PreDatA - Preparatory Data Analytics on Peta-Scale Machines

Fang Zheng[1], Hasan Abbasi[1], Ciprian Docan[2], Jay Lofstead[1], Scott Klasky[3], Qing Liu[3], Manish Parashar[2], Norbert Podhorszki[3], Karsten Schwan[1], and Matthew Wolf[1,3]

[1]College of Computing, Georgia Institute of Technology, Atlanta, GA 30332
Email: {fzheng,habbasi,lofstead,schwan,mwolf}@cc.gatech.edu
[2]Center for Autonomic Computing, Rutgers University, Piscataway, NJ 08854
Email: {docan,parashar}@cac.rutgers.edu
[3]Oak Ridge National Laboratory, Oak Ridge, TN 37831
Email: {liuq,klasky,pnorbert}@ornl.gov

***Abstract*—Peta-scale scientific applications running on High End Computing (HEC) platforms can generate large volumes of data. For high performance storage and in order to be useful to science end users, such data must be organized in its layout, indexed, sorted, and otherwise manipulated for subsequent data presentation, visualization, and detailed analysis. In addition, scientists desire to gain insights into selected data characteristics 'hidden' or 'latent' in the massive datasets while data is being produced by simulations. PreDatA, short for Preparatory Data Analytics, is an approach for preparing and characterizing data while it is being produced by the large scale simulations running on peta-scale machines. By dedicating additional compute nodes on the peta-scale machine as staging nodes and staging simulation's output data through these nodes, PreDatA can exploit their computational power to perform selected data manipulations with lower latency than attainable by first moving data into file systems and storage. Such in-transit manipulations are supported by the PreDatA middleware through RDMA-based data movement to reduce write latency, application-specific operations on streaming data that are able to discover latent data characteristics, and appropriate data reorganization and metadata annotation to speed up subsequent data access. As a result, PreDatA enhances the scalability and flexibility of current I/O stack on HEC platforms and is useful for data pre-processing, runtime data analysis and inspection, as well as for data exchange between concurrently running simulation models. Performance evaluations with several production peta-scale applications on Oak Ridge National Laboratory's Leadership Computing Facility demonstrate the feasibility and advantages of the PreDatA approach.**

I. Introduction

Scientific applications running on High End Computing (HEC) platforms can generate large volumes of output. As these grow to peta-scale and beyond, fast write and read accesses to massive data are becoming increasingly important, both to speed up the simulation and to accelerate exploration of data. A prerequisite to data exploration is that data is prepared in terms of data layout, indexing, and annotation. For example, some analysis tools prefer data to be laid out as contiguous arrays for quick loading [51], and queries can be accelerated if data is properly sorted and indexed [44]. In other words, appropriate data preparation is critical for data analytics, inspection, or visualization to operate. Finally, 'hidden' in the large data sets being output by scientific simulation are latent data characteristics of interest to end users, an example being statistical measures that can be used to validate the veracity of the ongoing simulation, gain understanding of the simulation progress, and potentially, take early action when the simulation operates improperly [19].

The object of our research and topic of this paper is the development of efficient methods that properly prepare data for subsequent inspection, storage, analytics, and even for input into concurrent simulation models (e.g., as in climate modeling). Our approach associates such data preparation with the output actions taken by HEC codes in ways that speed up output actions and thus improve application performance using minimal machine resources. The software artifact developed and used for these purposes is the PreDatA middleware. PreDatA provides scalable and flexible ways of associating data preparation operations with the I/O actions of HEC applications by generalizing the I/O stack used by HEC codes taking advantage of the ADIOS I/O library [29] used in a wide variety of peta-scale codes. With this enhanced I/O stack, output performance is improved by writing data to files using intermediate, log-structured data, avoiding the overheads caused by synchronization and meta-data generation [30] experienced when using standard file formats like HDF-5. At the same time, the use of these formats enables efficient operations on output data via predefined or user-provided computational functions. These functions are performed while I/O is ongoing by staging data to where PreDatA can leverage the computational power of selected machine nodes supporting I/O and/or connected to the storage subsystems. Further, by using PreDatA to index or properly annotate data, a reduction in the volume of subsequent reads performed by scientific workflows engaged in data analysis can be achieved. This improves performance by limiting interference at the parallel file system due to simultaneous writes used by output and reads used by scientific workflows.

The PreDatA middleware exploits the additional computational and memory resources provided by a staging area resident on the peta-scale machine. Output data are moved from compute to staging area nodes asynchronously to reduce write latency. PreDatA operations are applied to data prior to leaving the compute node and/or on data buffered in the staging area. The middleware provides a pluggable framework for executing user-defined operations such as data re-organization, real-time data characterization, filtering and reduction, and select analysis (or pre-analysis). These are specified in ways natural to the 'streaming' context. Despite this rich functionality, PreDatA offers levels of performance not provided by current file system-based approaches to analyzing output data, as shown with extensive experiments in this paper.

PreDatA performance is evaluated with several production peta-scale applications on Oak Ridge National Laboratory's Leadership Computing Facility platform. For one application, GTC [22], at the scale of 16,384 compute cores and with 1.5% additional resource usage, PreDatA hides write latency by up to 99.9%, improves total simulation time by 2.7%, and achieves a 1.5% saving in total CPU usage compared with performing pre-analytics in the compute nodes. In this experiment, PreDatA generates scientifically meaningful statistics from the 260GB output data in one simulation time step in about 40 seconds. For another application, Pixie3D [10], using PreDatA to re-organize the array layout of output data from 16,384 cores improves subsequent read performance for these output files by 10 times compared to when no such reorganization is performed. At the same time, total execution time of the simulation is increased by 1% with only 0.7% additional resource usage.

The remainder of the paper is organized as follows. Section II introduces the data management challenges for the two motivating applications. Sections III and IV present the design and implementation of PreDatA, respectively. Section V applies the PreDatA approach to the two driver applications, and Section VI evaluates the resulting performance demonstrating the advantage over other online and/or offline approaches. Section VII summarizes related work, and Section VIII concludes the paper.

II. Application Drivers

The development of PreDatA has been driven by the output and analysis needs of two production peta-scale codes, GTC and Pixie3D, both of which are capable of scaling to tens of thousands of cores and generating Terabytes of data in typical production runs.

A. The GTC Fusion Modeling Code

The Gyrokinetic Toroidal Code (GTC) [22] is a 3-Dimensional Particle-In-Cell code used to study micro-turbulence in magnetic confinement fusion from first principles plasma theory. It outputs particle data that includes two 2D arrays for electrons and ions, respectively. Each row of the 2D array records eight attributes of a particle including coordinates, velocities, weight, and particle label. The last two attributes, process rank and particle local ID within the process, together form the particle label which globally identifies a particle. They are determined on each particle in the first simulation iteration and remain unchanged throughout the particle's lifetime. These two arrays are distributed among all cores and particles move across cores in a random manner as the simulation evolves resulting in an out-of-order particle array. In a production run at the scale of 16,384 cores, each core can output two million particles roughly every 120 second resulting in 260GB of particle data per output. GTC employs the ADIOS BP format [29], a log-structured, write-optimized file format for storing particle data.

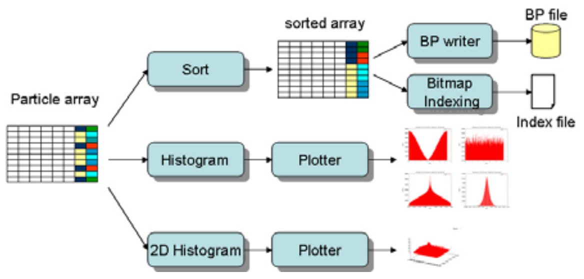


Fig. 1. Illustration of PreDatA Operations on GTC Particle Data

As illustrated in Fig. 1, three analysis and preparation tasks are performed on particle data. The first involves tracking across multiple iterations a million-particle subset out of the billions of particles, requiring searching among the hundreds of 260GB files by the particle label. To expedite this operation, particles can be (and for our example are) sorted by the label before searching. The second task performs a range query to discover the particles whose coordinates fall into certain ranges. A bitmap indexing technique [44] is used to avoid scanning the whole particle array and multiple array chunks are merged to speed up bulk loading. The third task is to generate 1D histograms and 2D histograms on attributes of particles [20] to enable online monitoring of the running GTC simulation. 2D histograms can also be used for visualizing parallel coordinates [20] in subsequent analysis.

B. The Pixie3D Code

Pixie3D [10] is a 3-Dimensional extended MHD (Magneto Hydro-Dynamics) code that solves the extended MHD equations in 3D arbitrary geometries using fully implicit Newton-Krylov algorithms. Pixie3D employs multigrid methods in computation and adopts a 3D domain decomposition. The output data of Pixie3D consists of eight, 3D arrays that represent mass density, linear momentum components, vector potential components, and temperature, respectively.

As illustrated in Fig. 2, various diagnostic routines are performed on Pixie3D output data to generate derived quantities such as energy, flux, divergence, and maximum velocity for diagnostics purpose. These derived quantities, along with the raw output data, are then read by visualization tools like VisIt for interactive visual data exploration. Pixie3D employs the BP file format for fast write performance. Array layout re-organization is performed to speed up read access.

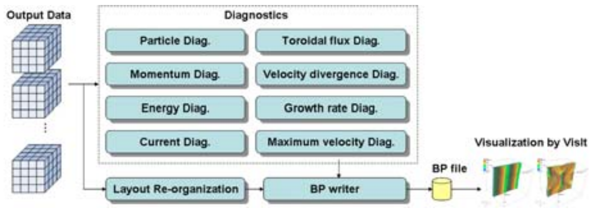


Fig. 2. Illustration of PreDatA Operations on Pixie3D Output Data

C. *Using the Staging Area for Flexible Scalable I/O and Pre-Data Analytics*

Conventionally, data preparation and analytics are performed either in compute nodes where the simulation is running (see Fig. 3(a)) or offline (see Fig. 3(b)):

In-Compute-Node approach: operations are performed in the compute nodes where output data is generated. The processed output is then written to the parallel file system.

Offline approach: the simulation dumps data to a parallel file system. Analysis codes running on other resources read such data and operate on it.

These two approaches to processing simulation output data differ in terms of their respective costs and limitations. For the In-Compute-Node approach, the overhead of data processing operations is visible to the simulation with consequent expenses in terms of CPU hours at scale. Performance advantages result if In-Compute-Node actions reduce output volumes, but severe performance penalties arise if data processing operations do not scale with the simulation. For the Offline approach, if the data volume is large, intermediate files may consume considerable storage resources, and parallel file system write and read times can be dominant causing high latencies and unacceptable levels of perturbation of peak file system performance. Therefore, it is clear that additional methods are needed to satisfy the I/O and data processing needs of the two representative peta-scale codes mentioned above.

One such method is the *Staging Area* approach shown in

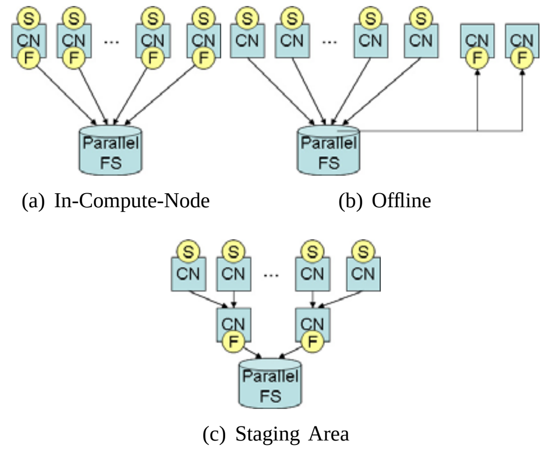


Fig. 3. Alternative Data Processing Methods for Scientific Simulations. ‘CN’ denoes compute tnode, ‘S’ denotes simulation, and ‘F’ denotes data operation

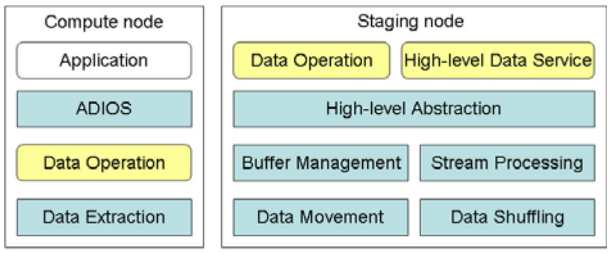


Fig. 4. PreDatA Middleware Architecture

Fig. 3(c). In this approach, a reasonable number of compute nodes are reserved as a Staging Area for staging data and hosting operations to apply to staged data before it reaches storage. Asynchronous execution within the Staging Area hides the processing costs from the simulation and affords an opportunity to employ less scalable operations ‘at scale’ since the Staging Area is small in comparison to the number of compute nodes being used (e.g., using a ratio of 128:1 for compute cores to staging cores). It is also possible to reduce disk accesses by pre-processing data so as to permit later analytics to focus on the data that is most relevant. Using these insights, the PreDatA middleware exploits the benefits of the Staging Area approach.

III. PreDatA Middleware Design

The PreDatA middleware design augments the current I/O stack on HEC platforms with data staging and in-transit processing capabilities by exploiting computational resources in both compute nodes and the staging area for preparatory data analytics.

As shown in Fig. 4, the PreDatA middleware resides in both the compute nodes on which the application runs and the staging nodes. Operations can be hosted in either location. When the application performs I/O actions, PreDatA acquires output data through the ADIOS I/O interface [27], stages data from compute nodes to staging nodes and performs in-transit data processing along the data flow.

There are several key features of PreDatA:

Asynchronous data movement. Data movement from compute to staging nodes is performed asynchronously to hide write latency from the simulation at a moderate cost of data buffering in the compute nodes. PreDatA explicitly schedules such asynchronous data movement to minimize interference with the simulation’s communications.

Pluggable pre-data analytics. PreDatA provides a pluggable framework making it straightforward for end users to specify, deploy, and debug data processing operations. The programming interface is general enough to implement a variety of operations, including data re-organization, real-time data characterization, filtering and reduction, and lightweight data analysis.

User-defined operations. The middleware supports user-defined data operations with common services for data access, buffer management, scheduling and executing data processing actions, and high performance data exchange and synchronization across staging nodes.

Higher-level Data Services. The middleware also provides supports for building higher-level data services ranging from data indexing and query to inter-application data exchange.

Integrated operations, separated from application codes. PreDatA hides from data processing codes the complexities of data access in the staging area while meantime offering high performance through permitting such codes to directly access buffered data. I/O stack integration is performed so as to separate application codes from the potential complexities of data processing actions.

IV. PreDatA Middleware Implementation

The PreDatA middleware's implementation leverages our earlier work [2] on efficiently scheduling data movement from compute nodes to the Staging Area. The EVPath [16] high performance event system is used for efficient data buffering and manipulation in the Staging Area. The FFS [17] binary data encoding facility is used for in-transit data to provide PreDatA operations access to buffered data with rich meta-data information. The ADIOS [27] library is the basis for integrating PreDatA with application I/O.

A. Data Extraction and Movement

PreDatA uses the ADIOS I/O library as the basis for both the simulation's I/O stack and for PreDatA operations to access data output by the simulation. ADIOS allows for introducing PreDatA processing into the compute nodes without requiring changes to application codes, thereby insulating application code from the complexities of additional processing actions in the I/O stack. ADIOS also explicitly defines the structure of application's output data, and such meta-data information is used as a common interface for application and PreDatA operations to coordinate sharing data.

PreDatA also uses the scheduled, asynchronous RDMA [7] operations explained in [2] for extracting and moving data from compute nodes to staging nodes. The use of asynchronous RDMA reduces the write latency visible at compute nodes and scheduling such RDMA operations helps minimize interference between communications performed by the simulation vs. those used for output. This is particularly important when output data movement overlaps collective communications among compute nodes and thereby may cause severe perturbation on simulation performance.

B. In-transit Data Processing along Data Flow

PreDatA augments the I/O stack resulting in the overall data flow shown in Fig. 5. There are four stages in the data flow: (1) data extraction and optional local processing in compute nodes, (2) optional aggregation in staging nodes, (3) asynchronous data movement from compute nodes to staging nodes, and (4) data stream processing in staging nodes.

When I/O is triggered in the compute nodes, output data is passed to the PreDatA runtime in the compute nodes via the ADIOS interface (shown as Stage 1 in Fig. 5). Typical output data of compute nodes consists of one or more scalars, local arrays, and/or partial chunks of global arrays. PreDatA executes

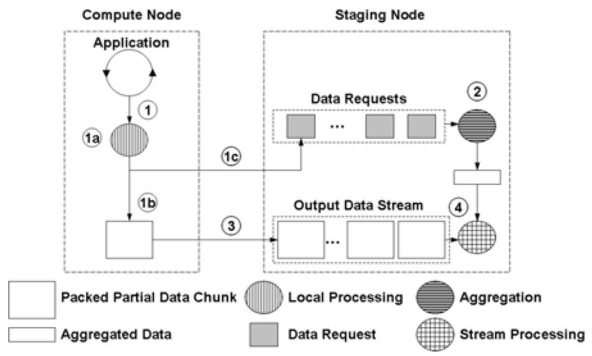


Fig. 5. Overall Data Flow of PreDatA

a user-defined routine, if provided, on the local output data (shown as Stage 1a in Fig. 5). This constitutes a optional first pass of processing on the output. Possible operations include generating meta-data such as array dimension information, calculating local min/max values of partial array chunks, and filtering out undesired regions. All output data (scalars, local arrays, partial chunk of global arrays) are then packed into a contiguous buffer, termed a *packed partial data chunk*, using the FFS [17] binary data encoding facility (shown as Stage 1b in Fig. 5). The structure of each packed partial data chunk is compatible with the ADIOS output data group definition, and metadata about the data structure is embedded in the packed partial data chunk. A data fetch request is sent to the staging node chosen by a user-overridable function *Route()* (shown as Stage 1c in Fig. 5). PreDatA provides an interface that permits the data operation in Stage 1a to attach small partial results to data fetch requests, allowing for additional flexibility in the staging area. The compute node then resumes computation while the data movement and operations are performed.

In the Staging Area, each staging node waits for data fetch requests from compute nodes. When the staging node finishes gathering requests from all compute nodes it serves, it extracts partial results attached to requests, if there are any, and performs user-defined aggregation functions on them to generate aggregated results such as global array size and offsets, prefix sum, and global min/max values (shown as Stage 2 in Fig. 5). Each staging node then begins to fetch packed partial data chunks from compute nodes (shown as Stage 3 in Fig. 5). Data chunks are processed by staging nodes one by one in a streaming manner (shown as Stage 4 in Fig. 5) and the aggregated results generated in Stage 2 are accessible from the stream processing operations.

In summary, the PreDatA middleware provides two passes across an application's output data. The first pass optionally done on compute nodes is suitable for operations that do not require global communications and/or synchronization. The second pass performed on staging nodes, in a data streaming fashion, can be used to compute global data properties and/or to reorganize data for later storage. Data streaming is critical because it is unlikely for staging nodes to have sufficient memory to hold all of the raw data generated by multiple and, often, even single simulation output steps. As is shown in Section V, this two-pass model is sufficient to implement a variety of useful data pre-analytics.

C. Stream Processing in the Staging Area

As mentioned above, the output data of each compute node is packed into a contiguous memory buffer, i.e., a *packed partial data chunk* and moved in its entirety into the Staging Area. From the Staging Area's perspective, incoming data consists of a finite number of packed partial data chunks streamed from compute nodes participating in the I/O dump. When there are multiple staging nodes, the packed partial data chunks are split into multiple streams across these nodes.

Each staging node is responsible for processing a stream of packed partial data chunks with each chunk from one compute process, which is the forth stage of the dataflow as shown in Fig. 5. The processing of such a stream is divided into five phases (as shown in Fig. 6):

Initialize: the *Initialize()* function of each operation is executed once at the beginning of an I/O dump with aggregated result data generated from the pre-fetch process (as shown in Fig. 5) as a parameter to initialize the operation-specific data structure and for other setup tasks.

Map: the *Map()* function of each operation is executed on each packed partial data chunk. Intermediate results are tagged and stored in a local buffer.

Shuffle: when the last chunk within the I/O dump is processed, partial results are combined locally, if the *Combine()* function is provided. Each staging node applies the *Partition()* function to route intermediate result to other staging nodes according to the associated tag.

Reduce: each staging node groups intermediate results, both local and those received from other staging nodes, by associated tags and then performs the *Reduce()* function on each group of intermediate results to aggregate results.

Finalize: when the Reduce phase finishes, each staging node executes the *Finalize()* function of each operation, which writes final results to disk, feeds data to other consumers, and/or performs necessary cleanup.

Note that this data processing model is similar to the MapReduce [11] paradigm, with five notable differences: (1) the data processing model requires that the operations only need to read data once so that data can be processed in a streaming fashion, (2) the addition of the initialize and finalize phases, (3) users can can customize the data shuffling (see Section IV-D), (4) there is no central master that has global knowledge of data location and task progress, and (5) there is only one round of data exchange.

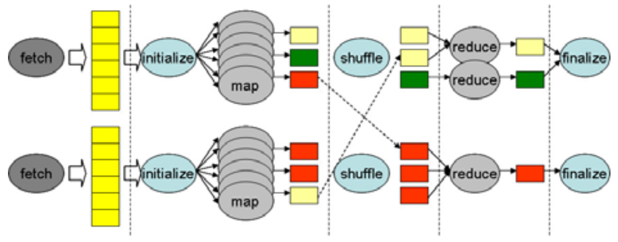


Fig. 6. Stream Processing in the Staging Area

A user can plug their own data operations into PreDatA middleware by implementing the functions mentioned above. They may also customize data movement scheduling policy to place data chunks within the data stream into specific order (e.g., fetching chunks in order of compute nodes' MPI rank for calculating prefix sum). Detailed API definitions are listed in Table I in Appendix.

The staging area is running as a separate MPI program launched independently with the simulation. Each MPI process runs on one staging node. Within each staging node, there are multiple threads in each MPI process that execute different pieces of the execution flow shown in Fig. 6 to exploit concurrency.

D. Data Shuffling

In the Shuffle phase of stream processing, the PreDatA runtime system adopts a general ring-based communication paradigm for shuffling data among staging nodes, as shown in Fig. 7. Since every pair of nodes needs to exchange data, a series of $N-1$ messaging rounds with the 'distance' between each pair starting at 1 and going to $N-1$ nodes. This scheme is also used in the HiMach framework [46].

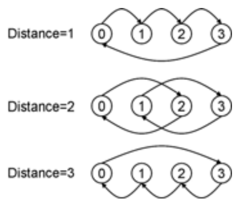


Fig. 7. Data Shuffling among Staging Nodes

Custom inter-staging-node communication can be employed using MPI for explicit message passing and using the provided access for the intermediate data buffers.

E. Buffer Management

On compute nodes, additional buffering is needed to hold packed partial data chunks with a buffer size roughly equal to the output data sizes and configurable through the ADIOS configuration file. On staging nodes, all incoming packed partial data chunks are stored in buffers provided by the PreDatA runtime. The runtime maintains reference counts for recycling a buffer when the input chunk has been processed by all operations. For intermediate data received from other staging nodes during shuffling, data operation routines indicate to the runtime system when to recycle those buffers. Private buffers maintained by individual operations are its own responsibility. The latter is consistent with a basic assumption about the staging area made by the PreDatA middleware, which is that all data is maintained in in-core buffers. This means that for extremely large datasets, it is the responsibility of specific PreDatA operations to be aware of and deal with memory limitations. For assistance, PreDatA provides explicit memory manipulation routines that retrieve information about available memory space and allocate/de-allocate buffer space. The in-core assumption is reasonable for our target application workloads and platform, since there are no local hard disks or Solid State Drives (SSD) [36] attached to staging nodes in the tested environment. If such were present or if there were fast access to a shared parallel file system as an external buffer without concerns about perturbing output performance [28], buffer management should be extended to include out-of-core functionality.

F. The DataSpaces Global Data Knowledge Service

The purpose of this section is to show that the 'in-transit' and 'online' approach of data output and manipulation used in PreDatA can be used to implement the model-to-model communications used in high performance coupled codes [3], [53]. Toward that end, we are integrating into PreDatA the high level 'DataSpaces' data indexing and query services. The intent is to demostrate the extent to which pre-data analytics can be enriched to also support the rich and flexible methods for online access to generated data required for general inter-application interactions. DataSpaces provides higher level programmable and managed services for (1) data sharing – between operations working on a common set of data; (2) data redistribution – between operations with different data discretization and running on a different number of processors; (3) data indexing – data hashing for fast access; and (4) data querying – application data retrieval based on custom selectors. With (1)–(4), it provides the abstraction of a virtual semantically-specialized shared data space that can be asynchronously and flexibly accessed using simple yet powerful operators (e.g., *put()* and *get()*) that are agnostic of the location or distribution of data.

DataSpaces incorporates flexible mechanisms that can fetch and index data, on-the-fly, from multiple different sources, as shown in Fig. 8. It can even extract data directly from a running application. It can store incoming data locally in the staging area or share it with the collaborating frameworks, index it for fast access, and serve it in response to logged or incoming user queries. Datasets composed of both, homogeneous data types, e.g., doubles, floats or integers, as well as heterogeneous data types, e.g., aggregate structures of doubles, floats or integers, are supported.

DataSpaces implements a flexible querying mechanism that allows applications to request individual values as well as contiguous regions of data based on simple descriptors that are semantically meaningful to the application. For example, in the case of typical simulation data, data can be indexed based on its geometric coordinates within the multi-dimensional discretization used by the simulation allowing it to be queried using geometric descriptors that are meaningful to the application. Queries may be generated by users or by other applications. For example, each instance of a distributed querying application running on multiple nodes can query distinct and relevant sub-regions of data as needed. Similarly, a user can query sub-regions of interest only when they are needed or can register sub-regions of interest for continuous querying. In the latter case, for example, the user is notified automatically every time new data items that lie within the regions of interest are inserted into the space.

DataSpaces also supports aggregation and reduction queries. For example, queries can request the maximum or minimum value for a particular field in a given sub-region, or the average value of a specified field within a given region. Note that, from the perspective of a querying end user or application, the querying and data transfer process is transparent and independent of data distribution, i.e., the data comprising the query response may come from different nodes of the application that generated the data and served by different DataSpaces framework nodes.

DataSpaces complements the indexing and querying services with an in-memory data storage service. The storage service can be used to maintain private copies of the data extracted directly from a running application or store shared copies of the data processed by collaborating frameworks. The storage service incorporates a data coherency protocol that manages interactions with the data and ensures data integrity when multiple entities simultaneously query the data.

DataSpaces maintains load balancing at two levels. First, the storage service distributes the data evenly across the DataSpaces nodes, and second, the indexing service dynamically distributes the index metadata across the DataSpaces nodes to distribute incoming queries across these nodes.

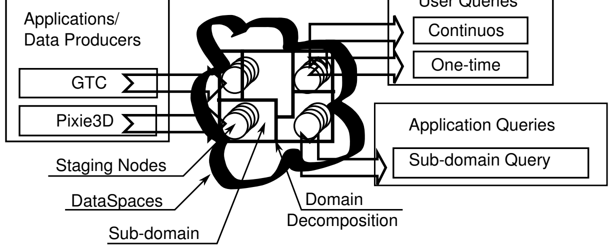


Fig. 8. Example of application to application coupling implemented using the querying framework.

V. Application Experiences

The two driver applications mentioned in Section II are used to evaluate the PreDatA middleware and higher level DataSpaces services. This section describes the implementation of various data operations for the two applications with the PreDatA middleware.

A. GTC

Sorting, histogram, and 2D histogram operations are tested for GTC. The sorted particle data are written into BP files from the Staging Area. The Bitmap index is not explored for paper space reasons.

Sorting and BP Writer: Two PreDatA operations are used to sort and store electron and ion arrays, respectively. For each operation, the first pass on the data, done in compute nodes, gathers the sizes of all local arrays. Based on that, the *Initialize()* function pre-allocates buffers to hold particles belonging to the local staging node and calculates prefix-sums for a subsequent merge sort. The *Map()* function emits each particle as a key-value pair, in which the key is the process rank attribute of the particle and the value consists of all eight attributes of each particle. The *Partition()* function routes particles to their respective staging nodes where they are grouped by key. The *Reduce()* function performs a merge sort and copies each particle to its proper location in the global array. The *Finalize()* function writes the sorted global array to a single BP file using ADIOS MPI-IO output method and frees the local buffers.

Histogram: The first pass on the data, done in compute nodes, gathers the min/max values of all attributes in local arrays. The *Initialize()* function calculates global min/max values from that and sets up bin boundaries. For each attribute of each particle, the *Map()* function determines which bin the attribute value falls into and emits a key-value pair where the key is in the form of (attribute ID, bin ID) and the value equals to 1. The *Combine()* function merges partial results. The *Partition()* function returns ‘staging node 0’ so that all intermediate count values are sent to that node. The *Reduce()* function merges intermediate counts to determine global counts. The *Finalize()* function generates a histogram graph for each attribute from global count values using the PGPLOT [38] library and saves both images and raw binary histogram data.

2D Histogram: the 2D histogram operation calculates a 2D histogram for each pair of adjacent attributes. The implementation is similar to Histogram except that it uses (attribute ID, bin ID of first attribute, bin ID of second attribute) as the key to tag the intermediate results.

B. Pixie3D

An array layout re-organization operation is created for Pixie3D. This operation merges partial array chunks into larger contiguous ones for each of the eight 3-Dimensional arrays in Pixie3D’s output and writes merged arrays to a BP file. Implementation of various diagnostic routines for Pixie3D output data are ongoing.

Array Layout Re-organization: the *Route()* function performed on each compute node redistributes 3-Dimensional array data to staging nodes along the minor dimension of 3-Dimensional arrays. This data redistribution scheme makes sure array chunks moved to the same staging node are adjacent. The *Initialize()* function pre-allocates a contiguous buffer to hold organized arrays. The *Map()* function copies each slice of an array to its destination in the buffer. There is no data exchange among staging nodes because of the redistribution schemes used on compute nodes. The *Reduce()* function is empty. The *Finalize()* function writes reorganized arrays to a single BP file using the ADIOS synchronous MPI-IO method and frees the buffer.

VI. Performance Evaluation

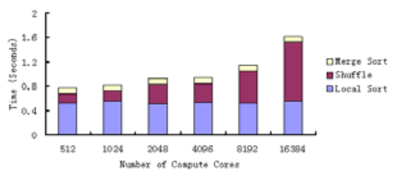
As described above, the placement of operations can greatly affect their performance, the timeliness of the output, and impact the overall system performance. By evaluating several different operators using different placement choices, the benefits of the flexible placement and in-transit processing is demonstrated. Operations for two peta-scale applications, GTC and Pixie3D, are used to evaluate the PreDatA middleware. Performance of DataSpaces global data knowledge service is also evaluated with GTC to demonstrate the feasibility of building high-level data services with PreDatA.

A. Experimental Environment

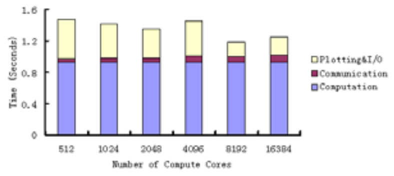
Experiments are run on Oak Ridge National Laboratory’s Cray XT4/XT5 Jaguar platform. The XT5 partition contains 18,688 compute nodes in addition to dedicated login/service nodes. Each compute node contains two quad-core AMD Opteron 2356 (Barcelona) processors running at 2.3 GHz, 16GB of DDR2-800 memory, and a SeaStar 2+ router. The resulting partition contains 149,504 processing cores, more than 300TB of memory, over 6 PB of disk space, and a peak performance of 1.38 petaflop/s. The XT4 partition contains 7,832 compute nodes in addition to dedicated login/service nodes. Each compute node contains a quad-core AMD Opteron 1354 (Budapest) processor running at 2.1 GHz, 8 GB of DDR2-800 memory, and a SeaStar2 router. The resulting partition contains 31,328 processing cores, more than 62 TB of memory, over 600 TB of disk space, and a peak performance of 263 teraflop/s. For each case described below, we run each test case 5 times and use the best samples in both In-Compute-Node and Staging configuration for plotting to control for interference in the shared experimental environment.

B. GTC Performance

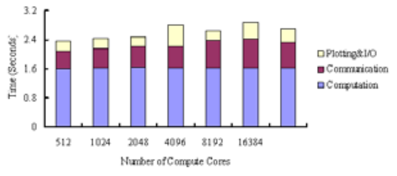
The GTC experiments are performed on the XT5 partition of Jaguar. As is typical with a production run, the GTC jobs are configured to deploy a single MPI process per node that spawns 8 OpenMP worker threads, one per core. I/O is only performed by the MPI processes. For GTC, three operations are tested: particle sorting, histogram generation, and 2D histogram generation. Each of these operators is applied to both the electron and ion particle arrays output with I/O interval of about every 120 seconds. Weak scaling is employed with 132MB total written per process for the two particle arrays. The Staging Area is configured to deploy 2 MPI processes per node with 4 worker threads per MPI process. The size of the Staging Area is adjusted to maintain a ratio of compute cores to staging cores of 64:1 (1.5%). That is, for each 64 nodes with compute processes (512 OpenMP worker threads), 1 node (2 staging processes for a total of 8 worker threads) is employed for staging. The tests are performed in two ways. First, all operations are performed in compute nodes and use synchronous MPI-I/O to write results (‘In-Compute-Node’ configuration). Then they are performed in Staging Area (‘Staging’ configuration).



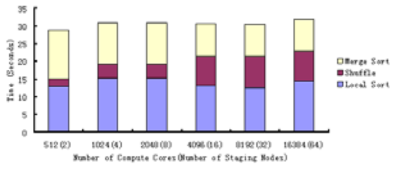
(a) Sorting in Compute Node



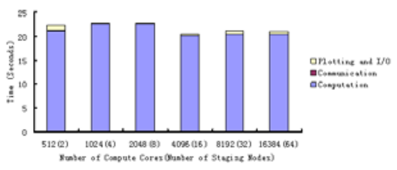
(b) Histogram in Compute Node



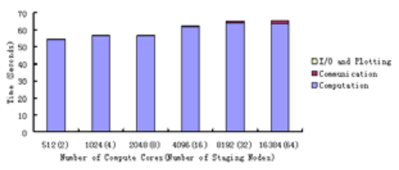
(c) 2D Histogram in Compute Node



(d) Sorting in Staging Area



(e) Histogram in Staging Area



(f) 2D Histogram in Staging Area

Fig. 9. Timing Results for Individual Operations

1) Performance of Individual Operations: In this section we study the performance results for each operation.

Sorting Operation: Fig. 9(a) and 9(d) compare the performance of sorting using the In-Compute-Node configuration and the Staging configuration. Sorting is an example of communication intensive operations because it involves all-to-all communication and has minimal computational demands. When sorting in compute nodes, the data shuffle time among compute nodes increases dramatically as the operation scales and such cost is visible to simulation. On the other hand, sorting in the Staging Area takes at most 33 seconds at all scales, which is much less than the 120-second I/O interval. Therefore, performing sorting operation in Staging Area can mask the cost of sorting from simulation because of asynchrony. There are, however, 30 seconds of latency in Staging configuration, two orders of magnitude longer than the In-Compute-Node configuration. This tradeoff demonstrates the importance of placement: if the goal is to optimize simulation time, placing the sorting operation in Staging Area is better, but if the latency of generating sorted data is more critical, placing the operator in compute nodes is a better choice.

Histogram Operation: As shown in Figs. 9(b) and 9(e), the histogram operation is computation dominant with communication contributing only a very small portion of the total operation time. While performing this style of computation intensive operation in the compute nodes takes less wall clock time, the perturbations to the total simulation time can be much larger due to the impact of I/O operation for saving histogram results. The time for writing the 8 MB histogram files ranges from 0.25 seconds to 7 seconds, which adds to the total simulation time. This reveals a different advantage for the Staging configuration: insulating simulation from variation in file system performance. Since the increased cost of computing the histogram is hidden by the asynchronous data transfer and operation savings, using the Staging configuration is still generally advantageous. For those cases where one has computation-intensive operations without a subsequent I/O operation or if latency is very important, using the 'In-Compute-Node' configuration is superior.

2D Histogram Operation: Similar to the Histogram operation, the 2D Histogram operation is computation dominant, as shown in Figs. 9(c) and 9(f). While the computation and communication requirements for generating the 2D histograms is larger, the same conclusions can be drawn.

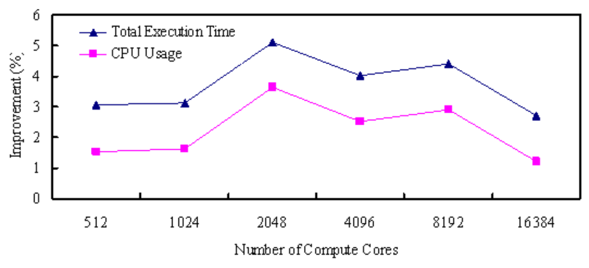
In summary, the results shown in this section demonstrate that for operations with different computation and communication characteristics, offloading operations from compute nodes to staging nodes generally helps mask the cost and variation of operation and associated I/O activity from simulation because of asynchrony but introduces longer latency for operation to finish because of the capacity mismatch between compute nodes and staging nodes. Depending on the latency requirements and variability in the system, performing these operations in a Staging configuration can contribute a performance improvement for some operations and insulation from system variability for others. In both cases, strict or weak latency requirements can override a short-term cost for an overall benefit.

2) Simulation Performance: This section evaluates the GTC simulation performance in two different configuration. Fig. 10(b) shows the total execution time of the GTC simulation for the two different configurations at various scales ranging from 512 to 16,384 compute cores. The Staging configuration improves the simulation's total execution time by 2.7% to 5.1% over the In-Compute-Node configuration (as shown in Fig. 10(a)).

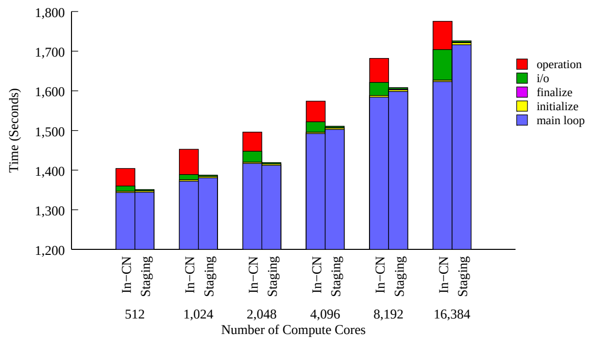
The breakdown of total execution time (shown in Fig. 10(b)) explains the performance advantage of the Staging over the In-Compute-Node approach:

Firstly, the Staging approach hides write latency via asynchronous data movement. For example, at the scale of 16,384 compute cores, 8.6 seconds are required, on average, to write 260GB of particle data with the ADIOS synchronous MPI-I/O method. The visible I/O blocking time with the Staging configuration is reduced to 0.30 seconds on average. This improvement of write latency increases with simulation scale.

Secondly, the Staging approach also insulates the simulation from the increasing time costs for performing the operations



(a) Improvement of GTC Simulation Performance and Cost



(b) GTC Total Execution Time Breakdown

Fig. 10. GTC Simulation Performance

as the simulation scales since these operations are done in the Staging Area concurrently and asynchronously with the simulation. For the In-Compute-Node configuration, the time spent in operations increases from 3.0% to 4.1% as the simulation scales from 512 to 16,384 cores. With the Staging approach, the simulation spends no time carrying out such operations. While it is true that the Staging Area experiences a larger proportional time in performing the operations, the time insulating effects of asynchronous I/O afford using more time without impacting the application wall clock time.

Thirdly, potential interference between asynchronous data movement with the simulation's communications is minimized by properly scheduling data movement. The comparison of main loop time for the two different configurations shows that Staging may slow down the computation due to contention on the shared network, especially at large scales. However, by properly scheduling data movement, this interference is controlled to be less than 6%.

Overall, the reduction in visible I/O and operation times on compute nodes outweighs the interference experienced by the simulation due to asynchronous I/O and the insulating effects of decoupling the simulation I/O from variations in the file system performance improves the total execution time and reduces variation in the performance in spite of some increased latencies for performing some styles of operations. In terms of total CPU usage cost, calculated as total simulation time multiplied by total cores used, the Staging configuration is less costly when compared with the In-Compute-Node configuration at all scales (as shown in Fig. 10(a)). There is a decline of savings from 8,192 to 16,384 cores mainly due to the interference of asynchronous data movement. At the scale of 16,384 compute cores, however, running the simulation with the Staging configuration still saves 98 CPU hours in total compared with the In-Compute-Node configuration for a 30-minute simulation run. This suggests that the Staging approach helps GTC achieve better scalability in terms of total cost of both simulation and data preparation.

3) Offline Operations Discussion: Considerations for using offline operations are different from online operations. Instead of compute time and communication load being dominant factors, data storage requirements and file system interference generated are major concerns. Typically, offline operations, while slower to perform and much longer latency to completion, can be done cheaply or free. For operations that do not generate a reduction in data and instead generate approximately equivalent data in a different organization, such as sorting and layout re-organization, an offline approach would cost additional storage resource for intermediate data and meanwhile impact the file system by reading all of the data and writing it again. For example, when running at the scale of 65,536 cores, the particle data of GTC is 1TB per I/O dump. Offline sorting would cost 1 TB additional storage space every 120 seconds and the entire 1 TB would have to be read back in before it is rewritten. This moves the data through the disk controllers three times rather than once. Secondly, given the huge volume of GTC data, the read and write latency would be hundreds of seconds making the offline approach unsuitable for online data monitoring. For these sorts of operations, in-transit data manipulation is a big win.

For operations like the histogram and 2D histogram, the advantage of in-transit is still present. Using the same 1 TB per I/O dump output, two trips through the disk controller are required. While the output of this style of operation is comparatively very small, the impact of reading all of the data to generate the histograms both generates potentially large latency and long-term impact to the file system performance.

4) Evaluation of the DataSpaces Global Data Knowledge Service: To evaluate if the DataSpaces query engine can service queries on particle data in a timely manner without blocking the simulation between two successive I/O operations, a prototype implementation of the DataSpaces indexing and querying service is deployed on the staging nodes. The particles output from the GTC application are first sorted using the sorting operation, and then indexed by DataSpaces based on the *local id* and *rank* attributes to create a $2 \cdot 10^6 \times 256$ 2-D domain space. This spaces is then uniformly distributed across the DataSpaces compute cores in the Staging Area. On average, at all simulation scales ranging from 512 to 16,384 cores, the time required to fetch data from the GTC simulation is 20.3 seconds, sorting takes 30.6 seconds, and indexing takes 2.08 seconds. In total, it takes no more than 55 seconds for DataSpaces to prepare the data for query.

A test querying application that queries the entire domain space is deployed on additional compute cores (referred to as 'querying application cores' in subsequent text). In the experiments, the querying application cores partition the particle data among themselves and issue 11 consecutive queries to disjoint regions of the data. The particle sub-regions is 200MB

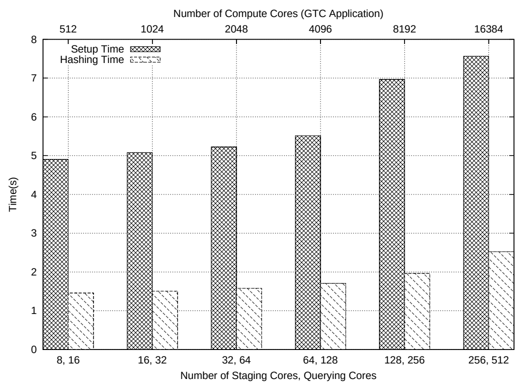


Fig. 11. Setup and Hashing Time

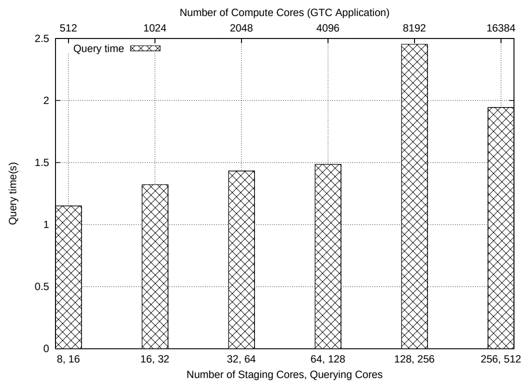


Fig. 12. Query Time

in size for each querying application core. Since no a-priori knowledge is assumed about the existence of the particles data or its distribution, the first query includes query setup operations such as hashing, data discovery, query routing and data retrieval, and is significantly more expensive to perform as seen in Fig. 11. However, it is a one-time cost and subsequent queries are much faster. The setup time shown in Fig. 11 is an average value across the number of querying application cores and the hashing time is an average over the number of setup queries received at each core running a DataSpaces server in the staging area.

The query execution time for different numbers of querying application cores is plotted in Fig. 12. The plotted times are an average over the number of queries executed and over the querying application cores. The query time increases with the number of cores used since the domain size increases and is mapped to a larger number of cores in the staging area. In the presented example, one instance of the querying application receives replies to its query from multiple cores in the DataSpaces. The longer query time for the 256 application querying cores is due to load variability and interferences in the host system – we are investigating this further.

Note that DataSpaces indexes particles data and responds to all queries in less than 80 seconds. Considering the I/O interval is about 120 seconds, such an online query service can function effectively and without blocking the simulation.

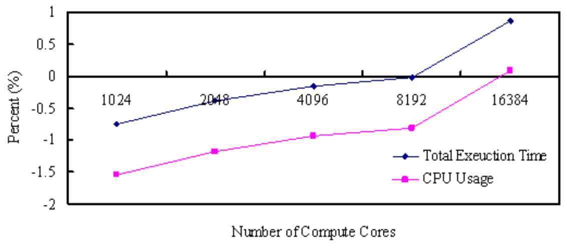
C. *Pixie3D Performance*

Pixie3D performance is evaluated on the XT4 partition of Jaguar. Production runs use one MPI process per compute core. The data output from each process mainly consists of eight double-valued arrays. Each local array is part of a 3D global array, respectively. The tested setting uses a 32x32x32 local array size, which is a typical setting for production runs. For each run, the simulation performs I/O about every 100 seconds. The ratio of compute cores to staging cores is maintained at 128:1 during weak scaling. Each process generates about 2 MB of data making this ratio workable.

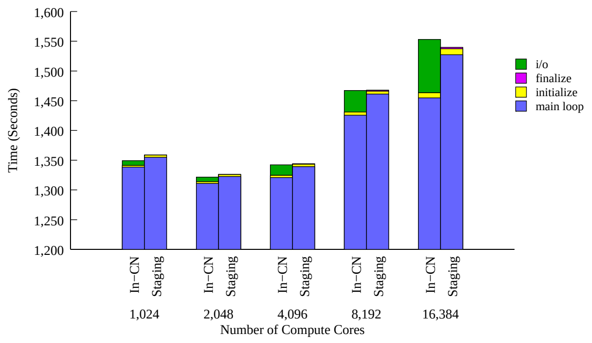
Pixie3D is tested with an In-Compute-Node configuration and a Staging configuration. For the In-Compute-Node configuration, each MPI process writes output data to a single BP file using the ADIOS synchronous MPI-IO method. This results in a file in which local array chunks are scattered. In the Staging configuration, output data of compute nodes are sent to the Staging Area where they are merged to form larger, contiguous global arrays.

Fig. 13(b) shows the total simulation execution time for both the In-Compute-Node and Staging configurations. The Staging configuration slows the simulation in most cases by 0.01% to 0.7% when compared against the In-Compute-Node configuration. Unlike GTC, Pixie3D does not have enough computation intensity for asynchronous I/O to be an effective technique for offloading data. In each iteration, the inner loop of pixie3d performs collective communications (MPI_Reduce and MPI_Bcast) multiple times and between the mass communications are computations that only last about 0.7 seconds making it difficult to overlap data movement with computation without impacting the intensive messaging. The results show that the main loop time is increased because of asynchronous data movement. Although the I/O blocking time is well hidden, since it is such a tiny portion of the total execution time, this savings cannot outweigh the slowdown of computation due to communication interference. The operations tested for the GTC application were all intended to be performed before any data analysis were performed in order to speed read operations. The same is true for this data reorganization operation. While GTC's operations were a win-win for both writing and reading at all scales, Pixie3D's data reorganization requires larger job sizes to reach a tipping point where simulation performance can be improved by employing staging. Figure 13(a) shows the total cost of CPU seconds. As the simulation scales up, the I/O overhead weighs more in total execution time, and hence the impact of computation caused by data staging becomes less evident. Overall, there is a trend that the cost of Staging approach catches up with that of In-Compute-Node approach with increased simulation scale.

It is worth examining the savings generated during reading operations due to the reorganized data. Fig. 14 shows the read performance on two files generated by two 4096-compute-core runs with Stingy and In-Compute-Node configuration, respectively. This result, along with the simulation cost shown in Fig. 13(a), shows that at the scale of 4096 compute



(a) Improvement of Pixie3D Simulation Performance and Cost



(b) Pixie3D Total Execution Time Breakdown

Fig. 13. Pixie3D Simulation Performance

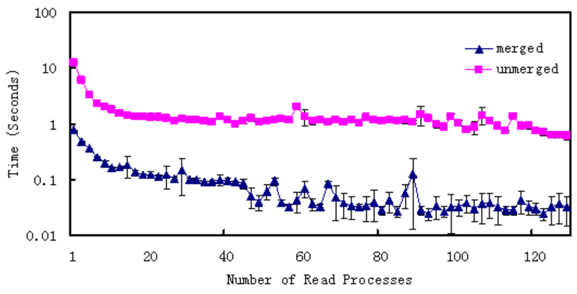


Fig. 14. Time to read one global array of one time step from two 80GB BP files. 'merged' denotes the read time from a file written from Staging Area and 'unmerged' denotes the read time from a file written from compute nodes directly. Both files are generated by 4096-compute-core runs.

cores, 0.93% additional cost in simulation yields 10 times improvement in read performance of output data. This saving is more evident as scale increases.

In summary, the performance results show that in-transit data manipulation enabled by PreDatA middleware can improve the latency to operation completion compared with offline approach, reduce overall wall clock time of simulation even compared to online configuration at large scales, and reduce the impact on the shared file system when compared against both online and offline configurations. It is also shown that high-level data services can be efficiently built on top of PreDatA middleware.

VII. Related Work

In this section we summarize previous research related to PreDatA work.

Scalable I/O and Data Analytics. Efficient access, understanding and management of voluminous and complex data generated by scientific simulations presents daunting challenges to both computational and computer scientists [18], [37]. Recent work in parallel file systems [8], [35], [50] and I/O middleware [21], [30], [40], [52], [54] aims at optimizing data storage and access for scientific application workloads. Beyond pure high I/O bandwidth, however, scientists also require complex data analysis, search, and visualization technologies to facilitate better understanding of their data. Specialized data preparation, such as sorting, filtering, and indexing, is needed before data can be understood or visualized [9], [41], [45]. Our work extends the I/O middleware stack to exploit computational power along the output data flow to perform data preparation, characterization, and reorganization, which would facilitate subsequent data analysis.

Data Staging and Offloading in supercomputers. Previous work on data staging and asynchronous I/O [4], [15], [24], [25], [32], [34], [43] derives substantial performance advantages from hiding I/O latency with asynchronous data movement. Our recent work [1], [2] shows the importance of minimizing interference of asynchronous data movement with the application to achieve overall improvements in simulation time. One observation is that the computational resources on staging nodes are often under-utilized and the time interval between I/O dumps are sufficient for extra processing on buffered data. In this paper, we take one step forward and demonstrate the use of staging nodes for a diversity of data operations to achieve not only high write performance, but high read performance and timely monitoring of output data and simulation.

Active Storage. Active Storage [39] deploys data processing operations directly on the storage nodes where the data are buffered to reduce the amount of data movement between storage and compute nodes. The storage nodes have limited computation and memory resources which are shared among applications, so one potential problem with Active Storage is how to manage such resources to meet deadlines for multiple applications and minimize performance downgrade of storage nodes. Abacus [5] demonstrates the benefit of flexible, dynamic function placement in Active Storage, and we are planning to investigate similar mechanism for Staging Area.

In-situ Data Analytics and Visualization. Hercules [48] applies an end-to-end approach to tightly couple together all simulation components, including meshing, partitioning, solver, and visualization, and runs all components on the same supercomputer. It eliminates intermediate I/O and data movement between simulation components to address the I/O bottleneck, but requires scaling data analysis and visualization to the level where simulation runs and all simulation components needed to be changed to efficiently share data with each other. PreDatA couples the staging area with the application more loosely and through the ADIOS interface, which requires minimal changes to application code and provides more flexibility of composing the simulation pipeline.

Scientific Workflows. Scientific workflow systems such as Pegasus [13] and Kepler [31] are used to automate scientific data and simulation management. Unlike the end-to-end approach used in In-situ visualization mentioned above, components in the workflow are usually connected through file-based interface, so the performance of workflow is very sensitive to data placement and movement and may easily affected by poor I/O performance [12]. PreDatA can serve as the early stage in the whole workflow pipeline and can applies application-specific data reduction, validation, and filtering operation before data hits disk to reduce data volume to be processed by subsequent steps in workflow.

Scientific Data Stream Processing. Scientific data stream processing, such as filtering [6], sampling [49], query [26], and transformation [22], is related to our work. PreDatA complements such work and can be used either as an in-transit data processing framework for implementing streaming processing tasks, or as a data forwarding layer to directly feed data to existing streaming processing systems.

Code Coupling. Memory-to-memory code coupling addresses some of the issues faced by PreDatA, such as data movement and re-distribution [3], [23]. PreDatA provides the underpinnings for supporting the rich model-model communications needed for inter-application interactions [14].

Interactive Computational Steering. Runtime steering can aid scientists in debugging and monitoring their simulations [19], [47]. The capability of extracting and inspecting data from running simulation with small overhead and interference makes PreDatA a potential infrastructure for online steering of running application.

Data-intensive Computing in the HPC Domain. Recently, there is increasing interest in building high-level abstractions and programming models for data intensive applications in HPC domain. HiMach [46] applies the MapReduce model to analyze molecular dynamics simulation trajectories and shows decent efficiency at tera-byte scale. On the other hand, experiences from implementing materialized ground models [42] show poor performance of MapReduce because some features provided by MapReduce is unnecessary for target application. AllPairs [33] gains similar insights that mismatch between the application workload and the available abstraction can result in poor performance. Currently, PreDatA provides a two-pass streaming processing model which has shown to be sufficient for the applications we have experienced. Investigating new abstraction and programming model will be on agenda when we find new requirements from applications.

By comparing and associating PreDatA with related research efforts, it is clear that PreDatA enhances the flexibility and scalability of current I/O stack on HEC platform and can be used to support a wide range of data analytics to facilitate the storage, monitoring, analysis, and understanding of massive scientific data.

VIII. Conclusions and Future Work

This paper presents the PreDatA middleware for preparing and characterizing data 'in-transit', that is, while data is being produced by the large scale simulations running on peta-scale machines. PreDatA offloads output data from a running simulation with low-overhead using asynchronous data extraction. It also exploits the computational power of dedicated staging nodes to perform select data manipulations. PreDatA enhances the scalability and flexibility of current I/O stacks on HEC platforms and is useful for data pre-processing, runtime data analysis, and inspection. The DataSpaces services now being integrated into PreDatA also demonstrate its potential utility for rich model-model interactions in large-scale HPC codes. Performance evaluations with several production scientific applications on ORNL's Peta-scale machines show the feasibility of the PreDatA approach and show the performance advantages derived from using the PreDatA I/O stack compared to existing synchronous approaches.

We observe that the following key features distinguish the class of preparatory data analysis we address here:

Easy data movement. Efficient PreDatA and DataSpaces operations rely on the ability of the infrastructure to transfer data to the preparatory processes without causing measurable overhead to the application.

Global data knowledge. The availability of global knowledge is essential in the pre-processing of data for analysis and for application interaction.

Flexible partitioning of pre-analytics pipeline. The performance requirements for a pre-analytics pipeline require the ability to flexibly partition complex data processing operations.

Streaming computation. The large size of data being processed and the limitation on available memory space within the processing area can limit the scope of viable operations in the processing pipeline. A streaming computational model circumvents these limitations by providing a window on the data in which more expansive pipelines can be utilized.

Standard programming model. Scientific developers are familiar with using standard APIs such as MPI for the development of analytical programs. An architecture that seeks to address the needs of the scientific user must be able to utilize standard parallel programs for the pre-analytic data processing pipeline.

Our future work leverages these insights in several ways. First, we plan to define a higher level programming model and abstractions to support a broader set of applications and pre-data analytics, including online data diagnostics and code coupling. Second, we are going to investigate mechanisms for dynamically adapting system configuration and operation placement to cope with changing resource availability or performance characteristics. Third, we will develop performance models for sizing staging areas and provisioning their services.

Acknowledgment

This research used resources of the National Center for Computational Sciences at Oak Ridge National Laboratory, which is supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC05- 00OR22725. This research is based in part upon work supported by the National Science Foundation through the High- End Computing University Research Activity (HECURA) Grant Number 0621538.

References

[1] Hasan Abbasi, Jay Lofstead, Fang Zheng, Scott Klasky, Karsten Schwan, and Matthew Wolf. Extending i/o through high performance data services. In *CLUSTER*, 2009.

[2] Hasan Abbasi, Matthew Wolf, Greg Eisenhauer, Scott Klasky, Karsten Schwan, and Fang Zheng. Datastager: scalable data staging services for petascale applications. In *HPDC*, 2009.

[3] Hasan Abbasi, Matthew Wolf, Karsten Schwan, Greg Eisenhauer, and A. Hilton. Xchange: coupling parallel applications in a dynamic environment. In *CLUSTER*, 2004.

[4] Nawab Ali, Philip Carns, Kamil Iskra, Dries Kimpe, Samuel Lang, Robert Latham, Robert Ross, Lee Ward, and P. Sadayappan. Scalable i/o forwarding framework for high-performance computing systems. In *CLUSTER*, 2009.

[5] Khalil Amiri, David Petrou, Gregory R. Ganger, and Garth A. Gibson. Dynamic function placement for data-intensive cluster computing. In *USENIX Annual Technical Conference*, 2000.

[6] Michael D. Beynon, Renato Ferreira, Tahsin M. Kurç, Alan Sussman, and Joel H. Saltz. Datacutter: Middleware for filtering very large scientific datasets on archival storage systems. In *MSST*, 2000.

[7] Ron Brightwell, Trammell Hudson, Kevin T. Pedretti, Rolf Riesen, and Keith D. Underwood. Implementation and performance of portals 3.3 on the cray xt3. In *CLUSTER*, 2005.

[8] Philip Carns, Sam Lang, Robert Ross, Murali Vilayannur, Julian Kunkel, and Thomas Ludwig. Small-file access in parallel file systems. In *IPDPS*, 2009.

[9] SDM Center. Scidac scientific data management center. https://sdm.lbl.gov/sdmcenter/, September 2009.

[10] L. Chacón. A non-staggered, conservative, $\nabla \dot{s} B \rightarrow= 0$, finite-volume scheme for 3D implicit extended magnetohydrodynamics in curvilinear geometries. *Computer Physics Communications*, 163:143–171, November 2004.

[11] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. In *OSDI*, 2004.

[12] Ewa Deelman and Ann Chervenak. Data management challenges of data-intensive scientific workflows. In *CCGRID*, 2008.

[13] Ewa Deelman, Gurmeet Singh, Mei hui Su, James Blythe, A Gil, Carl Kesselman, Gaurang Mehta, Karan Vahi, G. Bruce Berriman, John Good, Anastasia Laity, Joseph C. Jacob, and Daniel S. Katz. Pegasus: a framework for mapping complex scientific workflows onto distributed systems. *Scientific Programming Journal*, 13:219–237, 2005.

[14] Ciprian Docan, Manish Parashar, Julian Cummings, Norbert Podhorszki, and Scott Klasky. Experiments with Memory-to-Memory Coupling for End-to-End Fusion Simulation Workflows. Technical Report TR-104, Center for Autonomic Computing (CAC), Rutgers University, July 2009.

[15] Ciprian Docan, Manish Parashar, and Scott Klasky. Dart: a substrate for high speed asynchronous data io. In *HPDC*, 2008.

[16] Greg Eisenhauer. Evpath: event transport middleware layer. http://www.cc.gatech.edu/systems/projects/EVPath/, September 2009.

[17] Greg Eisenhauer, Fabián E. Bustamante, and Karsten Schwan. Native data representation: An efficient wire format for high-performance distributed computing. *IEEE Trans. Parallel Distrib. Syst.*, 13(12):1234–1246, 2002.

[18] Jim Gray, David T. Liu, Maria Nieto-Santisteban, Alex Szalay, David J. DeWitt, and Gerd Heber. Scientific data management in the coming decade. *SIGMOD Rec.*, 34(4):34–41, 2005.

[19] Weiming Gu, Greg Eisenhauer, Karsten Schwan, and Jeffrey S. Vetter. Falcon: On-line monitoring for steering parallel programs. *Concurrency - Practice and Experience*, 10(9):699–736, 1998.

[20] Chad Jones, Kwan-Liu Ma, Allen Sanderson, and Lee Roy Myers Jr. Visual interrogation of gyrokinetic particle simulations. *J. Phys.: Conf. Ser.*, 78(012033):6, 2007.

[21] Wei keng Liao and Alok N. Choudhary. Dynamically adapting file domain partitioning methods for collective i/o based on underlying parallel file system locking protocols. In *SC*, 2008.

[22] S. Klasky, S. Ethier, Z. Lin, K. Martins, D. McCune, and R. Samtaney. Grid -based parallel data streaming implemented for the gyrokinetic toroidal code. In *SC*. IEEE Computer Society, 2003.

[23] Jae-Yong Lee and Alan Sussman. High performance communication between parallel programs. In *IPDPS*, 2005.

[24] Jonghyun Lee, Robert B. Ross, S. Atchley, M. Beck, and Rajeev Thakur. Mpi-io/l: efficient remote i/o for mpi-io via logistical networking. In *IPDPS*, 2006.

[25] Jonghyun Lee, Robert B. Ross, Rajeev Thakur, Xiaosong Ma, and Marianne Winslett. Rfs: efficient and flexible remote file access for mpi-io. In *CLUSTER*, 2004.

[26] Ying Liu, Nithya Vijayakumar, and Beth Plale. Stream processing in data-driven computational science. In *GRID*, 2006.

[27] Jay Lofstead, Scott Klasky, Karsten Schwan, Norbert Podhorszki, and Chen Jin. Flexible io and integration for scientific codes through the adaptable io system (adios). In *CLADE at HPDC*, 2008.

[28] Jay Lofstead, Qing Liu, Scott Klasky, Michael Booth, Ron Oldfield, Karsten Schwan, and Matthew Wolf. High performance io on busy systems. In *PDSW at SC*, 2009.

[29] Jay Lofstead, Fang Zheng, Scott Klasky, and Karsten Schwan. Input/output apis and data organization for high performance scientific computing. In *PDSW at SC*, 2008.

[30] Jay Lofstead, Fang Zheng, Scott Klasky, and Karsten Schwan. Adaptable, metadata rich io methods for portable high performance io. In *In IPDPS*, 2009.

[31] Bertram Ludäscher, Ilkay Altintas, Chad Berkley, Dan Higgins, Efrat Jaeger, Matthew Jones, Edward A. Lee, Jing Tao, and Yang Zhao. Scientific workflow management and the kepler system. *Concurrency and Computation: Practice and Experience*, 18(10):1039–1065, 2006.

[32] Xiaosong Ma, Jonghyun Lee, and Marianne Winslett. High-level buffering for hiding periodic output cost in scientific simulations. *IEEE Trans. Parallel Distrib. Syst.*, 17(3):193–204, 2006.

[33] Christopher Moretti, Jared Bulosan, Douglas Thain, and Patrick J. Flynn. All-pairs: An abstraction for data-intensive cloud computing. In *IPDPS*, 2008.

[34] Arifa Nisar, Wei keng Liao, and Alok N. Choudhary. Scaling parallel i/o performance through i/o delegate and caching system. In *SC*, 2008.

[35] Ron Oldfield, Lee Ward, Rolf Riesen, Arthur B. Maccabe, Patrick Widener, and Todd Kordenbrock. Lightweight i/o for scientific applications. In *CLUSTER*, 2006.

[36] Stan Park and Kai Shen. A performance evaluation of scientific i/o workloads on flash-based ssds. In *IASDS at CLUSTER*, 2009.

[37] PDSI. Scidac petascale data storage institute. http://www.pdsi-scidac.org/, September 2009.

[38] PGPLOT. Pgplot graphics subroutine library. http://www.astro.caltech.edu/~tjp/pgplot/, September 2009.

[39] Juan Piernas, Jarek Nieplocha, and Evan J. Felix. Evaluation of active storage strategies for the lustre parallel file system. In *SC*, 2007.

[40] Milo Polte, Jiri Simsa, Wittawat Tantisiriroj, and Garth Gibson. Fast log-based concurrent writing of checkpoints. In *PDSW at SC*, 2008.

[41] Oliver Rübel, Prabhat, Kesheng Wu, Hank Childs, Jeremy Meredith, Cameron G. R. Geddes, Estelle Cormier-Michel, Sean Ahern, Gunther H. Weber, Peter Messmer, Hans Hagen, Bernd Hamann, and E. Wes Bethel. High performance multivariate visual data exploration for extremely large data. In *SC*, 2008.

[42] Steven W. Schlosser, Michael P. Ryan, Ricardo Taborda-Rios, Julio López, David R. O'Hallaron, and Jacobo Bielak. Materialized community ground models for large-scale earthquake simulation. In *SC*, 2008.

[43] K. E. Seamons, Y. Chen, P. Jones, J. Jozwiak, and M. Winslett. Server-directed collective i/o in panda. In *SC*, 1995.

[44] Rishi Rakesh Sinha and Marianne Winslett. Multi-resolution bitmap indexes for scientific data. *ACM Trans. Database Syst.*, 32(3):16, 2007.

[45] Kurt Stockinger, John Shalf, E. Wes Bethel, and Kesheng Wu. Dex: Increasing the capability of scientific data analysis pipelines by using efficient bitmap indices to accelerate scientific visualization. In *SSDBM*, 2005.

[46] 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 *SC*, 2008.

[47] Tiankai Tu, Hongfeng Yu, Jacobo Bielak, Omar Ghattas, Julio C. López, Kwan-Liu Ma, David R. O'Hallaron, Leonardo Ramirez-Guzman, Nathan Stone, Ricardo Taborda-Rios, and John Urbanic. Analytics challenge - remote runtime steering of integrated terascale simulation and visualization. In *SC*, 2006.

[48] Tiankai Tu, Hongfeng Yu, Leonardo Ramirez-Guzman, Jacobo Bielak, Omar Ghattas, Kwan-Liu Ma, and David R. O'Hallaron. Scalable systems software - from mesh generation to scientific visualization: an end-to-end approach to parallel supercomputing. In *SC*, 2006.

[49] Huai Wang, Srinivasan Parthasarathy, Amol Ghoting, Shirish Tatikonda, Gregory Buehrer, Tahsin M. Kurç, and Joel H. Saltz. Design of a next generation sampling service for large scale data analysis applications. In *ICS*, 2005.

[50] Brent Welch, Marc Unangst, Zainul Abbasi, Garth A. Gibson, Brian Mueller, Jason Small, Jim Zelenka, and Bin Zhou. Scalable performance of the panasas parallel file system. In *FAST*, 2008.

[51] Hongfeng Yu and Kwan-Liu Ma. A study of i/o methods for parallel visualization of large-scale data. *Parallel Comput.*, 31(2):167–183, 2005.

[52] Weikuan Yu, Jeffrey S. Vetter, and Sarp Oral. Performance characterization and optimization of parallel i/o on the cray xt. In *IPDPS*, 2008.

[53] Li Zhang and Manish Parashar. Seine: a dynamic geometry-based shared-space interaction framework for parallel scientific applications. *Concurrency and Computation: Practice and Experience*, 18(15):1951–1973, 2006.

[54] Xuechen Zhang, Song Jiang, and Kei Davis. Making resonance a common case: A high-performance implementation of collective i/o on parallel file systems. In *IPDPS*, 2009.

Appendix

The programming interface of PreDatA is listed in Table I.

TABLE I
PREDATA PROGRAMMING INTERFACE

Functions Implemented by User
typedef int (* SMORES_route_func) (uint64_t); // called by compute node to determine destination staging node.
typedef int (* SMORES_partial_calculate_func) (void *); // called by compute node to generate first-pass partial results.
typedef int (* SMORES_aggregate_func) (void *); // called by staging node to generate aggregate partial results.
typedef int (* SMORES_initialize_func) (void *); // called by staging node to initialize streaming processing.
typedef int (* SMORES_map_func) (void *, uint64_t); // called by staging node to perform Map on one packed partial data chunk.
typedef int (* SMORES_reduce_func) (void *, uint64_t, void *); // called by staging node to perform Reduce on a key and associated values.
typedef int (* SMORES_partition_func) (void *, uint64_t); // called by staging node to determine the staging node rank for the specified key.
typedef void (* SMORES_combine_func) (void *, uint64_t, void *); // called by staging node to perform Combine to aggregate local key values.
typedef int (* SMORES_compare_key_func) (void *, uint64_t, void *, uint64_t); // called by staging node to compare two keys.
typedef int (* SMORES_finalize_func) (void *); // called by staging node to finalize streaming processing.

Functions Provided by PreDatA Runtime
int SMORES_initialize(); // called by staging node to initialize PreDatA runtime.
int SMORES_finalize(); // called by staging node to exit PreDatA finalize.
SMORES_operation SMORES_create_operation(void*); // called by staging node to create a data operation.
int SMORES_delete_operation(SMORES_operation); // called by staging node to destroy a data operation.
int SMORES_add_initialize(SMORES_operation, SMORES_initialize_func); // called by staging node to register Initialize() function to a data operation.
int SMORES_add_map(SMORES_operation, SMORES_map_func); // called by staging node to register Map() function to a data operation.
int SMORES_add_reduce(SMORES_operation, SMORES_reduce_func); // called by staging node to register Reduce() function to a data operation.
int SMORES_add_partition(SMORES_operation, SMORES_combine_func); // called by staging node to register Partition() function to a data operation.
int SMORES_add_combine(SMORES_operation, SMORES_combine_func); // called by staging node to register Combine() function to a data operation.
int SMORES_add_finalize(SMORES_operation, SMORES_finalize_func); // called by staging node to register Finalize() function to a data operation.
int SMORES_add_compare_key(SMORES_operation, SMORES_compare_key_func); // called by staging node to register Compare_key() function to a data operation.
int SMORES_do_initialize(SMORES_operation, void *); // called by staging node to execution operation's Initialize() (blocking call).
int SMORES_do_map(SMORES_operation, void *, uint64_t); // called by staging node to execution operation's Map() (non-blocking call).
int SMORES_do_reduce(SMORES_operation); // called by staging node to execution operation's Reduce() (non-blocking call).
int SMORES_do_combine(SMORES_operation); // called by staging node to execution operation's Combine() (blocking call).
int SMORES_do_shuffle(SMORES_operation); // called by staging node to shuffle data and Group data by key (blocking call).
int SMORES_do_finalize(SMORES_operation, void *); // called by staging node to execution operation's Finalize() (blocking call).
int SMORES_barreir(SMORES_operation); // called by staging node to wait for Map or Reduce to finish. (blocking call).
int SMORES_extract_key_values(SMORES_operation, void *); // called by staging node to copy operation's key value buffer.
int SMORES_merge_key_values(SMORES_operation, void *); // called by staging node to add key-values to operation's key value buffer.
int SMORES_delete_key_values(SMORES_operation); // called by staging node to free operation's key-values buffer.
int SMORES_emit_intermediate(void *, uint64_t, void *, uint64_t); // called by staging node (in Map or Combine) to register intermediate key value pair.
int SMORES_emit(void *, uint64_t, void *, uint64_t); // called by staging node (in Reduce) to register key-value pair.
uint64_t SMORES_count_chunks(int64_t); // called by staging node to get number of chunks in the incoming stream.
int SMORES_get_next_chunk(int64_t, void *, uint64_t *); // called by staging node to get a chunk in the incoming stream.
int SMORES_attach(void *, uint64_t); // called by compute node (in Partial_Calculate function) to attach first-pass partial result.