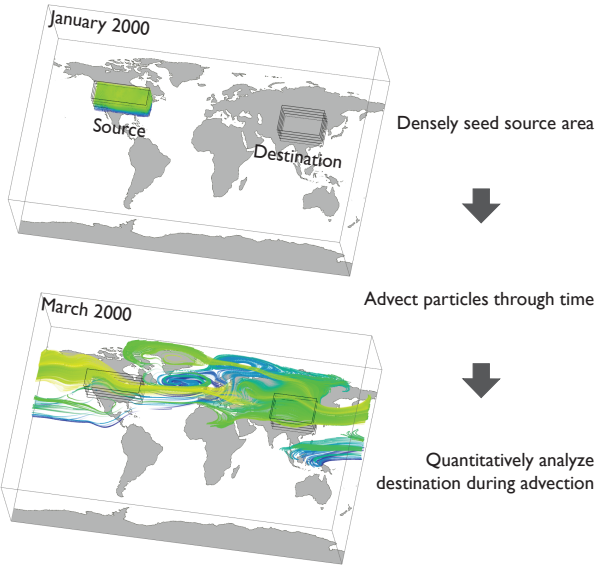


Figure 6: Detecting correlations in atmospheric flow from source to destination (teleconnection) is a challenging domain traversal problem. Given an unsteady flow field, what type of quantitative characteristics can be derived from interactions between a given source and destination?

5.1 GEOS-5 Ensemble Runs

NASA Goddard Space Flight Center (Lesley Ott) has provided us with state-of-the-art atmospheric simulation data (GEOS-5) for researching better and more sophisticated types of teleconnection and internal-model variability analysis. The GEOS-5 general climate model (GCM) uses a flux-form semi-Lagrangian finite-volume dynamical core with floating vertical coordinates developed by [17]. 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 $\approx$2.3 terabytes of data.

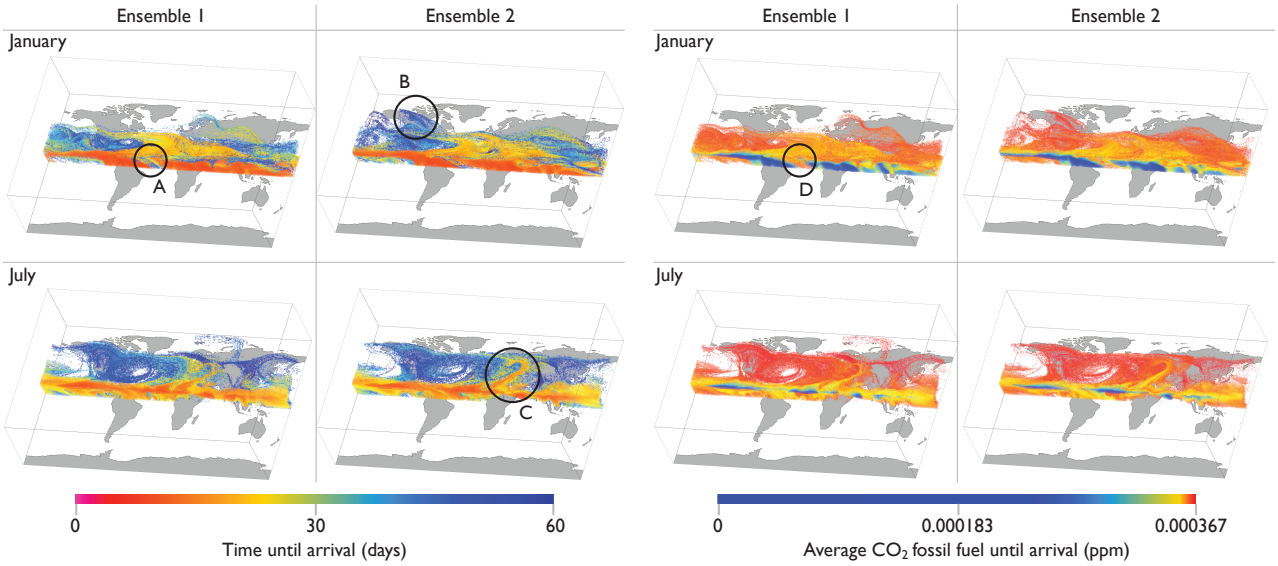


Figure 7: Three-dimensional direct volume renderings of the time until arrival and average CO_2 concentrations from January and July in two GEOS-5 ensemble runs. The circled areas are explained in Section 5.3.

5.2 Technical Use Case

The technical requirements of our studies involved quantitatively assessing relationships among the flows from different sources to different destinations. Figure 6 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 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 3, 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.

5.3 Application Impact – 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 7. 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 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. We plotted the absolute differences in probability in Figure 8. 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.

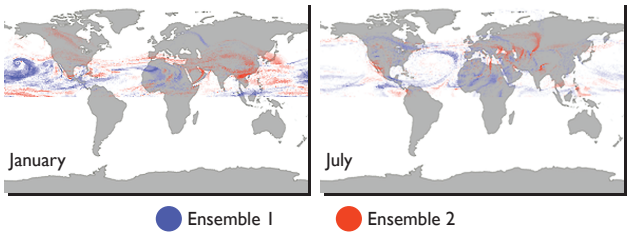


Figure 8: Differences between two ensemble runs are revealed by examining their probability distributions in the vertical direction. Color indicates the ensemble that had a higher probability of flow traveling from the Northern to Southern Hemisphere within two months. The opacity of this color is modulated by the absolute difference between the two ensembles, revealing the areas that are different.

5.4 Performance Evaluation

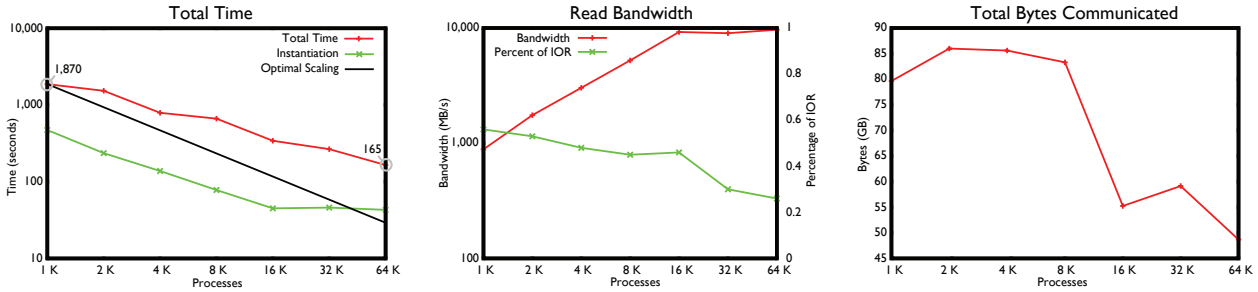
We have evaluated performance of the DStep Runtime in the context of our driving application. Our testing environment is *Intrepid*, an IBM BlueGene/P supercomputer at Argonne National Laboratory. Intrepid contains 40,960 nodes, each containing four cores. We used virtual node mode on Intrepid, which 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) Total Time (log-log scale) (b) Read Bandwidth (log-log scale) (c) Bytes Communicated

Figure 9: Performance using DStep to conduct entire analysis of inter-hemisphere exchange.

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 9(a). 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 9(b). 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 [3] on Intrepid. IOR provides a baseline benchmark of the maximum obtainable bandwidth. We used similar IOR parameters from the study in [16] 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 9(c) 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.

6. CONCLUSION

We have shown that the challenging problem of parallelizing domain traversal can be solved in an elegant manner. Our solution, which can efficiently hide and scale management of complex parallel tasks, has provided a new capability for data-intensive scientific analysis tasks. Along with being applicable to our system, we believe the two-tiered communication and work management strategy can be effectively utilized by other communication-bound applications.

Besides having a user interface that offers an expressive and succinct serial programming interface, another fruitful aspect for our application scientists is the ability to perform analysis directly after a simulation on its native architecture with its native data format. Furthermore, the ability to allow combination of unsteady flow features with local analysis of scalar quantities has also allowed our users to perform innovative full-range ensemble analysis.

In the future, we plan to explore out-of-core techniques, heterogeneous processor architectures, and integrate our system with fault-tolerant techniques. We are also interested in integrating our work with production analysis tools such as NCL and also adopting early users from other application domains.

7. ACKNOWLEDGMENT

This work is funded primarily through the Institute of Ultra-Scale Visualization (http://www.ultravis.org) under the auspices of the SciDAC program within the U.S. Department of Energy. Our work would be impossible without the facilities provided by the Argonne Leadership Computing Facility (ALCF) and the Oak Ridge Leadership Computing Facility (OLCF). We thank Lesley Ott and NASA Goddard Space Flight Center for providing us with GEOS-5 data. We also thank Robert Latham and Rob Ross at Argonne National Laboratory for their comments and help with improving I/O performance. We also acknowledge Scott Simmerman for his suggestions that improved the work.

8. REFERENCES

[1] Boost C++ libraries. `http://www.boost.org`.

[2] IDL: Interactive data language. `http://www.ittvis.com/idl`.

[3] IOR benchmark. `http://www.cs.sandia.gov/Scalable_IO/ior.html`.

[4] NCL: NCAR command language. `http://www.ncl.ucar.edu`.

[5] Protocol Buffers: Google's data interchange format. `http://code.google.com/p/protobuf`.

[6] Visit: Software that delivers parallel interactive visualization. `https://wci.llnl.gov/codes/visit`.

[7] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified data processing on large clusters. In *OSDI '04: Sixth Symposium on Operating System Design and Implementation*, 2004.

[8] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified data processing on large clusters. *Communications of the ACM*, 51:107–113, January 2008.

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

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

[11] Kui Gao, Wei keng Liao, Arifa Nisar, Alok Choudhary, Robert Ross, and Robert Latham. Using subfiling to improve programming flexibility and performance of parallel shared-file i/o. *International Conference on Parallel Processing*, pages 470–477, 2009.

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

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

[14] Wesley Kendall, Markus Glatter, Jian Huang, Tom Peterka, Robert Latham, and Robert Ross. Terascale data organization for discovering multivariate climatic trends. In *SC '09: Proceedings of ACM/IEEE Supercomputing 2009*, Nov. 2009.

[15] Wesley Kendall, Jian Huang, Tom Peterka, Rob Latham, and Robert Ross. Visualization viewpoint: Towards a general I/O layer for parallel visualization applications. *IEEE Computer Graphics and Applications*, 31(6), Nov./Dec. 2011.

[16] Samuel Lang, Philip Carns, Robert Latham, Robert Ross, Kevin Harms, and William Allcock. I/O performance challenges at leadership scale. In *SC '09: Proceedings of ACM/IEEE Supercomputing 2009*, 2009.

[17] S-J Lin. A "vertically lagrangian" finite-volume dynamical core for global models. *Monthly Weather Review*, 132:2293–2307.

[18] Jay Lofstead, Fang Zheng, Qing Liu, Scott Klasky, Ron Oldfield, Todd Kordenbrock, Karsten Schwan, and Matthew Wolf. Managing variability in the I/O performance of petascale storage systems. In *SC '10: Proceedings of ACM/IEEE Supercomputing 2010*, 2010.

[19] Grzegorz Malewicz, Matthew H. Austern, Aart J.C Bik, James C. Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. Pregel: A system for large-scale graph processing. In *Proceedings of the 2010 International Conference on Management of Data*, SIGMOD '10, pages 135–146, 2010.

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

[21] Tom Peterka, Robert Ross, B. Nouanesengsey, Teng-Yok Lee, Han-Wei Shen, Wesley Kendall, and Jian Huang. A study of parallel particle tracing for steady-state and time-varying flow fields. In *IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, May 2011.

[22] Steven J. Plimpton and Karen D. Devine. Mapreduce in mpi for large-scale graph algorithms. 2011.

[23] Dave Pugmire, Hank Childs, Christoph Garth, Sean Ahern, and Gunther H. Weber. Scalable computation of streamlines on very large datasets. In *Proceedings of the 2009 ACM/IEEE conference on Supercomputing*, 2009.

[24] Kurt Stockinger, John Shalf, Kesheng Wu, and E. Wes Bethel. Query-Driven Visualization of Large Data Sets. In *Proceedings of IEEE Visualization 2005*, pages 167–174. IEEE Computer Society Press, October 2005. LBNL-57511.

[25] Jeff Stuart and John Owens. Multi-GPU MapReduce on GPU clusters. In *IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, May 2011.

[26] Tiankai Tu, Charles A. Rendleman, David W. Borhani, Ron O. Dror, Justin Gullingsrud, Morten Ø. Jensen, John L. Klepeis, Paul Maragakis, Patrick Miller, Kate A. Stafford, and David E. Shaw. A scalable parallel framework for analyzing terascale molecular dynamics simulation trajectories. In *Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, SC '08, 2008.

[27] Hongfeng Yu, Chaoli Wang, and Kwan-Liu Ma. Parallel hierarchical visualization of large time-varying 3d vector fields. In *Proceedings of the 2007 ACM/IEEE conference on Supercomputing*, SC '07, pages 1–24, 2007.