FlexIO: Location-flexible Execution of In Situ Data Analytics for Large Scale Scientific Applications

Fang Zheng, Hongbo Zou, Greg Eisenhauer, Karsten Schwan, Matthew Wolf, Jai Dayal, Tuan-Anh Nguyen, Jianting Cao, Hasan Abbasi*, Scott Klasky*, Norbert Podhorszki*, Hongfeng Yu[†]
College of Computing, Georgia Institute of Technology, Atlanta, GA, USA
{fzheng, hzou7, eisen, schwan, mwolf, jdayal3, tuananh, jcao32}@cc.gatech.edu
*National Center for Computational Sciences, Oak Ridge National Laboratory, Oak Ridge, TN, USA
{habbasi, sklasky, pnorbert}@ornl.gov
[†]Sandia National Laboratory, Livermore, CA 94550, USA
hyu@sandia.gov

***Abstract*—Increasingly severe I/O bottlenecks on High-End Computing machines are prompting scientists to process simulation output data while simulations are running and before placing data on disk – ”in situ” and/or ”in-transit”. There are several options in placing in-situ data analytics along the I/O path: on compute nodes, on staging nodes dedicated to analytics, or after data is stored on persistent storage. Different placements have different impact on end to end performance and cost. The consequence is a need for flexibility in the location of in situ data analytics. The FlexIO facility described in this paper supports flexible placement of in situ analytics, by offering simple abstractions and methods that help developers exploit the opportunities and trade-offs in performing analytics at different levels of the I/O hierarchy. Experimental results with several large-scale scientific applications demonstrate the importance of flexibility in analytics placement.**

Keywords-I/O, In Situ Processing, Staging, Placement, Data Analytics

I. Introduction

Peta-scale scientific simulations in domains such as Fusion [21], Astrophysics[29], and Combustion[18] now routinely generate terabytes of data in a single run, and such data volumes are only expected to increase. Such massive data is key to the scientific processes being undertaken, so the ability to rapidly store, move, analyze, and visualize it is critical for end user productivity. Yet there are already serious I/O bottlenecks on current High-End Computing (HEC) machines, evident from the significant imbalance between the I/O and computational abilities of HEC engines like Jaguar XT5, Intrepid BG/P -- up to six orders of magnitude[26], and the movement toward Exascale computing platforms is expected to further accelerate this trend. Causes include inherent limits in interconnect bandwidth (e.g., due to caps on energy consumption), contention on shared resources [28], complex patterns of parallel I/O [26], and the scalability limitations and costs of highly parallel file systems.

An undesirable outcome is for scientists to face situations in which either a substantial portion of their simulation runtime is spent in writing data to the storage system [31] or where they must forgo writing out scientifically relevant data in order to keep total I/O within reasonable bounds.

Online or ‘streaming’ data analytics has emerged as an effective way to overcome the increasingly severe I/O bottleneck for scientific applications running at the petascale and beyond. By processing data as it moves through the I/O hierarchy, streaming analytics can reduce data movement costs (both in time and in power), extract and deliver valuable insights from live simulation output in a timely manner, better prepare data for subsequent deep analysis and visualization, and gain improved end-to-end performance and reduced total data movement cost compared to solely offline approaches. The utility of the approach is demonstrated by its wide use on today’s petascale machines, including by leading scientific application teams like the S3D combustion modeling team [44], the GTC[21], GTS[43], and XGC fusion modeling teams [1], CTH [24], and FLASH [12]. Furthermore and enabled by standard parallel I/O interfaces like ADIOS [27], HDF5 [13], and MPI-IO [10], there are emerging infrastructures that support online data analytics, including PreDatA/SmartTap [46][1], Nessie [25], FastBit [41], FFS and EvPath[8][9], GLEAN [39][40], and others [11][37].

For real-time processing of the outputs generated by petascale machines, a key factor distinguishing different data processing techniques is ”where” analytics are placed along the I/O path: on compute nodes, on separate nodes dedicated to analytics (termed ’staging nodes’), or offline (after data is placed into persistent storage). Such placements determine which resources are allocated to particular analytics computations and how/when/which data is moved. In fact, when carefully placing computation along the I/O path, there are significant positive impact both on the performance (i.e., running time) and the cost (i.e., CPU hours) of the resulting coupled simulation and analysis codes [3]. This has been validated with experimental results and explained with analytical performance models [45] that shown that the performance and cost impact of different placement strategies depend on the particular analytics codes, data volumes, and scale of operation. The interesting consequent insight is that *no single, specific placement will be ‘best’ for all applications and analytics pipelines.* This means that *flexibility in placement is a key requirement for efficient ’in situ’ data analytics.*

This paper describes the FlexIO middleware for coupling parallel simulations with in-situ analytics, in ways that offer placement flexibility to those online analytics and visualization codes. FlexIO offers the following functionality: (1) flexibility in where analytics codes are placed -- on compute nodes (either inline or on dedicated ’helper’cores), on staging nodes, on both, or offline; (2) the ability to alter such placements without requiring application codes to be changed or updated; (3) online performance monitoring of computation and data movement; and (4) additional support for data movement reduction through “data conditioning (DC)plug-ins” created and deployed at runtime.

FlexIO supports on-compute node analytics with efficient, lock-free, in-memory queuing and buffering structures. Such on-node analytics seamlessly couple with the RDMA-based inter-node transport for ‘staging node’-based analytics, thus making it easy to place analytics on either set of nodes. Further, since FlexIO adopts the ADIOS I/O API as a coupling interface, selection of the underlying transport used for data movement transport can be hidden from application codes, making changes in their placement transparent to simulation and analytics core algorithms and their implementations. Efficient methods for online monitoring assess the effects of placing analytics on certain nodes, and placements are guided by a semi-automated placement planning procedure that determines the resource allocation of simulation and analytics offline and binds computation to physical resources at job launch time. In addition and also based on online monitoring, DC Plug-ins can be deployed and/or dynamically migrated anywhere in the I/O pipeline. These codelets, written in a subset of the C programming language and created on demand via dynamic binary code generation, provide a simple and efficient way to move app-specific computations across process address spaces and machines to where they can most reduce the volumes of data being moved and/or improve other metrics of interest to end users, such as the end-to-end latency between when a simulation generates output data vs. when its online analytics have been completed, which we term “TimeToData” (TTD).

FlexIO and its abstractions are fully implemented, which makes it possible to evaluate them with realistic scientific codes and their in situ analytics. Experiments conducted on Cray XT5 and an Infiniband cluster show that leveraging the placement flexibility enabled by FlexIO to choose the best placement can improve total execution time by up to 33% compared to inline-only solutions. They also show that the `best’ placements vary across different applications, thereby validating the argument for flexible placement support for in situ analytics.

The remainder of the paper is organized as follows. Section 2 presents background information and motivates the need for flexible placement of in situ data analytics on high end machines. Section 3 describes the design and implementation of the FlexIO middleware. Section 4 describes how FlexIO can be used to exploit flexible placement of analytics through semi-automated offline planning and the automated runtime placement of data conditioning plug-ins. Section 5 presents the performance improvement of two large-scale scientific applications due to flexible placement of analytics. Section 6 reviews related work, and Section 7 concludes the paper.

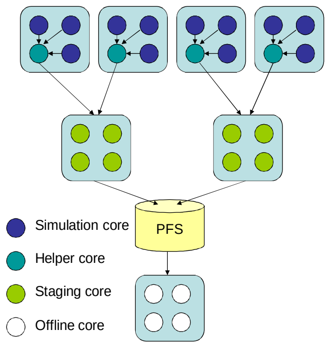

Figure 1. Different In-Situ I/O Placement Options.

II. Background and Motivation

A. Need for Flexible Placement of In Situ Analytics

The placement of in situ data analytics (i.e., “where” analytics computations are placed along the I/O path) determines which resources are allocated to which computations and how/when/which data is moved. Consequently, placement decisions significantly impact both the performance (e.g., running time) and the cost (e.g., CPU hours) of the resulting coupled simulation and analysis codes. However, cost/performance tradeoffs among different placement strategies are not universal, and they vary across different analytics codes, data volumes, and scales, as demonstrated with two representative analytics scenarios described next.

In one application scenario, scientists wish to first slice a portion of the raw output data and then apply a principal component analysis to extract certain features (such as the examples described in [11]). Slicing and feature extraction operations are lightweight and scalable, and most importantly, they can significantly reduce the I/O data volumes compared to offline analyses that first place data on disk. As shown in Figure 1, there are multiple placement choices for these analytics: (i) running them on compute node does not cause measurable slowdown of the simulation and greatly reduces the resources required for moving and buffering the data downstream; (ii) placement on staging nodes requires sufficient memory space to hold all of the raw output data; it is also wasteful, as most of the data moved into staging nodes will be filtered out; and at large scale, such bulk data movement may cause serious contention on the interconnect and slow down the simulation, even if it is done asynchronously [3].

Another application scenario is one in which scientists need to generate and store data at multiple levels of resolution, and then index the data so that they can later carry out exploratory analysis and visualization [38]. Both operations effectively expand the output data by multiple times. If placed on compute nodes (assuming there is sufficient memory unused by the simulation), this may cause contention on their internal memory hierarchies. In addition, since the final output (which may be much larger than the raw output data) must be written to storage, such write operations performed by some very large number of compute nodes will likely incur significant I/O overheads with consequent blocking causing wasted CPU cycles on compute nodes. Their alternative execution on staging nodes is well-known to reduce compute node-level resource consumption as well as disk I/O overheads [28].

From the two examples above, it is clear that best vs. worst placements will likely cause substantially different levels of performance and must involve careful assessments of analytics costs and data movement/volumes. They also show that the `data reduction/expansion ratio' of analytics plays an important role in deciding placement. Additional factors impacting the simulation's as well as the analytic pipeline's performance are the resource availability in compute and staging nodes, interconnect-dependent data movement costs/overheads, and analytics scalability [45], where the latter is strongly affected by the use of collective operations, which when used by analytics running on compute nodes, will have considerably higher cost than when run on the relatively smaller number of staging nodes.

B. *Placement Support in Existing Systems*

A number of different in situ analytics placements and techniques have been documented in the literature. Adopting a derivation of the classification proposed in [44], we briefly review these related research efforts.

Inline Processing: analysis/visualization routines are synchronously performed by the simulation. ParaView's co-processing library [11], VisIt's remote visualization [38] and other in situ visualization work [37] [44] fall into this category.

Helper Cores: some cores on the compute nodes used by the simulation are dedicated to perform select analysis actions. Examples include Functional Partitioning [23] and Software Accelerator [35].

Staging Area Processing: on its way from computation to storage, data is routed via an additional set of compute nodes on which it can be temporarily buffered, analyzed, and visualized. DataStager [3], PreDatA [46], Nessie [25], FFS and EvPath [8][9], GLEAN [39][40] and HDF5/DSM [13] follow this approach.

Active Storage: certain computational routines may be deployed directly on I/O or storage nodes and triggered to operate whenever data is written and/or read [31].

Offline Processing: data written to storage is read back for additional or long term analysis or visualization [14], typically assisted by workflow tools [22].

A common shortcoming of existing systems is that they each support certain, fixed placement choices. Some offer limited flexibility in running analytics at different locations [3][45], but require adopting particular coding patterns or substantial re-coding efforts. Others are based on a specific model of how analytics codes must access and represent data. In comparison to such work, the FlexIO approach described in this paper not only provides flexibility in placement, but also makes no assumptions about how analytics or visualization codes are written (as long as they use the widely adopted ADIOS I/O interface). It is not limited by the aggregate amount of memory present in the staging area, provides explicit support for placement planning, and permits the dynamic creation and deployment of data selection and filtering codes – DCplugins -- that can further enhance the performance of the streaming I/O processing carried out in analytics pipelines.

C. *ADIOS I/O Framework*

FlexIO is implemented as an extension of ADIOS (Adaptable Input/Output System), which is a high level parallel I/O library providing a meta-data rich read/write interfaces to simulation and analysis codes. ADIOS has a set of built-in I/O methods underlying the higher level API to support various file I/O (including HDF5, NetCDF, POSIX, and MPI-IO) and data staging methods. Switching between different methods can be configured through an external XML configuration file without modification to application codes. ADIOS has been used by several leadership scientific codes (like GTC, GTS, XGC, S3D, Pixie3D, and Chimera), and integrated with several popular analysis and visualization tools such as ParaView, VisIt and Matlab. For details about ADIOS API, we refer readers to its manual[16].

For analytics components using the ADIOS API, FlexIO couple simulation and analytics codes so as to easily vary the analytics placement. The goal of such flexibility is to obtain the performance metrics desired by end users, including high performance for simulation codes, reduced Time-To-Data (TTD) for simulation/analytics pipelines, and acceptable costs and overheads in machine use.

III. FLEXIO DESIGN AND IMPLEMENTATION

A. *Overview*

Summarizing briefly, FlexIO has the following functionality: (1) flexibility in where analytics codes are placed; (2) the ability to alter such placements without requiring application codes to be changed or updated; (3) efficient data movement between simulation and analytics; (4) online performance monitoring of computation and data movement; and (5) additional support for data movement reduction through "Data Conditioning Plug-ins" created and deployed at runtime.

The FlexIO software stack is shown in Figure 2. Simulation and analytics codes use the ADIOS read/write API for data exchange, but in addition, FlexIO also extends this API to allow chunk-based data exchanges for incremental data processing by analytics codes (Ongoing work in the ADIOS team is creating a standard API extension for chunk-based data movement and processing). The FlexIO runtime handles buffer management, parallel data re-distribution, and performance monitoring. It also manages the dynamic creation and placement of "DCPlug-ins", which are mobile codelets compiled, deployed, and executed at runtime for on-the-fly data manipulation. Online performance monitoring provides information for scheduling data movement and dynamic DC

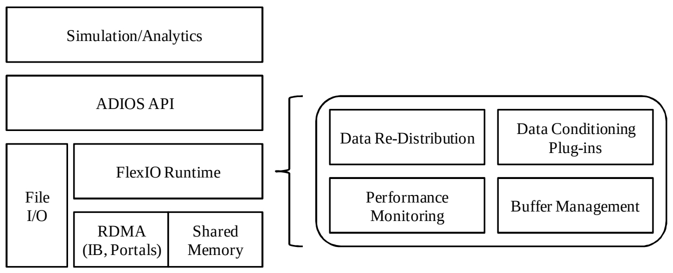

Figure 2. FlexIO Software Stack

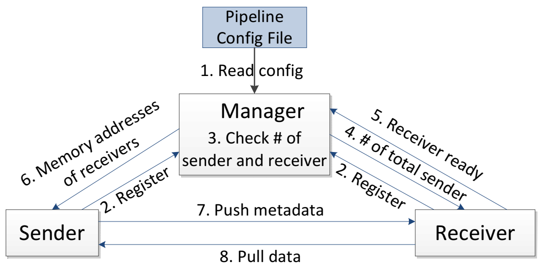

Figure 3. Parallel Data Re-distribution.

Plug-in placement. At the lowest transport level, FlexIO uses efficient RDMA and shared memory data movement for inter- and intra- node setups, respectively. The choice of low level transport is automatically configured according to the placement of in situ analytics.

FlexIO inherits from ADIOS useful features like I/O componentization and file I/O methods (to enable offline placement), and it leverages our own previous work on efficient RDMA-based data movement on HEC machines, enhanced with and dynamic code generation for on-the-fly data manipulation[3]. The newly added features are the ADIOS chunk read interface, parallel multi-dimensional array re-distribution, efficient on-node data movement via shared memory, enriched performance monitoring, and automatic, adaptive placement of DC Plug-ins. The resulting system supports diverse placement options and efficient data movement between simulation and analytics.

B. *Programming Interface*

By using the ADIOS read/write interface, FlexIO can be applied to any application and any analytics or visualization code using the ADIOS API. All such codes simply continue to read and write data as they may previously have done for their disk-directed I/O. FlexIO, then, controls how/when/where data is actually moved. Potential use cases operate as follows. (1) The simulation passes data to analytics by writing/appending to a named “file”, one timestep of output data at a time. (2) The analytics polls for the next available timestep of output data on the “file”, and issues adios_read_var() calls to load data into its read buffers. (3) Including for global multi-dimensional arrays, the analytics only needs to specify the geometric ranges of arrays to read as parameters to adios_read_var() calls, whereupon FlexIO automatically moves and re-organizes needed data into the appropriate destination read buffers. (4) DCPlug-ins can be used to customize data before it is moved, reduce it, or select from a single write-side buffer only those data offsets that are needed by each reader [3].

(5) FlexIO adds to the ADIOS interface a chunk-based read API. This is intended for analytics that can incrementally consume the output data generated by the simulation within a single I/O timestep. The granularity of one single chunk is the entire set of output variables emitted by a single simulation process in one I/O action. Such “batched” movement and processing of a bundle of related variables is easily supported by the underlying data movement layer, and our experiences show that this relatively coarse-grained abstraction does not improve undue restrictions on many analytics implementations. DCPlug-ins can customize data within chunks.

(6) Underlying transports can be switched without changing application code, thereby making it easy to alter placement without affecting simulation or analytics codes. For example, to place analytics in compute nodes where simulation is also running, the user can choose the underlying transport method to be the `shared memory transport’ (which we will describe in more detail next), by specifying the transport choice in the ADIOS’s XML configuration file. To alternatively run the same analytics code on a staging node, a one-line change to the configuration file simply indicates that the RDMA staging method should be used.

An additional advantage of using FlexIO’s uniform I/O interface is that it simplifies development of in situ analytics. Users can use file I/O during coding and debugging, and then switch to the staging or shared memory transport when bringing analytics online for production runs, without further code modification.

C. *Parallel Data Redistribution*

There is considerable complexity in presenting to users a convenient API yet also providing the placement flexibility needed for high performance. Key to this complexity is that the FlexIO runtime must translate the high-level ADIOS API calls into actual data movements between simulation and analytics processes using low-level RDMA or shared memory transports, as shown in Figure 3. The following functionality ensures efficient data movement and processing.

(1) Connection establishment: before actual data movement, simulation and analytics processes establish connections to each other, with assistance of an external manager. To avoid overloading this manager, simulation and analytics processes, respectively, elect a local manager. That manager aggregates messages among peer processes and interacts with the external manager. Finally, the latter is involved only in connection establishment and is, therefore, not in the critical path of actual data movements.

When moving global array data between two parallel programs, data must be distributed according to the data distributions used at both ends. The ADIOS write and read API captures the array distribution among simulation and analytics processes, respectively. Based on this information, FlexIO runtime generates the re-distribution mapping, as explored in previous work as the MxN data re-distribution problem [30]. In order to maintain full compatibility with the ADIOS API, FlexIO’s MxN data re-distribution is done through receiver-side shuffling. Each sender process sends a meta-data request describing its data distribution to a receiver process according to some fixed sequential mapping. The receiver processes with their knowledge about data distributions on both sides then fetch data via RDMA or the shared memory transport and then shuffle data among themselves using a ring network. This alleviates the work needed at on the sending (e.g., .simulation) side at the expense of additional data movement and buffer usage on the receiving (e.g., analytics) side. Well-aligned data distributions at both sides obviate the shuffle step and in many cases, DC Plug-ins can be used to appropriately move data to avoid receiver-side shuffling [30], but the strategy is suboptimal when the distributions at two sides are ill-matched and the amount of data to be shuffled is large. We leave the optimization of MxN data re-distribution algorithm as part of future work.

D. Shared Memory Transport

The FlexIO shared memory transport is a one-way data movement facility for two parallel programs placed on the same node. It can be seamlessly switched with the RDMA-based data staging transport for data movement between different nodes across the interconnect.

With the shared memory transport, each simulation writer process has a local data queue. The data queue is implemented as a circular, lock-free FIFO queue inspired by Fastforward [15]. Each entry in the queue corresponds to the writer's chunk of the new time step of output data. The writer and reader have separate pointers to the next entry to put or get, and these are guaranteed to be placed into different cache lines to reduce cache coherency traffic. Each entry for a time step has a status flag with four possible states: writing, full, reading, empty.

A reader (i.e., analytics process) waits on a memory fence on this flag, signaling that a new chunk is available. On the writing side, the writer process must first check that the next entry in the circular queue is marked as having been read before new data is copied into the queue. The flag is then set to 'full', which signals the reader. The use of this shared memory transport, therefore, implicitly enforces synchronization between writer and reader processes at the granularity of output time steps, which is appropriate for the in situ data analytics targeted by our work.

For each timestep, the entry contains the actual data as well as metadata. Metadata includes the ADIOS file name, group name, timestep value, and variable name, data type and dimensions. Such metadata then supports the higher-level ADIOS read/write semantics.

As shown by performance measurements in Section V, supporting the ADIOS high-level abstraction does not incur significant overheads, so that the performance of data movement is bounded only by the hardware's available memory bandwidth. Achieving high bandwidth, however, requires the shared memory transport to carefully align the layout of entries in data queues and pad them to make sure entries do not share cache lines, so as to reduce false sharing. Additionally, this transport is designed to purposely shift as much work to the reader side as possible, as in our common usage scenario, the reader side will be performing analytics, whereas the writer should be minimally disturbed in its execution of core simulation functionality.

E. Data Conditioning Plugins

Data Conditioning Plug-ins are mobile codes embedded in the FlexIO transport. They are triggered to perform operations on data during the exchange of data between simulation and analytics. DC Plug-ins can be executed within the address space of either the simulation or analytics, and they can be migrated across address spaces at runtime. Such runtime code mobility further enhances placement flexibility, offering fine-grain runtime control and data manipulation capabilities.

DC Plug-ins are stateless codelets created on the reader side (e.g., by analytics) side to customize writer side outputs on the fly. Useful examples of DC Plug-ins include data markup, annotation, sampling, bounding box, unit conversion, etc. They are typically lightweight in terms of compute and memory usage, and easily coded with the C subset offered by the C-on-Demand (CoD) used to program them[3].

DC Plug-ins are specified as parameters to ADIOS read calls. Their code strings are compiled and installed in the the appropriate process' address space through the dynamic binary code generation offered by CoD. The code can be executed at either the analytics side or simulation side. Runtime deployment of DC Plug-ins from the analytics side into simulation processes is through a communication channel separate from the ones used for data flow. The DC Plug-in placement decision can be informed by explicit control from the caller, or it can be determined by the FlexIO runtime based on performance monitoring information, e.g., to adapt to load imbalance between simulation and analytics and/or to reduce data movement between them (see Section IV for concrete use cases).

F. Performance Monitoring

FlexIO provides performance monitoring for simulation, in situ analytics, and DC Plug-ins. There are measurement points at all levels of the FlexIO's software stack to gather a variety of information, including the timing of data movement and DC Plug-in execution, as well as the transferred data volumes. Additional information about the computation and communication behavior of simulation and analytics can also be obtained by explicitly instrumenting the codes.

Performance information is used in two ways. When used for offline performance tuning, monitoring information can be dumped to trace files, and the developer can use it to understand and tune analytics codes. When used for runtime management, monitoring data captured from the simulation side can be gathered online at the analytics side. The analytics process(es) can then use it to dynamically schedule data movement and decide the placement of DC Plug-ins.

IV. METRICS-DRIVEN PLACEMENT

Given that analytics placement can significantly impact the performance and cost of coupled simulation and analytics, end users will want placement decisions to be based on **end-to-end** performance and cost goals for the integrated simulation **and** analytics pipeline. This is because it is the scientific insights derived from data analytics that drive science end users and determine their productivity.

A. Performance and Cost Metrics

We introduce the following two end-to-end performance and cost metrics, initially defined in [45], for quantitative comparison between different placements of in situ analytics.

Total Execution Time: the time from the start of simulation and analysis to the completion of both, also termed "Time to Data" in our prior work [57].

Total CPU Hours: the total nodes used multiplied by the total execution time (in units of hours). This metric measures the cost of a run, as supercomputing centers commonly charge users with the number of CPU hours consumed by their jobs.

Other useful metrics include the end-to-end total power consumption, time-for-first-image, and time-to-solution [45]. Compared to those metrics, *Total Execution Time* and *Total CPU Hours* are more end-user oriented and characterize the whole simulation/analysis workflow. We will consider how placement impacts other metrics in future work.

B. Placement Planning

For static scenarios in which the in situ analytics to be performed are known a priori and where there are no significant dynamic variations of system or workload, placement of in situ analytics can be planned before the production run. Placement planning involves provisioning appropriate resources to simulation and analytics, respectively, binding their processes to resources, and choosing data movement transports accordingly. Although the obvious basic principle is to balance the data generation rate of the simulation with the consumption rate of analytics, the optimal placement will depend on the performance/cost objective used, the scalability of simulation and analytics codes, the volumes of data moved both within the simulation and analytics and between them, and the degree of contention on whatever resources are shared by simulation and analytics. This makes placement difficult to fully automate, and it gives rise to an iterative process in which a satisfactory placement is determined by repeated, monitored profiling runs, performance evaluation, and placement adjustment.

FlexIO helps with placement planning indirectly, by making it unnecessary to changes codes when placements are altered and directly, by providing performance monitoring data that can be examined offline, e.g., to help developers understand whether analytics are slower than the simulation's I/O interval, to assess how well computation and data movement are overlapped, etc. Such understanding can be leveraged to adjust resource provisioning and binding.

C. Dynamic Placement of Data Conditioning Plug-ins

While profiling can plausibly be done offline, DC Plug-ins and their runtime deployment make them inherently online constructs. Toward this end, FlexIO (1) provides applications with explicit control over Plug-in creation and deployment, (2) it permits them to specify policies that guide these actions, and (3) there are simple (configurable) default policies. The default policy in current use prefers placements that try to reduce Total Execution Time. When a DCPlug-in is created at the analytics side, the FlexIO runtime first chooses to move raw simulation output data and run the DC Plug-in locally. The DC Plug-in is invoked on every incoming chunk and performance information regarding its execution is recorded. Characteristics recorded include the per-invocation runtime and the input/output data reduction ratio seen. Also captured is timing information about the simulation's and analytics' execution, and the data movement times between both. The runtime then uses the following formula to estimate the potential benefit of moving the DC Plug-in upstream to the simulation side.

$$Benefit = 1 - T_1 / T_2$$
$$T_1 = \max\{\, Tsim + Tsend \,, Trecv + n \times T_{DCP} + Ta \,\}$$
$$T_2 = \max\{Tsim + T_{dcp} + Tsend \times r, Trecv \times r + Ta\}$$

In this formula, T_1 is the span of steady state of the simulation-analytics pipeline when the DC Plug-in is executed by downstream analytics. T_2 is the span of steady state of the pipeline when the DC Plug-in is executed by the upstream simulation. Note that moving the DC Plug-in upstream not only shifts computation to the simulation side, but also changes the data movement time (as shown in $Tsend \times r$ and $Trecv \times r$ where r denotes data reduction ratio of DC Plug-in). Also note that the DC Plug-in is executed by simulation processes in parallel, but invoked by downstream analytics process one chunk at a time (hence the $n \times T_{DCP}$ portion in T_1).

The simple policy currently used states that if the estimated benefit of deploying the DC Plug-in to the simulation is greater than some threshold value, then the analytics side deploys DC Plug-in to simulation side.

When there are multiple analytics processes, each process is responsible for a subset of the simulation processes from which it receives data. Each analytics process gathers simulation performance information (*Tsim* and *Tsend*) from its corresponding subset along with local information (*Ta*, *Trecv*, *r*, and T_{DCP}). All analytics processes then coordinate to calculate the maximum values, and the root analytics process calculates the estimated aggregate benefit. The placement decision is then broadcast to all analytics processes and is actuated locally by those processes. This uniform decision helps accommodate skews and may reduce load imbalance among peer simulation and peer analytics processes.

Runtime placement of DC Plug-in occurs at I/O timestep boundaries, meaning that the change of placement takes effect for the next I/O action. Such relatively coarse-grained adaptation limits the potential overheads incurred by the process. It also means that this placement policy is not suited for coping with transient variations. Finally, we use the threshold value to express the degree of tolerance for performance variations and to reduce the odds of repetitive ping-pong re-deployment. More sophisticated history-based methods may be used for more general stability control.

We highlight two effects that make it beneficial to place DCPlug-ins on upstream simulation nodes.

(1) Data reduction: if the DC Plug-in can significantly reduce data volume, then placing it upstream reduces data movement time, contention on the interconnect, and analytics side memory usage, resulting in improved Total Execution Time.

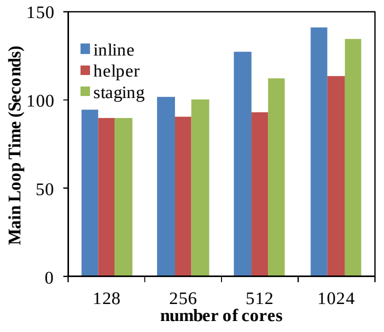

Figure 4. Total Exuetion Time

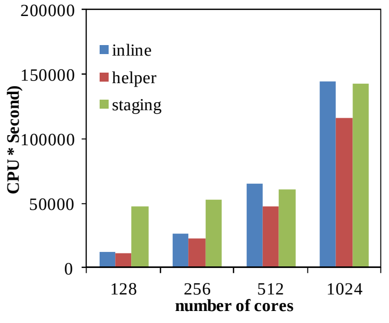

Figure 5. Total CPU Seconds

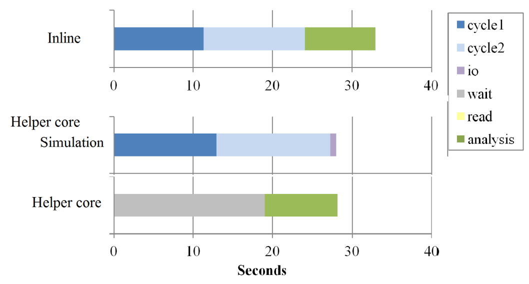

Figure 6. Timeline of GTS and Analytics

(2) Load balancing: if the analytics runtime is slower than the simulation, shifting computation to simulation nodes better balances the load between simulation and analytics, leading to improvements in overall pipeline performance.

V. Performance Evaluation

In this section, we present experimental results obtained from different placements for in situ data analytics for three large scale scientific applications: GTS [44], Pixie3D [13], and S3D[44]. The purpose of the experiments is to examine the difference of end to end performance and cost caused by different placement options of in situ analytics.

A. *Experimental Setup*

Experiments are run on Oak Ridge National Laboratory's Jaugar Cray XT5 and Smoky cluster. At the time of our experiments, Jaguar is equipped with Experiments with 18,688 compute nodes in addition to dedicated login/service nodes. Each compute node contained dual hex-core AMD Opteron 2435 (Istanbul) processors running at 2.6GHz, 16GB of DDR2-800 memory, and a SeaStar 2+ router. The resulting partition contains 224,256 processing cores, 300TB of memory, and a peak performance of 2.3 petaflop/s.

Smoky is an 80 node Linux cluster. Each compute node has four quad-core 2.0GHz AMD Opteron processors (16 core per node), 32 GB of memory (2GB per core), DDR Infiniband interconnect, and access to Spider, the center-wide Lustre-based file system. The compute node runs CentOS Linux and has no swap space.

B. *GTS Performance*

GTS (Gyrokinetic Tokamak Simulation) is a global three-dimensional Particle-In-Cell (PIC) code used to study the microburbulence and associated transport in magnetically confined fusion plasma of tokamak torodial devices. During its run, GTS simulation outputs particle data containing two 2D particle arrays for zions and electrons, respectively, at regular time intervals. The two arrays contain seven attributes for each particles. including coordinates, velocity, weight and particle ID. The particle data is then used by various analysis tasks.

We have implemented an instance of in situ analytics on GTS particle data. The particle data output from GTS is processed by a series of analysis steps, including the calculation of particle distribution function and a range query on the velocity attributes of all particles. The query result is ~20% of the original output particles. 2D histograms on each every 7 attributes of the resulting particles are then generated and written to files. Those 2D histograms are a suitable basis for visualization of parallel coordinates.

We run GTS with a typical production run configuration, which results in particle data output size of 220MB. We run GTS at various scales from 128 to 1024 cores with weak scaling. GTS is run in OpenMP/MPI hybrid mode, as suggested by the GTS team. GTS outputs particle data every two timesteps, as desired by science users. When running with 256 MPI processes, this results in 55GB output data every 35 seconds. Performance comparisons of different placement options are evaluated on Smoky cluster with identical simulation input setups.

When running analytics inline, the GTS processes directly calls analytics routine. On Smoky whose compute nodes has 16 cores each, we run GTS with 4 OpenMP threads per MPI process and place 4 MPI processes on each compute node.

When placing in situ analytics on helper cores, GTS is configured to run with 3 OpeMP threads per MPI process, and every 4 MPI processes are placed on each compute node. In this way, there are 12 GTS OpenMP threads on each compute node. We place 4 analytics processes on the remaining 4 cores of each node (i.e., the helper cores). The GTS processes pass data to analytics processes through shared memory transport.

When placing analytics on staging nodes, we provision additional staging nodes and keep the staging node to compute node count ratio to be 1:8.

The Total Execution Time of the integrated GTS simulation and analytics is shown in Figure 4. Helper core-based placement appears to outperform both inline and staging placements in most configurations. This is because there are sufficient spare resources within compute nodes to accommodate the analysis workload used in the experiment. We found that GTS running with 4 OpenMP threads cannot make full use of all cores within a compute node due to the fact that there is about 10% of it runtime when only main thread is active. Therefore, taking 1 core out of 4 from a GTS process causes 12% slowdown of GTS itself (as indicated by the increase of simulation "cycle1" and "cycle2" in Figure 6); meanwhile, offloading the analysis (which weighs more than 30% of) to this dedicated helper core can reduce/improve the

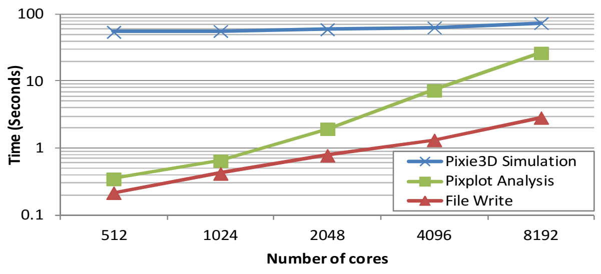

Figure 8. Pixie3D In-Situ I/O Processing.

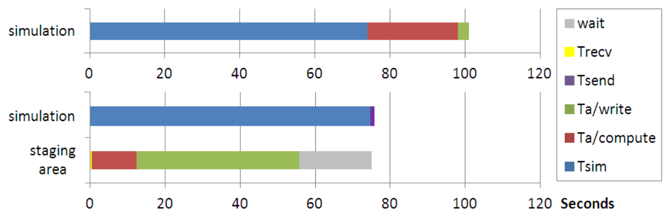

Figure 9. Timeline of Inline and Staging Placement, shown in upper and lower charts, respectively. In both cases Pixie3D runs on 8192 cores.

runtime of GTS compared to an inline configuration where GTS compute cores perform this additional work.

When placing analytics onto separate, additional staging nodes, data movements to staging from compute nodes cause notable slowdown for the simulation. One reason is contention on the interconnect, which we have observed by noting that the simulation's MPI_Barrier call immediately after its ADIOS close call is slowed down significantly because of the interconnect's simultaneous use by the call and for staging data movement. One reason is because data movements use a simple rate-controlled scheduling strategy, rather than the more advanced state-aware data movement methods described in [2].

In terms of Total CPU Hours (shown in Figure 5), staging based placement is the worst because it uses additional nodes but does not reduce execution time in comparison to the other two placements. Since helper core placement can achieve the best execution time without adding nodes, it also results in the best cost.

C. *Pixie3D Performance*

Pixie3D is a 3-Dimensional extendedMHD code that solves the extended MHD equations in 3D arbitrary geometries using fully implicit Newton-Krylov algorithms. As llustrated in Figure 4, Pixie3D I/O processing uses analytics structured as a three-stage pipeline. The first stage is the Pixie3D simulation, generating output data that consists of eight 3D arrays that represent mass density, linear momentum components, vector potential components, and temperature, respectively. The second stage is an analysis code called ”Pixplot”, which performs various diagnostic routines on Pixie3D output data to generate derived quantities, such as curl, gradient, flux, and divergence. The third stage uses the Paraview visualization tool to read the derived quantities generated by Pixplot for visual data exploration.

We run Pixie3D and Pixplot on ORNL's Jaguar Cray XT5. Due to the limitation of Cray Compute Node Linux, co-running two different executables on the same node is not allowed, so we were not able to examine the helper core placement with Pixie3D. Therefore we keep our comparison

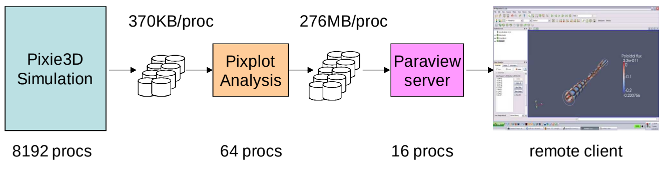

Figure 7. Pixie3D In-Situ I/O Processing.

between inline, staging and offline placements.

The Inline approach performs poorly for the Pixie3D case at large scale. Figure 8 shows the weak scaling of time the for Pixie3D simulation (Tsim) and Pixplot analysis (Ta). The time to write the output of Pixplot from simulation nodes is also shown in the figure. As seen, both Pixplot and I/O have worse scalability than Pixie3D. If Pixplot analysis is performed inline, the time portion spent in Pixplot analysis will increase from 0.06% of the Pixie3D simulation time on 512 cores, to 35.6% on 8192 cores. Also, the time to write analysis results increases from 0.39% of the Pixie3D simulation time on 512 cores to 3.8% on 8192 cores. The insufficient scalability of Pixplot is due to its intensive use of MPI collective communications and ”aggregating-to-one-process” style computation, the latter not uncommon in analysis codes.

When running the Pixie3D simulation on 8192 cores with Pixplot placed in a staging area of 64 cores, data movement between Pixie3D and Pixplot is via FlexIO's RDMA-based transport. The detailed timing in Figure 8 shows that placing Pixplot in staging area can effectively overlap analysis and writing (Ta/compute and Ta/write, respectively) with simulation (Tsim). The simulation-side visible send time (Tsend) is 1.3% of Tsim. The staging area side data movement time (Trecv) is less than 0.5 seconds. Overall, with a staging area which is of 0.78% the simulation nodes, the measured speedup of Staging over Inline is 1.333. When running Pixie3D on 8192 cores, the minimal size of staging area determined by the minimal memory space required to run Pixplot, is 64 cores. Figure 9 shows that even when the staging area is of the minimal feasible size it still spends 25% of time waiting for output data from simulation, indicating that the staging area is already over-provisioned. Also note that scaling down Pixplot from 8192 to 64 cores actually reduces its runtime by half. This is due to the communication-bound nature of Pixplot and improved locality when data are aggregated onto a smaller number of cores.

Compared to the offline placement, by which Pixie3D writes output into a BP file using MPI-IO and Pixplot reads data from the file for analysis, the Staging approach hides file write latencies almost completely and improves Pixie3D's total execution time by 2.3% to 16.5% among 5 test runs at scale of 8192 cores.

D. *S3D Performance*

The third application is S3D. S3D is a state-of-the-art flow solver for performing direct numerical simulation (DNS) of turbulent combustion. We use a modified version of S3D code

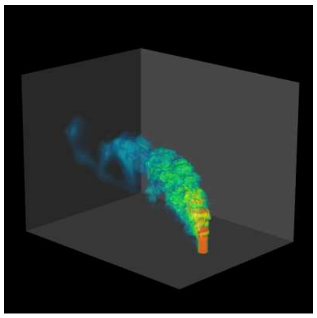

Figure 10. S3D Visualization Output.

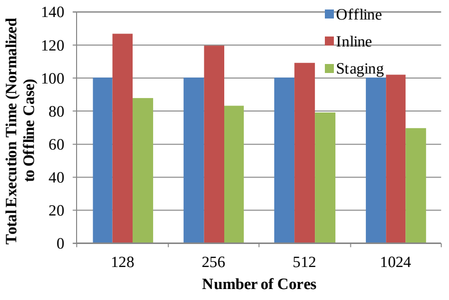

Figure 11. Total Execution Time

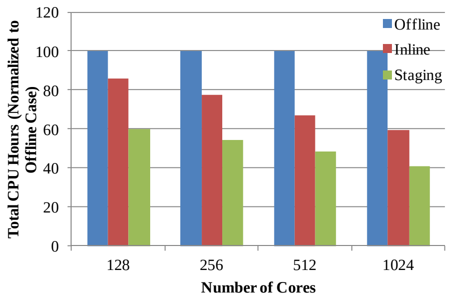

Figure 9. Total CPU Hours

called S3D_Box created by the S3D team for our test. S3D_Box performs a portion of the full S3D simulation. We set S3D_Box to load initial data from a restart file originally generated by a S3D run on 96000 cores. During its execution, S3D_Box periodically outputs species data which are 22 3-dimensional double-typed arrays. The per-process output data size is 1.7MB, which is the same as typical production S3D simulation. The species data is fed into a parallel visualization code to render images for each every species (Figure 10 shows a sample image for H2).

We run S3D_Box and the parallel visualization code on Smoky cluster at different scales using weak scaling. At each scale, we measure the performance of different placement options. As shown in Figure 11 and 12, the staging based placement gives the best performance (in terms of Total Execution Time) and cost (in terms of Total CPU Hours). Similar to the Pixie3D case, staging placement is better than inline due to the pipelining effect, and offloading visualization computation (and writing rendered image to files in PPM format) to a separate staging area via asynchronous data movement can hide the cost of I/O and analytics computation.

E. *Utility of Data Conditioning Plug-in*

We use the GTS application to demonstrate the utility of Data Conditioning Plug-ins. We run GTS on 128 cores and run the in situ analytics described earlier on 1 staging node. During the execution, the analytics instantiates a sampling DC Plug-in which samples one out of every 100 particles of the original simulation output data. The communication of the DCPlug-in code is performed between timesteps and dynamic code generation requires only .5msecs, so code deployment has an insignificant impact on the running system. The resulting sampling code requires only 220 x86 instructions.

Due to the substantial data reduction ratio, the FlexIO runtime deploys it onto simulation processes. Table 1 compares the steady state time before vs. after DCPlug-in is deployed. The sampling DCPlug-in helps reduce data movement time and downstream analytics computation time. It also helps reduces simulation time due to reduced network contention.

The ability to dynamically deploy DC Plug-in computation to remote simulation side can be used to achieve various runtime adaptation actions. The simple sampling example shown here, for instance, can be used to dynamically throttle data volume in case the analytics computation becomes seriously slowed down, so that the back pressure to simulation can be mitigated.

TABLE I. IMPACT OF SAMPLING DC PLUG-IN

	simulation		***analytics***		
	compute	***write***	***read***	***compute***	***wait***
before	23.2	0.208	5.61	6.52	11.3
after	22.0	0.0431	0.349	0.914	20.8

F. *Summary*

Several interesting observations can be made from the experiment results. First, performing data analytics in situ can significantly improve Total Execution Time and/or Total CPU Hours compared to offline file-based approach for all four applications, indicating the promise of in situ data analytics to address the I/O bottleneck on HEC platforms.

Second, there is a notable difference between alternative placements of in situ analytics, and the best placement varies for different simulation/analytics/machine combinations. Particularly, those analytics which are non-scalable and expend data are better offloaded from simulation and placed in staging nodes or helper cores. On the other hand, scalable, data reduction operations may be better placed inline or near the simulation. Such diversity justifies the flexible placement functionality offered by FlexIO.

Third, FlexIO system is capable of supporting a variety of simulation and analytics workloads at large scale through flexible placement options, efficient data movement, and dynamic deployment of data manipulation functionalities.

VI. RELATED WORK

In-situ data analytics and visualization has gained much recent attention from the HPC community. Current work on in-situ data analytics falls into two categories: (1) in-situ data analytics and visualization algorithms, including indexing, compression [20], feature extraction [6] [41], and various visualization techniques [44][38], and (2) supporting tools and infrastructures such as those mentioned in Section 2. Among those supporting infrastructures, SmartTap [3] used runtime binary code generation to improve buffer management and data movement from simulation and staging nodes. CoDS [46] provides a runtime framework to host in-situ analytics tasks organized as DAGs on either staging nodes or compute nodes and supports MxN data exchange among parallel tasks. GLEAN [39][40] enables data analysis between BlueGene compute nodes and analysis clusters.

Compared to other supporting infrastructures, FlexIO aims to provide greater flexibility in where to place analytics, which is a key feature to support the growing diverse set of analytics scientists may want to run with their simulations. Using the widely adopted ADIOS I/O interface as the coupling mechanism between simulation and analysis codes makes our system complementary with existing analysis tools. In fact, we have been working with the ParaView [11] and VisIt teams to integrate ADIOS with these visualization engines to enable online visualization of simulation output data [38]. The metrics-driven placement strategy supported by FlexIO can be generalized and adopted by other systems for determining the proper placement of in-situ analytics. FlexIO does not yet support the MxN couplings needed to associate arbitrary analytics with simulations, but our team is leveraging prior work in [4] on publish/subscribe systems [9] for doing so.

Computation placement is an extensively studied topic in distributed systems due to its significant impact on application performance and cost. Particularly relevant is previous work on computation placement within the Active Storage context. Abacus [2] uses an online performance model to guide the dynamic placement of application and file system functions among clients and servers to adapt to a variety of application and system runtime characteristics, but it assumes a progressive, per-record computation. [42] studies load distribution of a class of streaming computation in an active storage system. Diamond [17] aggressively places filters to data sources to reduce search operation costs.

The importance of placement has also been exploited in other distributed computing models such as Streaming Processing and Data Grid. Streaming operator placement on wide-area overlay network has been studied in [33]. COLA [19] applies graph partitioning to place a streaming processing dataflow onto a cluster of nodes with load balance and throughput as the major optimization objectives. Armada [32] uses similar graph partitioning techniques to distribute on-network operations within a Data Grid environment to improve I/O access performance. Comparing to those work, our system focuses more on optimizing the end-to-end performance and cost of both simulation and in-situ analysis.

Scientific workflow systems like Pegasus [7] and Kepler [22] are often used to orchestrate the automatic execution of analysis tasks. They mainly use files as the data exchange mechanism. The explosive growth of scientific data, however, will stress the I/O system and can easily overwhelm overall workflow performance. Therefore, it is expected that in the near future, more and more analysis will be deployed online and run in-situ with simulation, especially those which can achieve early data reduction or prepare data for better use by downstream analyses. Our system can be readily interact with scientific workflow systems to enable such online usage.

At the implementation level, our shared memory transport borrows cache optimizations from FastForward's lock-free queue [15]. There is also existing work on high performance MPI intra-node messaging implementations [6].

VII. Conclusions and Future Work

The FlexIO system presented in this paper is designed to flexibly place in situ data analytics so as to enable developers to better manage the combined resources used by simulation codes running jointly with online analytics and visualization on high end machines. Evaluation results obtained with the large scale scientific applications GTS, S3D and Pixie3D verify the argument for flexible placement and reveal important trade-offs among the placement options of inline, on-node, in-staging, or offline. We also offer an initial proposal for a streaming-like read API within the ADIOS adaptive I/O system, called chunk read, and demonstrate its suitability for some analysis application needs. Finally, online placement of Data Conditioning Plug-ins aids in reducing the data volumes moved across nodes of the petascale machine, guided by end user visible optimization metrics, described next.

Future work focuses on enhancements to FlexIO for dynamic resource allocation and placement, to deal with cases where analytics and/or simulations vary while they run (e.g., AMR codes). The modeling will be strengthened to serve as a decision engine for this dynamic allocation. Additional development to support the CPU, GPU, and Gemini Interconnect upgrades on the Cray XK6 platform.

Acknowledgment

We thank Jay Lofstead from Sandia National Laboratory for his helpful comments on this paper. This work was funded in part by Sandia National Laboratories under contract DE-AC04-94AL85000, by the DOE Office of Science, Advanced Scientific Computing Research, under award number DE-SC0005505, program manager Lucy Nowell, and by the Department of Energy under Contract No. DEAC05-00OR22725 at Oak Ridge National Laboratory. Additional support came from the resources of the National Center for Computational Sciences at Oak Ridge National Laboratory, a grant from NSF as part of the HECURA program, a grant from the Department of Defense, a grant from the Office of Science through the SciDAC program, and the SDM center in the ASCR office.

References

[1] S. Ashby, P. Beckman, J. Chen, P. Colella, B. Collins, D. Crawford, J. Dongarra, D. Kothe, R. Lusk, P. Messina, T. Mezzacappa, P. Moin, M. Norman, R. Rosner, V. Sarkar, A. Siegel, F. Streitz, A. White, and M. Wright, "The Opportunities and Challenges of Exascale Computing," Summary Report of the Advanced Scientific Computing Advisory Committee (ASCAC) Subcommittee, 2010, pp. 1-77.

[2] K. Amiri, D. Petrou, G. R. Ganger, and G. A. Gibson, "Dynamic Function Placement for Data-intensive Cluster Computing,"Proc.Annual conference on USENIX Annual Technical Conference (ATC'00), 2000.

[3] H. Abbasi, G. Eisenhauer, M. Wolf, K. Schwan, and S. Klasky, "Just In Time: Adding Value to the I/O Pipelines of High Performance Applications with JITstaging," Proc. ACM Symp. on High-Performance Parallel and Distributed Computing (HPDC'11), 2011, pp. 27-36.

[4] H. Abbasi, M. Wolf, K. Schwan, G. Eisenhauer, and A. Hilton, "Xchange: Coupling Parallel Applications in A Dynamic Environment," Proc. IEEE International Conference on Cluster Computing (Cluster'10), 2010, pp. 471-480.

[5] C. S. Chang, and S. Ku, “Spontaneous Rotation Sources in a Quiescent Tokamak Edge Plasma,” Proc. Physics of Plasmas, 2008.

[6] L. Chai, P. Lai, J. W. Jin, and D. K. Panda, “Designing an Effiecient Kernel-level and User-level Hybrid approach for MPI Intra-node Communication on Multi-core Systems,”, Proc. International Conference on Parallel Processing, (ICPP’08), 2008,pp. 222-229.

[7] E. Deelman, G. Singh, M. Su, etc. “Pegasus: A Framework for Mapping Complex Scientific Workflow onto Distributed Systems,” Proc. Journal of Scientific Programming, 2005, pp. 219-237.

[8] G. Eisenhauer, M. Wolf, H. Abbasi, S. Klasky, and K. Schwan, “A Type System for High Performance Communication and Computation,” Proc. The Workshop on D3Science associated with e-Science11, 2011.

[9] G. Eisenhauer, M. Wolf, H. Abbasi, S. Klasky, and K. Schwan, “Event-based Systems: Opportunities and Challenges at Exascale,” Proc. The 3rd ACM International Conference on Distributed Event-Based Systems, 2009, pp. 16-25.

[10] Message Passing Interface Forum. MPI-2: Extension to The Message-Passing Standard. 1997.

[11] N. Fabian, K. Moreland, D. Thompson, A. C. Bauer, P. Marion, B. Geveci, M. Rasquin, and K. E. Jansen, “The ParaView Coprocessing Library: A Scalable, General Purpose In Situ Visualization Library,” Proc. IEEE Symp. on Large-Scale Data Analysis and Visualization (LDAV2011), 2011, pp 89-96.

[12] B. Fryxell, K. Olson, P. Ricker, F. X. Timmes, M. Zingale, D. Q. Lamb, P. MacNeice, R. Rosner, and H. Tufo, “FLASH: An Adaptive Mesh Hydrodynamics Code for Modelling Astrophysical Thermonuclear Flashes,” Astrophysical Journal Supplement, 2000, pp 273-334.

[13] The HDF Group, “Hierarchical Data Format Version 5,” 2000-2010, http://www.hdfgroup.org/HDF5.

[14] A. Gerndt, B. Hentschel, M. Wolter, T. Kuhlen, and C. Bischof, “VIRACOCHA: An Efficient Parallelization Framework for Large-Scale CFD Post-Processing in Virtual Environments,” Proc. ACM/IEEE Conference on Supercomputing (SC04), 2004, pp. 50-61.

[15] J. Giacomoni, T. Moseley, and M. Vachharajani, “Fastforward for Efficient Pipeline Parallelism: A Cache-Optimizaed Concurrent Lock-free Queue,” Proc. ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP’08), 2008.

[16] S. Hodson, S. Klasky, Q. Liu, J. Lofstead, N. Podhorszki, F. Zheng, M. Wolf, T. Kordenbrock, H. Abbasi, N. Samatova, “ADIOS 1.3.1 User’s Manual,” Oak Ridge National Laboratory, 2011, pp. 1-95.

[17] L. Huston,etc. “Diamond: A Storage Architecture for Early Discard in Interactive Search,” Proc. The 3rd USENIX Conference on File and Storage Technologies, 2004.

[18] J. C. S. E. R. Hawkes, R. Sankaran, and J. H. Chen, “Direct Numerical Simulation of Turbulent Conbustion: Fundamental Insights towards Predictive Models,” Proc.Journal of Physics: Conference Series, 2005, pp. 65-79.

[19] R. Khan, K. Hildrum, etc. “Cola: Optimizing Stream Processing Application via Graph Partitioning,” Proc.the 10th ACM/IFIP/USENIX International Conference on Middleware, 2009.

[20] E. Jeannot, B. Knutsson, M. Bjorkman, “Adaptive Online Data Compression,” Proc. ACM Symp. on High-Performance Parallel and Distributed Computing (HPDC’02), 2002, pp. 379-388.

[21] S. Klasky, S. Ethier, Z. Lin, K. Martins, D. McCune, and R. Samtaney, “Grid-based Parallel Data Streaming Implemented for the Gyrokinetic Toroidal Code,” Proc. ACM/IEEE Conference on Supercomputing (SC03), 2003, pp. 24-35.

[22] B. Ludascher, I. Altintas, C. Berkley, D. Higgins, E. Jaeger, M. Jones, E. A. Lee, J. Tao, and Y. Zhao, “Scientific Workflow Management and the Kepler System,” Proc. Special issue: Workflow in Grid Systems, 2006.

[23] M. Lin, S. S. Vazhkudai, A. R. Butt, F. Meng, X. Ma, Y. Kim, C. Engelmann, and G. Shipman, “Functional Partitioning to Optimize End-to-End Performance on Many-core Architectures,” Proc. ACM/IEEE Conference on Supercomputing (SC10), 2010, pp. 1-12.

[24] S. N. Laboratory, Cth shock physics, http://www.sandia.gov/CTH/. Jan. 2012.

[25] J. F. Lofstead, R. Oldfield, T. Kordenbrock, and C. Reiss, “Extending Scalability of Collective I/O Through Nessie and Staging,” Proc. 6th Parallel Data Storage Workshop (PDSW 2011), 2011, pp. 7-12.

[26] J. F. Lofstead, M. Polte, G. A. Gibson, S. Klasky, K. Schwan, R. Oldfield, M. Wolf, and Q. Liu, “Six Degrees of Scientific Data: Reading Patterns for Extreme Scale Science I/O,” Proc. ACM Symp. on High-Performance Parallel and Distributed Computing (HPDC’11), 2011, pp. 49-60.

[27] J. F. Lofstead, F. Zheng, S. Klasky, and K. Schwan, “Adaptable, Metadata Rick I/O Methods for Portable High Performance I/O,” Proc. IEEE International Parallel and Distributed Processing Symp (IPDPS’09), 2009, pp. 1-10.

[28] J. F. Lofstead, F. Zheng, Q. Liu, S. Klasky, R. Oldfield, T. Kordenbrock, K. Schwan, and M. Wolf, “Managing Variability in the I/O Performance of Petascale Storage Systems,” Proc. ACM/IEEE Conference on Supercompuing (SC10), 2010, pp. 1-12.

[29] O. E. B. Messer, S. W. Bruenn, J. M. Blondin, W. R. Hix, A. Mezzacappa, and C. J. Dirk, “Petascale Supernova Simulation with Chimera,” Proc. Journal of Physics Conference Series, 2007, 78(1) pp. 1-5.

[30] P. K. Moreland, R. Oldfield, and S. Klasky, “Examples of In Transit Visualization,” Proc. Petascale Data Analytics Challenges and Opportunities (PDAC-11), 2011.

[31] R. Oldfield, S. Arunagiri, P. J. Teller, S. R. Seelam, M. R. Varela, R. Riesen, and P. C. Roth, “Modeling the Impact of Checkpoints on Next-generation Systems,” Proc. IEEE Conference on Mass Storage System and Technologies (MSST 2007), 2007, pp. 30-46.

[32] R. Oldfield, and D. Kotz, “Improving Data Access for Computational Grid Application,” Proc. IEEE International Conference on Cluster Computing (Cluster’10), 2010, pp. 79-89.

[33] P. Pietzuch, etc. “Network-aware Operator Placement for Stream-processing Systems,” Proc. The 22nd International Conference on Data Engineering (ICDE’06), 2006, pp. 49-58.

[34] J. Piernas, J. Nieplocha, E. J. Felix, “Evaluation of Active Storage Strategies for the Lustre Parallel File System,” Proc. ACM/IEEE Conference on Supercomputing (SC07), 2007, pp. 1-10.

[35] A. Singh, P. Balaji, and W. Feng, “GePSeA: A General-Purpose Software Acceleration Framework for Lightweight Task Offloading,” Proc. The 38th International Conference on Parallel Processing (ICPP), 2009, pp. 261-268.

[36] Visualizaiton Toolkit, Opens Source 3D Computer Graphics, Image Processing and Visualization, http://www.vtk.org.

[37] T. Tu, H. Yu, L. Ramirez-Guzman, J. Bielak, O. Ghattas, K. Ma, and D. R. O’Hallaron, “Scalable Systems Software – From Mesh Generation to Scientific Visualization: an End-to-End Approach to Parallel Supercomputing,” Proc. ACM/IEEE Conference on Supercomputing (SC06), 2006, pp. 1-15.

[38] Paraview VisIt, OpenSource Scientific Visualization and Graphical Anaysis Tool, https://wci.llnl.gov/codes/visit

[39] V. Vishwanath, M. Hereld, M. E. Papka, R. Hudson, G. Cal Jordan IV, and C. Daley, “In Situ Data Analysis and I/O Acceleration of FLASH Astrophysics Simulation on Leadership-Class System Using GLEAN,” Proc. SciDAC, Journal of Physics: Conference Series, 2011.

[40] V. Vishwanath, M. Hereld, M. E. Papka, “Toward Simulation-Time Data Analysis and I/O Acceleration on Leadership-Class Systems,” Proc. IEEE Symp. on Large-Scale Data Analysis and Visualization (LDAV2011), 2011, pp. 9-11.

[41] K. Wu, S. Ahern, E.W. Bethel, J. Chen, H. Childs, E. Cormier-Michel, C. Geddes, J. Gu, H. Hagen, B. Hamann, W. Koegler, J. Lauret, J. Meredith, P. Messmer, E. Otoo, V. Pere., A. Posk., P. O. Rubel, A. Sho., A. Sim, K. Stock., G. Weber, and W. Zhang, “FastBit: Interactively Searching Massive Data,” Proc. SciDAC, Journal of Physics: Conference Series, 2009.

[42] R. Wick, J. S. Chase, and J. S. Vitter, “Distributed Computing with Load-managed Active Storage,” Proc. ACM Symp. on High-Performance Parallel and Distributed Computing (HPDC’02), 2002.

[43] W. X. Wang, Z. Lin, W. M. Tang, W. W. Lee, S. Ethier, J. L. V. Lewandowski, G. Rewoldt, T. S. Hahm, and J. Manickam, “Gyro-kinetic

Simulation of Global Trubulent Tranport Properties in Tokamak Experiments," Proc. Physics of Plasmas, 2006, pp 59-64.

[44] H. Yu, C. Wang, R. W. Grout, J. H. Chen, and K. Ma, "In-situ Visualizaiton for Large-scale Combustion Simulations", Proc. IEEE Computer Graphics and Applications, 2010, pp. 45-57.

[45] F. Zheng, H. Abbasi, J. Cao, J. Dayal, K. Schwan, M. Wolf, S. Klasky, N. Podhorszki, "In-Situ I/O Processing: A Case for Location Flexibility," Proc. 6th Parallel Data Storage Workshop (PDSW 2011), 2011, pp. 37-42.

[46] F. Zheng, H. Abbasi, C. Docan, J. F. Lofstead, Q. Liu, S. Klasky, M. P, N. Podhorszki, K. Schwan, and M. Wolf, "Predata-Preparatory Data Analytics on Peta-scale Machines", Proc. IEEE International Parallel and Distributed Processing Symp (IPDPS'10), 2010, pp 1-12.