2013 IEEE 27th International Symposium on Parallel & Distributed Processing

# FlexIO: I/O Middleware for Location-Flexible Scientific Data Analytics

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
*Oak Ridge National Laboratory, Oak Ridge, TN, USA
†Department of Computer Science and Engineering, University of Nebraska-Lincoln, Lincoln, NE, USA

***Abstract*—Increasingly severe I/O bottlenecks on High-End Computing machines are prompting scientists to process simulation output data online while simulations are running and before storing data on disk. There are several options to place data analytics along the I/O path: on compute nodes, on separate nodes dedicated to analytics, or after data is stored on persistent storage. Since different placements have different impact on performance and cost, there is a consequent need for flexibility in the location of data analytics. The FlexIO middleware described in this paper makes it easy for scientists to obtain such flexibility, by offering simple abstractions and diverse data movement methods to couple simulation with analytics. Various placement policies can be built on top of FlexIO to exploit the trade-offs in performing analytics at different levels of the I/O hierarchy. Experimental results demonstrate that FlexIO can support a variety of simulation and analytics workloads at large scale through flexible placement options, efficient data movement, and dynamic deployment of data manipulation functionalities.**

***Keywords- I/O, In Situ Data Analytics, Placemen, Flexibility***

## I. INTRODUCTION

Peta-scale scientific simulations running on high end computing machines in domains like Fusion [22], Astrophysics [14], and Combustion [20] now routinely generate terabytes of data in a single run, and these data volumes are only expected to increase. Since this massive data is key to scientific discovery, the ability to rapidly store, move, analyze, and visualize data is critical for scientists' productivity. Yet there are already serious I/O bottlenecks on current HEC machines, and movement toward the Exascale is further accelerating this trend.

Online data analytics has emerged as an effective way to overcome the I/O bottleneck for scientific applications running at the Peta-Scale and beyond. By processing data as it moves through the I/O path, online analytics can extract valuable insights from live simulation output in a timely manner, better prepare data for subsequent deep analysis and visualization, and gain improved performance and reduced data movement cost (both in time and in power) compared to solely file-based offline approaches. The utility of the approach is evident from its wide adoption of leading scientific applications like the S3D combustion simulation [49], the GTC [22] and GTS [47] fusion simulations, Trillions [27], CTH [29], and FLASH [42].

For real-time processing of the outputs generated by large scale simulations, a key problem to address is "where" analytics are placed along the I/O path: on compute nodes integrated with application codes, on compute nodes as separate software components, on nodes dedicated to analytics (also termed 'staging nodes'), or offline (after data is placed into persistent storage) (as illustrated in Figure 1). Placing data analytics involves deciding the resources to allocate to analytics computation and realizing the data movements between simulation and analytics. Previous experimental results and analytical models [52] show that analytics placement can significantly impact the performance (e.g., runtime) and cost (e.g., CPU hours) of the coupled simulation and analytics and that the best placement depends on the particular analytics codes, data volumes, scale of operation, and machine characteristics. The consequent insight is that *no single, specific placement will be 'best' for all applications and analytics*.

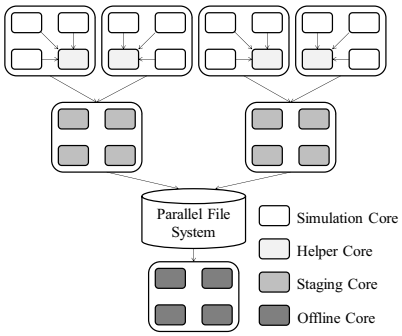



Figure 1. Analytics Placement Options along I/O Path.

Such variation has important implications to both scientists and the software that supports analytics. Scientists desire the performance benefit from good placement, but it is a burden for them to tune placement every time a different analytics code is run, especially when this requires significant coding effort. There is a need, therefore, for infrastructure that makes it easy to decide, enforce, change, and tune analytics placement. At the same time, if such an analytics software infrastructure aims to support a broad range of simulations and analytics, lacking placement flexibility limits its applicability, since fixed placements may cause negative or even disastrous impact on application performance at large scale. Flexible placement, therefore, is a critical element of analytics infrastructure.

### A. Placement Support in Existing Systems

A number of systems and tools have been developed to support online analytics and visualization.

1530-2075/13 $26.00 © 2013 IEEE
DOI 10.1109/IPDPS.2013.46

1) *Analytics and Visualization Libraries.* ParaView's co-processing library [13], VisIt's remote visualization [43], and other online visualization work [41][49] offer software libraries with collections of analytics and visualization routines. Although those libraries do not impose restrictions on where they can be executed and can be directly invoked as function calls, the libraries must rely on external data movement support to run in staging nodes or on dedicated compute node cores, termed 'helper' cores in Figure 1.

2) *Helper Core Processing.* Functional partitioning [24], software accelerator [35], and Damaris [10] perform file I/O and analytics on dedicated cores in compute nodes and leverage shared memory to pass data. Such placement can be beneficial for some cases (e.g., when the simulation cannot scale to use all cores), but its applicability is restricted by the memory space on compute nodes made available by the simulation.

3) *Staging Area Processing.* Data Services [2], Nessie [27], GLEAN [40], and HDF5/DSM [5] use a set of additional staging nodes to buffer and process simulation output data. Placing analytics on staging nodes requires provisioning additional resources, and moving massive simulation output data to staging area can be costly and negatively interfere with simulation.

4) *Active Storage.* Certain computational routines may be deployed directly on storage nodes and triggered to operate when data is written and/or read [33]. Due to resource limitations on storage nodes, the deployed analytics are usually restricted kernel functions. Further, access to storage nodes is not generally allowed in production environments.

5) *Offline Processing.* Data written to storage is read back for additional or long-term analysis or visualization [16], typically assisted by workflow tools [9][25].

6) *Hybrid Online Processing.* Systems like PreDatA [53] and DataSpaces [4] provide programming models that allow analytics to be broken down into separate pieces and deployed onto both compute nodes and staging nodes.

Most existing systems support certain, fixed placement choices (Categories 1-5) and therefore, each is efficient or applicable to certain classes of analytics. Some permit analytics to run at different locations (Category 6), but require adopting particular coding patterns or re-placement involves substantial re-coding. Further, they do not support seamlessly switching analytics between online and offline, nor do they allow dynamic changes in analytics placement. Typical causes of limited flexibility are (1) inability to handle the alternative data movements between simulation and analytics required by different placement options (i.e., supporting inter-node and intra-node data transfer and file I/O); (2) lack of uniform higher-level interfaces that hide data movement detail; (3) imposition of specific computation models for analytics; and (4) inability to achieve those requirements with high performance and scalability.

### B. Technical Contributions

The FlexIO middleware described in this paper is designed to provide data movement between simulation and analytics with both high performance and location flexibility. It offers the following functionalities:

1) Flexibility in where analytics codes are run -- on compute nodes, on staging nodes, and/or any combination thereof. This is realized through FlexIO's high performance intra- and inter-node data movement transports which are implemented with shared memory queues and RDMA, respectively, and tuned for high throughput, contention-avoidance, and memory efficiency.

2) Analytics placements can be altered without requiring application codes to be changed. FlexIO's high level programming interface makes changes in placement transparent to simulation and analytics codes. Users can even seamlessly switch analytics to run offline when there are insufficient online resources for their timely execution.

3) Runtime performance monitoring collects information about computation and data movement that is useful to scientists and automated runtime management systems for performance understanding and placement decisions.

4) Mobile codelets, termed "Data Conditioning (DC) Plug-ins", can be dynamically deployed and migrated along the I/O path, to perform useful on-the-fly data manipulation such as data selection, sampling and transformation.

5) FlexIO enables various placement policies to exploit the location flexibility for tuning application performance, CPU usage, and data movement cost. Based on FlexIO, we implement a holistic placement policy which reduces both inter and intra program data movement costs. We also devise a node topology aware policy which takes into account the impact of cache topology on analytics placement.

FlexIO operates on both Infiniband and the new Cray XK6 with Gemini interconnect. It has been applied to two leadership scientific applications: GTS fusion simulation and S3D combustion simulation. Experiments show that leveraging the flexibility enabled by FlexIO to tune analytics placement can improve applications' performance by up to 30% compared to inline-only solutions and the benefit is more evident at larger scales..

The remainder of the paper is organized as follows. Section II describes the design and implementation of the FlexIO middleware. Section III describes how to automate placement of analytics driven by performance and cost metrics. Section IV shows performance improvements for two large-scale scientific applications due to flexible analytics placement. Section V reviews related work and Section VI concludes the paper.

## II. FLEXIO DESIGN AND IMPLEMENTATION

### A. System Overview

The FlexIO software stack is depicted in Figure 2. Simulation and analytics codes use the ADIOS [28] read/write API for data exchange. The FlexIO runtime handles buffer management, parallel data re-distribution, and performance monitoring. It also manages "Data Conditioning Plug-Ins" which are mobile codelets compiled, deployed, and executed at runtime for on-the-fly data manipulation. Runtime performance monitoring provides information for scheduling data movements and for dynamic DC Plug-in placement and can also be retained for offline performance tuning. At the lowest transport level, FlexIO uses efficient

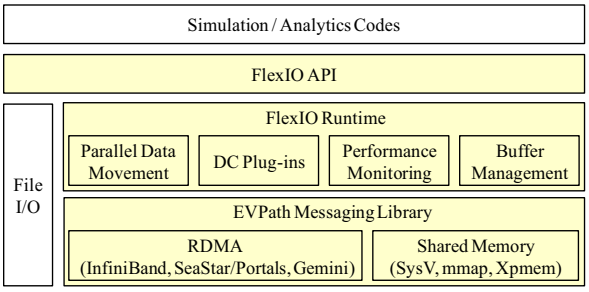



Figure 2. FlexIO Software Stack

RDMA and shared memory data movements for inter- and intra- node movements, respectively. The choice of low level transport is automatically configured according to the placement of online analytics.

FlexIO leverages the ADIOS [28] parallel I/O library which provides meta-data rich read/write interfaces to simulation and analysis codes. ADIOS has a set of built-in I/O methods under its higher level API to support various file I/O methods (such as MPI-IO, HDF5, and NetCDF) as well as data staging methods [2] [4]. 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 and is integrated with popular analysis and visualization tools that include Matlab, ParaView, and VisIt.

FlexIO inherits from ADIOS useful features like its high level API and file I/O methods (to enable offline placement), and its implementation benefits from our previous work on RDMA-based data movement and staging. New functionalities compared to such prior work include an ADIOS stream read interface, methods for parallel memory-to-memory multi-dimensional array re-distribution, efficient data movement for Cray XK6 systems with the Gemini interconnect, cache efficient on-node data movement via shared memory, intelligent buffer management, enriched performance monitoring, and mobile DC Plug-ins. In total, FlexIO is a system that supports diverse analytics placements via efficient built-in methods for data movement between a simulation and analytics components.

### *B. Hig Level Interface*

The high-level interface of FlexIO extends the existing ADIOS file read/write API with three goals: 1) expressiveness in supporting common I/O patterns for simulation and analytics codes; 2) backwards compatibility with the existing ADIOS file I/O interface; and 3) easily switched underlying transports.

Conceptually, the FlexIO interface allows simulations to pass data to analytics via "files", and to operate on these "files" in either file or stream modes. In both modes, the data model is compatible with the existing ADIOS data model, where the simulation output data is logically time-indexed, and each timestep of output data is a group of variables of scalar or array types. In the file mode, data is written to the file system and read back by analytics, using one of ADIOS' file I/O methods. The file mode is for backwards compatibility with the existing ADIOS file I/O interface.

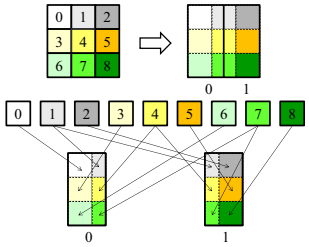



Figure 3. Global Array Re-distribution. A 2D array is distributed among 9 simulation processes and passed to 2 analytics processes.

The newly added stream mode is specifically intended for memory-to-memory data movement between simulation and online analytics. Here, the simulation creates a "file" with some unique name, and the analytics opens the named "file", but internally, this establishes connections to simulation processes via the underlying transport. Simulation processes, then, periodically write data to the "file", and the data is passed to analytics as return parameters of their read calls (again, the underlying transport handles actual data movement). When the simulation closes the "file", the connections are closed by the transport and analytics components receive End-of-Stream as return values from their read calls. As a result, stream mode is compatible with file I/O in that it can be switched with file mode without code changes.

For stream mode, there are two common I/O patterns for high end applications. The first is for process-group-oriented data exchanges: during each I/O timestep, the variables written from each simulation process are conceptually packed into a group, called "Process Group", and the analytics specifies the process groups it wants to read by simulation processes' MPI ranks. The other pattern is a global array data exchange, where some multi-dimensional array, distributed among several simulation processes, is passed to several analytics processes. As in other MxN data exchanges [3], however, analytics processes may specify an array distribution or layout different from that present on the simulation side. In response, FlexIO properly chunks, splits, transfers, and re-organizes the array data exchanged between simulation and analytics, as shown in Figure 3.

The high-level API makes it easy to change underlying transports, without the need to change applications. A one-line update to the configuration file is sufficient to switch between file I/O and online data movement transports, and intra- vs. inter-node transports are automatically configured according to the placements of communicating simulation and online analytics processes. To tune transports, transport-specific parameters specified as hints in an XML configuration file are passed to the FlexIO runtime. We refer readers to [18] for details of the interface specification.

### *C. Data Movement Protocols*

There is considerable complexity in presenting to end users a convenient API, yet also providing placement flexibility and high performance. Key to this complexity is that the FlexIO runtime must translate the high-level 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 MxN mapping, i.e., which simulation process should send which piece of its data to which analytics processes, is determined by the overlapping portion(s) of data specified in the simulation's write and analytics' read calls, respectively. Establishing the necessary connections and transferring data efficiently at large scales is a non-trivial task. Below we describe the connection management and data transfer protocols used by the FlexIO runtime.

*1) Connection Management*

Before actual data movement, simulation and analytics programs connect to each other via assistance from an external directory server. To avoid overloading this server, simulation and analytics processes, respectively, elect a local coordinator. When creating a file in stream mode, the coordinator of the simulation registers with the directory server a file name associated with its own contact information. When the analytics opens that file, its coordinator looks up the server with the file name, retrieves the contact information of the simulation's coordinator, and makes a connection with it. The directory server is involved only in discovery and connection setup and is not in the critical path of actual data movements.

*2) Data Movement*

To move global array data between two parallel programs, array data must be transferred according to the data distributions at both sides. The FlexIO write and read API captures the array distribution among simulation and analytics processes, respectively. Based on this information, the FlexIO runtime generates the re-distribution mapping. At each side, coordinators first gather array distributions for all processes (Steps 1.s and 1.a, respectively), exchange the distribution information with each other (Step 2), and then broadcast the peer-side distribution to all processes (Step 3). At this point, each process knows the array distribution of all other peer processes, so that it can calculate the mapping independently. Each sender process packs strides for each receiver process with overlapping array index range, and sends the packed strides to each receiver process (Step 4.s). Each receiver prepares a receive buffer based on the mapping and copies received strides into the appropriate target buffer (Step 4.a). The Process-Group-oriented data movement pattern is implemented in a similar fashion.

There are several optional optimizations. First, write side calls can be either synchronous or asynchronous. The asynchronous API helps overlap data movement with other activities like the simulation's computation.

The second optimization is batching. The default granularity of data movement is per-variable. Users can instruct FlexIO to pack multiple variables and transfer them in a batch. This will cause both handshaking and data messages to be aggregated.

The third optimization is caching to reduce the cost of handshaking. By default, the complete handshaking protocol (Step 1 to 4 as described above) is performed for each variable at each I/O timestep. If distribution information and buffer addresses are unchanged across timesteps, then some or all of the handshaking steps can be avoided by reusing existing information from previous timesteps. The sender or receiver can inform the FlexIO runtime about three possible caching options:

i) NO_CACHING: perform the full handshaking protocol;

ii) CACHING_LOCAL: re-use local side distribution information (skip Steps 1), but still exchange distribution information with peer side (perform Step 2 to 4);

iii) CACHING_ALL: re-use both local and peer sides' distribution data, so that handshaking is completely avoided.

FlexIO uses the EVPath messaging library [12] to implement its data movement protocols. EVPath provides point-to-point messaging and data marshaling capabilities. Its modular architecture supports multiple messaging transports, and we have added to it the shared memory transport and the RDMA transport required by FlexIO.

### *D. Shared Memory Transport*

The shared memory transport is for intra-node data movement. Using it, small messages like handshaking messages are passed through data queues in shared memory segments. Each data queue is a single-producer, single-consumer, circular, lock-free FIFO queue inspired by Fastforward [17]. The producer and consumer have separate pointers to the next entry to enqueue or dequeue, and these pointers are guaranteed to be placed into different cache lines to reduce cache coherency traffic. Each entry in queue has a payload field of fixed-size and a status flag with two possible states: full or empty. Entries in data queues are carefully aligned and padded to make sure they do not share cache lines, so as to reduce false sharing. During data movement, the consumer polls the flag of the next entry to dequeue. The producer first checks that the next entry to enqueue is marked as "empty" before copying data into it. The flag is then set to "full"; this signals the consumer, which then copies data from the entry into the target receive buffer and sets flag to "empty" to release the entry to the producer. On systems with weak memory consistency, additional memory fences are inserted.

For large messages such as actual simulation output data, a shared memory buffer pool is used. The producer pre-allocates a shared memory buffer pool indexed with a free list. When sending a large message, the producer tries to find a buffer of the closest size in the pool (and allocates one if not found), copies the message into it, sends a control message to the data queue, and returns if it is an asynchronous movement. The consumer extracts the address and length from the control message, copies data from the shared memory buffer into target buffer, and returns the buffer to the producer's free list. Thus two memory copies are needed for sending large messages asynchronously.

On the Cray XK platform, our shared memory transport leverages page mapping support from the XPMEM kernel module [48] to reduce memory copy overheads. During synchronous large message transfers, the producer makes its source buffer available for sharing by calling xpmem_make(), and sends the shared memory segment id through the data queue. The consumer then gets the memory handle, maps the producer's send buffer into its address space, and copies data to the target receive buffer.

### E. RDMA Transport

The RDMA transport in EVPath is for inter-node data movement. It is built on top of Sandia National Laboratory's NNTI library [27]. NNTI implements a uniform set of APIs (including Connect, Memory Register/Unregister, RDMA Put and Get) above ibverbs, Portals, and uGNI. It therefore, provides a portability layer among different interconnects (IB, SeaStar and Gemini). Based on NNTI, the EVPath RDMA transport implements buffer management and several optimizations for high performance RDMA data movement.

Dynamic buffer allocation and memory registration can cause significant overheads in RDMA-based data movement. Figure 4 demonstrates this with a point-to-point RDMA Get bandwidth test on the Cray XK 6. This is particularly an issue for applications generating particle data, since the number of particles written by a simulation process may change across timesteps due to particle movement. One solution to reduce this cost is to use a persistent buffer and registration cache, as in MPI [34] and Charm++ [38]. We use a similar approach: allocated and registered send and receive buffers are temporarily kept in a buffer pool; later data transfers try to reuse those buffers whenever possible. A configurable threshold value controls total memory usage and triggers buffer reclamation, if necessary.

For small messages, a pair of message queues is established between two interacting processes for two way messaging. The sender process uses NNTI's RDMA Put to send a message into the receiver process' message queue. For the Cray Gemini interconnect, this uses FMA Put to send the data. For large message transfers, we use receiver-directed RDMA Get for data movement. The sender process first copies the message into a send buffer acquired from the buffer pool and sends to the receiver a small control message containing the address and size of the send buffer. The receiver prepares a receive buffer, and issues RDMA Get to fetch data according to some scheduling policy. For Gemini, RDMA Get is implemented with uGNI's BTE RDMA operation. The scheduling technique is leveraged from our previous work in data staging [2]; and its use can effectively reduce network contention.

### F. Data Conditioning Plug-ins

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.

DC Plug-ins are stateless codelets created on the reader side (e.g., analytics) 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 are easily programmed with the C subset offered by the C-on-Demand (CoD) [11].

DC Plug-ins are specified as parameters to FlexIO read API calls. Their code strings are compiled and installed in 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 movement. DC Plug-in placement is informed from the caller. Compared to our previous work [2], DC Plug-in has better scalability and is fully integrated with the FlexIO infrastructure; we have also implemented various runtime data manipulation functionality and management policies with DC Plug-ins to further enhance the I/O path (more details in Section IV).

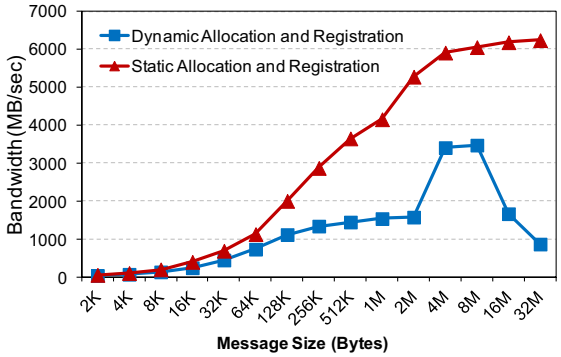



Figure 4. Cost of Dynamic Buffer Allocation and Registration in RDMA Get on Cray XK6 with Gemini Interconnet.

### G. Performance Monitoring

FlexIO monitors the performance of simulation, analytics, and DC Plug-ins. There are measurement points at all levels of the FlexIO software stack to gather a variety of information, including the timing of data movement and DC Plug-in execution, as well as transferred data volumes. Dynamic memory allocation points within FlexIO are also instrumented to record memory usage during data movement. Optionally, 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. For offline performance tuning, monitoring information can be dumped to trace files, and the developer can use it to understand and tune analytics codes. For runtime management, monitoring data captured from the simulation side can be gathered online and transferred to the analytics side. The analytics process(es) can then use it to dynamically schedule data movement and decide the placement of DC Plug-ins.

### H. Implementation Status

FlexIO has been implemented and operates on Cray XT5, XK6,and InfiniBand clusters. Earlier, we applied FlexIO to an online analysis and visualization pipeline for the Pixie3D application on the Cray XT5 [54]. We have also used it to implement analytics for two other applications -- GTS and S3D (details in Section IV). Regarding resiliency, the current version uses simple timeout-and-retry schemes to cope with errors and failures during data movement, but we are planning to incorporate our recent work on a distributed transaction protocol [26] into future version of FlexIO. Features of FlexIO are publically available in the latest release of the ADIOS [18] software.

## III. EXPLOITING PLACEMENT FLEXIBILITY

FlexIO makes it possible to tune analytics placement to improve performance and/or reduce cost: it provides performance information that can aid in placement decision, and it can automatically configure the underlying transport to enforce any placement decision made by users. To illustrate the importance of this, this section describes three placement algorithms realizing different placement policies. Our purpose is not to show 'best' policies, but instead, to show how FlexIO makes it easy to implement alternative methods suitable for different usage scenarios. This is important because there may be frequent changes to analytics codes and to the configurations of I/O pipelines, to support evolving scientific processes. The heuristic algorithms shown find satisfactory placements for large scale simulation and analytics within reasonable time frames. They assume that simulation and analytics exhibit steady runtime behavior so that placements can be statically determined and enforced at job launch time. Such assumption holds for most of the practical use cases we have encountered.

Placement algorithms: 1) optimize some objective (e.g., minimizing total execution time); 2) use a resource allocation policy that determines how much resource to allocate to simulation and analytics components; and 3) carry out a resource binding policy that decides the process/thread to physical resource mapping.

### A. Performance and Cost Objectives

The following performance and cost metrics (initially defined in [52]) are of interest to science end users.

*Total Execution Time:* the time from the start of simulation and analytics to the completion of both.

*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 CPU hours consumed by their jobs.

*Data Movement Volume:* the amount of data moved between simulation and analytics.

### B. Placement Algorithms

*1) Data Aware Mapping.* The data aware mapping algorithm introduced in [51] takes as input a communication matrix recording the data movement volume between simulation processes and analytics processes. It applies graph partitioning to divide simulation and analytics processes into as many groups as the number of nodes, and then assigns each process group to a node with each process mapped to one core. Data aware mapping is essentially a resource binding algorithm, and it tends to place frequently communicating processes from different programs onto the same node.

*2) Holistic Placement.* We extend data aware placement to holistically treat two additional issues: i) to carry out resource allocation, in addition to simply deciding resource bindings, and ii) to also consider the data movements within parallel simulation and analytics programs (e.g., their MPI communications). Termed 'holistic placement', we have experimented with two algorithm variants, for synchronous vs. asynchronous data movement scenarios, respectively.

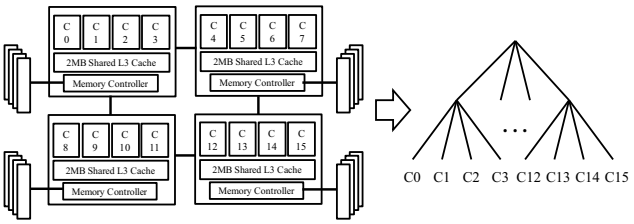



Figure 5. A Multi-Socket NUMA Node Architecture

These algorithms take as input the input configuration of the simulation and the strong scaling function of analytics. Performance profiling is used to obtain such information.

When data movement between simulation and analytics is synchronous, holistic placement works as follows. During resource allocation, the analytics are scaled to match the data generation rate of the simulation. The idea is that simulation and analytics form a two-stage pipeline and hence, matching the analytics' data consumption rate with simulation's data generation rate leads to minimal pipeline stalls. The output of the resource allocation step is the number of processes needed to run analytics.

During resource binding, the algorithm constructs a communication matrix that records both inter- and intra-program data movement. It models the target parallel machine as a two-level tree in which cores of the same node are siblings and have less communication cost with each other than with cores on different nodes. It then uses the graph mapping algorithm provided by the SCOTCH library [36] to map the communication graph to the architecture graph. Compared to data aware mapping, holistic placement 1) captures the trade-off between inter- and intra- program communication, and 2) can be easily extended to model the machine architecture in greater detail (as will be shown in the third algorithm).

When data is moved asynchronously between simulation and analytics, the algorithm additionally considers the asynchrony effect of data movement. Asynchronous data movement overlaps with other activities such as the simulation's computation. Accordingly, unlike the synchronous case, the resource allocation step must ensure that the sum of data movement time and analytics computation time is no larger than the simulation's I/O interval. Data movement time is estimated as total data size divided by point-to-point RDMA transport bandwidth. This estimation is conservative because it assumes data are moved to analytics sequentially (from one simulation process at a time) through the interconnect instead of shared memory, and it may lead to resource over-provisioning for analytics. However, given that analytics usually runs at a much smaller scale than the simulation, such over-provisioning is unlikely to cost significant additional resources and may even be beneficial to accommodate variations in analytics running times. The resource binding step for asynchronous case is the same as for synchronous case described above.

*3) Node Topology Awareness:* To demonstrate the ease with which placement policies can be changed in FlexIO, we explore one additional generalization of the holistic mapping algorithm, designed to take into account the complicated

cache topologies and deep memory hierarchies of modern multi-core processors. Figure 5 shows the memory structure of a machine with four quad-core AMD Barcelona processors and four NUMA domains. Cores share different levels of cache and memory resources, which results in non-uniform on-node communication times between cores.

Node topology aware placement, then, further extends holistic placement by modeling the target machine as a multi-level hierarchy: cores within the same node are placed at different levels of the tree according to the cache topology. The graph mapping algorithm (used in holistic placement as described above) is then applied to map the communication graph onto the hierarchical architecture tree and generate process-to-core binding.

For NUMA machines, the algorithm not only decides process-to-core binding, but also determines the placement of FlexIO's internal buffers in memory. Our default policy is that the shared memory data queues and buffer pools are placed into simulation processes' local NUMA domain no matter where communicating analytics processes are located. This arrangement facilitates the simulation's access to those data structures but may penalize analytics' access. The idea is that in most cases, the simulation is the performance-bounding part in the producer-consumer pipeline, while the analytics are more tolerant of slower data movement.

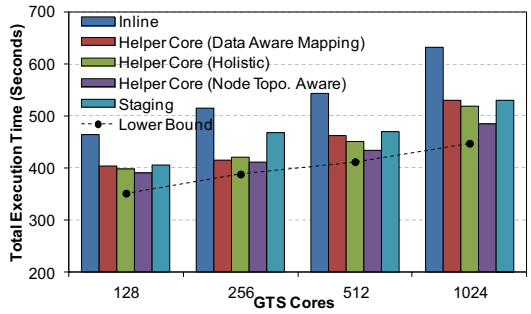
## IV. Performance Evaluation

In this section, we present experimental results obtained from tuning placements of analytics for two large-scale applications: GTS and S3D. We also demonstrate the utility of Data Conditioning Plug-ins to enable dynamic placement of analytics at runtime.

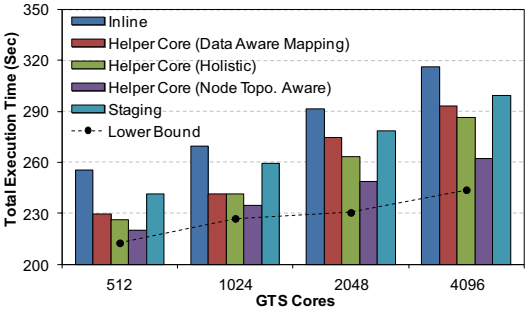
Experiments are run on Oak Ridge National Laboratory's Titan Cray XK6 and Smoky cluster. Titan is upgraded from the Jaguar Cray XT5 and equipped with 18,688 compute nodes, 960 of which contain GPUs. Each compute node has a 16-core 2.2GHz AMD Opteron 6274 (Interlagos) processor and 32GB of RAM. Titan uses the Gemini interconnect. Smoky is an 80 node cluster. Each compute node has four quad-core 2.0GHz AMD Opteron processors (as shown in Figure 5) and 32 GB of memory. The Smoky cluster uses DDR InfiniBand interconnect. Both Titan and Smoky have access to the center-wide Lustre file system.

### A. GTS Performance

GTS (Gyrokinetic Tokamak Simulation) is a global three-dimensional Particle-In-Cell (PIC) code used to study the microturbulence and associated transport in magnetically confined fusion plasma of tokamak torodial devices [47]. GTS simulation outputs particle data containing two 2-dimensional particle arrays for zions and electrons, respectively. The two arrays contain seven attributes for each particle, including coordinates, velocity, weight and particle ID. The particle data 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. 1D and 2D histograms are generated from the query results and written to files which can then be used for




(a) Total Execution Time on Smoky




(b) Total Execution Time on Titan

Figure 6. GTS Performance Tuning on Smoky and Titan.

parallel coordinates visualization. The analytics code uses FlexIO's stream mode to read particles data and follows the process-group-oriented I/O pattern.

We run GTS with a typical production run configuration, which results in particle data output size of 110MB per process. GTS is run in OpenMP/MPI hybrid mode, as suggested by the GTS team. It outputs particle data every two simulation cycles, as desired by scientists.

#### 1) Tuning Placement of Analytics

We use the approaches described in Section III to place analytics for GTS. For resource allocation, we apply the holistic placement policy to decide the number of processes to run analytics so that the data consumption rate matches GTS simulation's I/O frequency. After completing resource allocation, all three placement algorithms leverage inter-process communication volumes to determine process to core binding. Furthermore, since GTS itself can be strong-scaled for a fixed input problem size by varying number of OpenMP threads per MPI process, we decide the placement for each of GTS configurations with different number of OpenMP threads. We compare the resulting performance and cost of different configurations and placements.

Figure 6 (a) shows the Total Execution Time of the coupled GTS simulation and analytics with different placements at various scales on Smoky (weak scaling is applied). At all scales, all three algorithms decide to place analytics on *Helper Cores* in compute nodes (there are still differences among them, as will be explained later). The particular helper core placement found by node topology aware algorithm consistently shows the best performance: GTS is configured to run with 3 OpenMP threads per MPI process, and every 4 MPI processes are placed on each compute node; 4 analytics processes are placed on the

remaining 4 cores of each node (i.e., the Helper Cores); GTS processes pass data to analytics processes through shared memory transport whose internal buffers are pinned in local NUMA domains. Besides, the GTS processes are placed onto nodes so that their 2D grid communication pattern is aligned with the target machine modeled as a 3-level tree.

In comparison, the holistic placement algorithm maps GTS and analytics processes onto nodes in the same way as node topology aware placement. However, since it ignores the NUMA structure of each node, it maps GTS threads and analytics processes linearly to cores within each node. OpenMP threads of some GTS processes are placed across NUMA boundaries, which hurts performance by up to 7.0% on Smoky. The placement found by data aware mapping algorithm has comparable performance as holistic placement. Largely this is because GTS performance is in-sensitive to process placement and hence ignoring its internal communication in placement decision does not cause notable performance penalty. However, data aware mapping is still outperformed by node topology aware placement by up to 9.5% due to its ignorance of NUMA structure.

We also place analytics inline and on a set of separate staging nodes. With inline placement (Case 2 in Figure 7), the GTS processes directly call 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. In comparison, the helper core placement takes 1 core from GTS and offloads analytics onto that core (Case 1 in Figure 7). Such offloading is beneficial for two reasons. On one hand, GTS running with 4 OpenMP threads cannot make full use of all cores within a compute node due to the fact that there are code regions in GTS where only main thread is active. Taking 1 core out of 4 from a GTS process only slows down GTS by 2.7% (as indicated by the increase of simulation "cycle1" and "cycle2" time from Case2 to Case 1). On the other hand, the inline analytics weighs 23.6% of GTS runtime, so offloading analytics to helper core reduces Total Execution Time.

When placing analytics onto separate staging nodes, data are moved to staging nodes through RDMA transport. Compared to the helper core placement, the pitfalls of placing analytics in staging nodes are: 1) huge amounts of particle data are moved through interconnect which consumes more power than on-node movement; 2) asynchronous bulk data movement can interfere with simulation's MPI communication. We have to carefully set the asynchronous data movement scheduling policy to keep the GTS slowdown under 15%.

In terms of CPU hours cost, Inline placement is the worst due to penalty of running non-scalable analytics at large scales. Helper core placement use the same core counts as Inline placement but consumes less CPU hours by finishing faster. Staging placement is worse than helper core placement since it allocates additional nodes but does not achieve better total execution time.

In terms of data movement volume, both inline and helper core placement avoid moving particle data between simulation and analytics through interconnect, while staging placement causes all particle data moved through interconnect. On the other hand, since staging placement maps analytics processes closer to each other than the other two placements, staging placement helps reduce the amount of analytics' internal MPI communication which go through interconnect. Overall, since inter-program data movement is dominant and analytics runs local query to reduce data, helper core and inline placement reduces inter-node data movement by about 90% over staging placement.

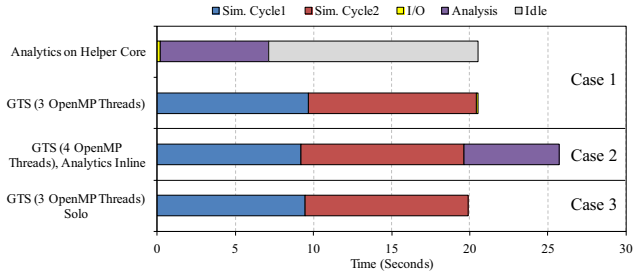



Figure 7. Detailed Timing of GTS and Analytics. GTS runs with 128 MPI processes on Smoky.

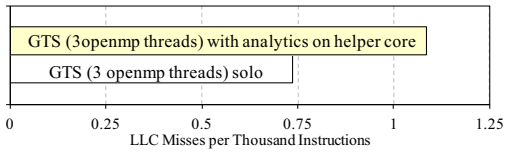



Figure 8. Last Level Cache Miss Rates of GTS on Smoky.

Figure 6 (b) shows placement tuning results on Titan. Similar to Smoky results, on Titan which has 2 NUMA domains and 8 cores in each, running GTS with 7 OpenMP threads per MPI process and analytics on a separate helper core within the NUMA domain results in the best performance and cost.

*2) A Closer Look at Helper Core Placement*

Figure 7 (Case 1) shows that GTS and analytics experience nearly invisible I/O overhead thanks to the shared memory transport. It also shows that analytics processes are idle for 67% of time, indicating over-provisioning for analytics due to our conservative resource allocation policy.

The downside of placing simulation and analytics on the same node is interference between them due to contention on shared on-node resources. To assess such interference, we test two cases: i) GTS with 3 OpenMP threads runs in solo and does no I/O or analytics (Case 3 in Figure 7) vs. ii) GTS with 3 OpenMP threads runs with analytics placed on helper cores (Case 1 in Figure 7). Figure 8 shows the aggregated L3 cache miss rate (measured in L3 cache misses per 1K instructions) seen by all GTS threads in simulation main loop in two cases (hardware performance counters are recorded with PAPI [31]). GTS experiences 47% more L3 cache misses when analytics runs on helper core and share L3 cache with it, and its simulation time ("cycle1" and "cycle2" in Figure 7) increases by 4.1%. Achieving better performance isolation between simulation and analysis when they are placed on the same node is part of our future work.

*3) How Close is Our Solution to the Optimal?*

The runtime of GTS which runs solo with 4 OpenMP threads and does not perform I/O or analytics can be considered as the Total Execution Time when data

movement and analytics are "free" (no resource usage) and infinitely fast. This value is therefore less or equal to the optimal Total Execution Time of coupled simulation and analytics. The best placement solution which we have found is at most 7.9% larger than this lower bound (dashed lines in Figure 6) with the same core count used at all scales on Titan, and at most 8.4% on Smoky.

### B. S3D Performance

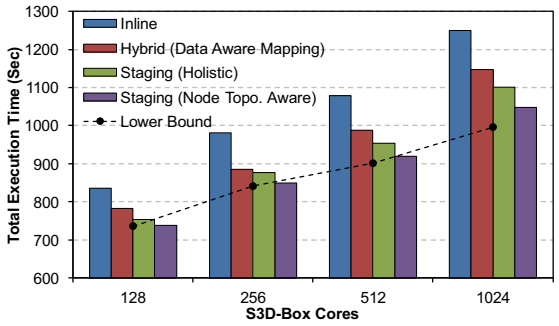
S3D is a state-of-the-art flow solver for performing direct numerical simulation (DNS) of turbulent combustion [20]. We use a modified version of S3D code called S3D_Box created by the S3D team for our test. S3D_Box performs a portion of the full S3D simulation. During its execution, S3D_Box periodically outputs species data which are 22 3-dimensional double-typed arrays. The species data is fed into a parallel volume rendering code [49] to visualize images for each every species. The visualization code reads data by specifying global array index ranges and FlexIO handles MxN data re-distribution underneath. We set the input parameters so that during each I/O action, the total size of 22 arrays generated by each simulation process is 1.7MB, which is the same as typical production S3D runs. The simulation writes species data out every ten simulation cycles.

#### 1) Tuning Data Movement

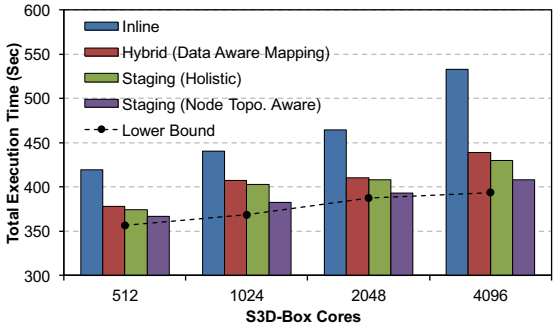
Data movement between simulation and visualization exercises FlexIO's Global-Array-oriented I/O pattern. Since arrays' distribution and memory addresses do not change over time at neither simulation nor visualization side, we set the caching option to CACHE_ALL to avoid several gather/scatter and handshaking messages during data movement (as described in Section II.C). We also enable batching so that all 22 arrays are packed and sent together in a batch. Besides, simulation's write calls are set to be asynchronous. Those tuning efforts significantly reduce the simulation-visible data movement time on both Titan (from 1.2 to 0.053seconds when S3D_Box runs on 1K cores with RDMA transport) and Smoky (from 4.0 to 0.077seconds when S3D_Box runs on 1K cores with RDMA transport). And due to the small output data size, asynchronous data movement does not cause visible impact on simulation's internal communication. The tuning is enforced through setting hints in external XML configuration file and requires no changes to simulation or visualization source code.

#### 2) Tuning Placement of Analytics

We apply the three heuristic algorithms to decide placement of the visualization for S3D_Box. The resource allocation step determines a 128:1 ratio between simulation and analytics processes. For S3D case, the intra-program MPI communication volume is dominant over inter-program data movement due to relative small output data size and low I/O frequency. Under this situation, both holistic placement and node topology aware placement deploy visualization processes onto separate nodes (i.e., Staging Nodes) and use RDMA transport to move data between simulation and visualization processes. They also place S3D_Box in a 3D block decomposed fashion to respect S3D_Box's logical 3D process layout. Node topology aware placement achieves slightly better performance than holistic placement by further




(a) Total Execution Time on Smoky




(b) Total Execution Time on Titan

Figure 9. S3D_Box Performance Tuning.

aligning processes' communication with compute node's NUMA structure (as shown in Figure 9).

The Data Aware Mapping algorithm places each analytics process close to those simulation processes which intensively communicate with it. This ends up placing visualization processes in a hybrid manner: a visualization process receives data from simulation processes both on local node and on remote nodes. Since the inter-program data movement volume is much less than internal MPI communication, putting simulation and visualization close to each other does not pay off sufficiently, but meanwhile such hybrid placement increases the amounts of S3D_Box's MPI communication that goes across interconnect and increase Total Execution Time compared to the staging placement.

We measure the performance with inline placements. Staging placement is better than inline because asynchronous data movement and running simulation and visualization computation (and writing rendered image to files in PPM format) as a two-stage pipeline can effectively hide the cost of I/O and analytics computation. Due to insufficient scalability of file I/O, the advantage of staging placement over inline increase at larger scales. Staging placement also consumes less CPU hours than Inline, since it use 0.78% additional resources but improves Total Execution Time by up to 19% and 30% on Smoky and Titan, respectively.

#### 3) How Close is Our Solution to the Optimal?

The runtime of S3D_Box when it runs solo and does not perform I/O or analysis gives the lower bound of the optimal Total Execution Time (dashed lines in Figure 9). With less than 1% extra resources, the staging placement is at most 3.6% larger than the lower bound on Titan, and 5.1% on Smoky.

In summary, for S3D, placing visualization on a set of staging nodes and aligning both inter- and intra-program data movement with underlying architecture gives the best performance and cost, and the savings of tuning placement is more evident at larger scales.

### C. *Utility of Data Conditioning Plug-ins*

FlexIO's Data Conditioning Plug-in enables dynamic and flexible computation placement along I/O path. Applications can leverage this feature to implement effective runtime policies to customize simulation output data on-the-fly, or adapt to dynamic variations in workloads or environment. We demonstrate the utility of DC Plug-ins with examples.

#### 1) *Dynamic Data Selection*

A common practice for scientists to monitor simulation status is to let simulation periodically output a set of variables and run validation codes on those data. We implement an instance of DC Plug-in for GTS simulation with which validation code can specify the particle attribute(s) it wants to check, and this DC Plug-in can be dynamically deployed and run at either simulation or local analytics side. We use three data selection instances which select 1, 3, and 7 attributes from all 7 attributes of particles, respectively. On Smoky, we run GTS on 256 cores and the validation analytics on separate 32 cores. Figure 10 shows the measured simulation runtime and data movement volume between simulation and analytics when data selection plug-in is deployed at simulation side, and compares the results when all the original particle data are moved to validation analytics ("No Plug-in"). Deploying data selection plug-in with large data reduction ratio onto data source (simulation) can effectively reduce data movement, and cause negligible overhead to simulation blocking I/O time. In fact, reducing data movement volume also improves simulation runtime due to reduced contention on interconnect.

#### 2) *Load Shedding*

If the analytics consumes data slower than simulation generates data, it will blocks simulation and may cause huge waste of CPU cycles at large scale. Under this situation, DC Plug-ins can be used to either shift workload from analytics to simulation or reduce data being moved downstream so that load on analytics side is alleviated. To demonstrate this, we implement a data staging service for GTS which asynchronously moves output data from simulation and dumps data to files. We emulate a situation where the file system is experiencing severe congestion so writing to file is very slow (which does happen in practice) and causes back-pressure to simulation. To cope with this situation, the staging server instantiates a sampling DC Plug-in at simulation side which samples one out of every 100 particles of the original simulation output data. A simple policy is used to trigger load shedding: sampling plug-in is installed to simulation side if monitored simulation's running-average blocking I/O time exceeds a pre-defined threshold value. The dynamic code generation requires only 0.5msecs, so code deployment has an insignificant impact on the running system. The resulting sampling code requires only 220 x86 instructions. Figure 11 compares the steady state time before vs. after DC Plug-in is deployed. The sampling Plug-in helps

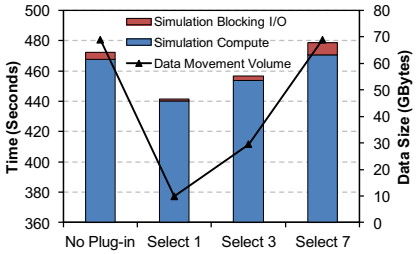



Figure 10. Performance Impact of Data Selection Plug-in to Simulation.

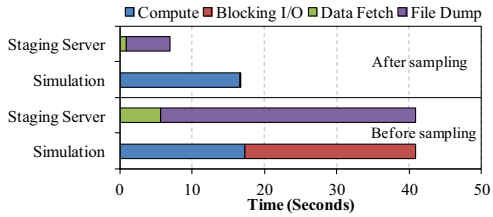



Figure 11. Load Shedding to Adapt to Slow Staging Server. GTS runs on 128 cores and Staging server runs on 16 cores on Smoky.

reduce data fetch time and staging server's file writing time and releases GTS simulation from blocking.

To summarize, experiments show that FlexIO can support a variety of simulation and analytics workloads at large scales through flexible placement options, efficient data movement, and dynamic deployment of useful data manipulation functionalities.

## V. Related Work

Online data analytics and visualization has gained much recent attention from the HPC community. Current work falls into two categories: (1) new data analytics and visualization algorithms, including in situ indexing [45], compression [23], feature extraction [4], and various visualization techniques [41][49], and (2) supporting tools and infrastructures like those mentioned in Introduction.

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 [1] 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 model. [46] studies load distribution of a class of streaming computation in an active storage system. Diamond [19] aggressively places filters to data sources to reduce search operation costs.

The importance of placement has also been exploited in other distributed computing models. Streaming operator placement on wide-area overlay network has been studied in [31]. COLA [21] 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 [30] uses similar graph partitioning techniques to distribute in-network operations within a Data Grid to improve I/O performance. Although the environments targeted by those work are different from the HEC platforms targeted by FlexIO, most placement

algorithms can be supported by FlexIO thanks to its diverse placement options and performance monitoring information.

There have emerged many data intensive computing platforms such as IBM's System S streaming system [15], SciDB [37], and Hadoop/MapReduce-related systems (e.g., SciHadoop [6], Himach [40], and SciMATE [44]). Those are self-contained frameworks with specific programming models and built-in runtime to manage computation distribution. FlexIO as an I/O middleware is beneficial to those frameworks in that they may leverage FlexIO to couple with simulation for online data processing and enjoy the location-flexibility brought by FlexIO.

Scientific workflow systems like Pegasus [9] and Kepler [25] are often used to orchestrate the execution of analysis tasks. They mainly use files as the data exchange mechanism. The explosive growth of scientific data, however, can easily stress the I/O system and overwhelm overall workflow performance. Therefore, it is expected that 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. FlexIO can be readily integrated 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 [17]. There is also similar work on high performance MPI intra-node communication [7][8]. Besides, although MPI may be used to achieve similar flexibility as FlexIO, MPI does not support seamless switch to file I/O, and the code coupling tools built on top of it are shown to have efficiency issues for large data exchange [50].

## VI. Conclusions and Future Work

The FlexIO middleware is designed to flexibly couple data analytics with simulation on high end machines. Evaluation results obtained with two large scale scientific applications GTS and S3D verify the argument for flexible placement and demonstrates FlexIO's ability to support common I/O patterns and diverse placement options. In addition, various placement policies can be implemented with FlexIO to effectively tune application performance and cost. Finally, Data Conditioning Plug-ins enable dynamic deployment of computation along I/O path based on which useful runtime functionalities can be implemented.

Our future work includes: 1) enhancing FlexIO to support dynamic resource allocation and placement policies, to deal with cases where analytics and/or simulations vary their runtime behaviors (e.g., Adaptive Mesh Refinement codes); 2) providing better performance isolation between simulation and analytics for helper core placement scenario; 3) enhancing FlexIO for various failure situations.

## Acknowledgement

We thank Jay Lofstead from Sandia National Laboratory for his helpful comments on this paper. We also thank Ray Grout from National Renewable Energy Laboratory for his help on S3D application. This work was funded by Scientific Data Management Center, U.S. Department of Energy, and Center for Exascale Simulation of Combustion in Turbulence (ExaCT), U.S. Department of Energy. Additional support came from the resources of the National Center for Computational Sciences at Oak Ridge National Laboratory.

## References

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

[2] 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," in Proc. of ACM Symp. on High-Performance Parallel and Distributed Computing (HPDC'11), 2011.

[3] H. Abbasi, M. Wolf, K. Schwan, G. Eisenhauer, and A. Hilton, "Xchange: Coupling Parallel Applications in A Dynamic Environment," in Proc. of IEEE International Conference on Cluster Computing (Cluster'04), 2004.

[4] J. C. Bennett, H. Abbasi, P. Bremer, R. Grout, A. Gyulassy, T. Jin, et al. "Combining In-situ and In-transit Processing to Enable Extreme-scale Scientific Analysis". in Proc. of the 2012 ACM/IEEE Supercomputing (SC 2012), 2012.

[5] J. Biddiscombe, J. Soumagne, G. Oger, D. Guibert, and J. G. Piccinali, "Parallel Computational Steering and Analysis for HPC Applications Using a Paraview Interface and the HDF5 DSM Virtual File Driver," in Proc. of EGPGV, 2011.

[6] J. B. Buck, N. Watkins, J. LeFevre, K. Ioannidou, C. Maltzahn, N. Polyzotis, S. Brandt. "SciHadoop: array-based query processing in Hadoop", In Proc. of 2011 International Conference for Supercomputing (SC '11), 2011.

[7] D. Buntinas, B. Goglin, D. Goodell, G. Mercier, and S. Moreaud, "Cache-Efficient, Intranode, Large-Message MPI Communication with MPICH2-Nemesis," In Proc. of the 2009 International Conference on Parallel Processing (ICPP '09), 2009.

[8] 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,", in Proc. of International Conference on Parallel Processing, (ICPP'08), 2008.

[9] E. Deelman, G. Singh, M. Su, etc. "Pegasus: A Framework for Mapping Complex Scientific Workflow onto Distributed Systems," in Proc. of Journal of Scientific Programming, 2005.

[10] M. Dorier, advised by G. Antoniu, "Damaris - Using Dedicated I/O Cores for Scalable Post-petascale HPC Simulations," In Proc. of International Conference on Supercomputing (ICS'11), 2011.

[11] 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.

[12] G. Eisenhauer, M. Wolf, H. Abbasi, S. Klasky, and K. Schwan, "Event-based Systems: Opportunities and Challenges at Exascale," In Proc. of the 3rd ACM International Conference on Distributed Event-Based Systems (DEBS'09), 2009.

[13] 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," In Proc. of IEEE Symp. on Large-Scale Data Analysis and Visualization (LDAV2011), 2011.

[14] B. Fryxell, K. Olson, P. Ricker, F. X. Timmes, M. Zingale, D. Q. Lamb, et al, "FLASH: An Adaptive Mesh Hydrodynamics Code for Modelling Astrophysical Thermonuclear Flashes," Astrophysical Journal Supplement, 2000, pp 273-334.

[15] B. Gedik, H. Andrade, K. Wu, P. S. Yu, and M. Doo, "SPADE: the system s declarative stream processing engine," In Proc. of the 2008 ACM SIGMOD international conference on Management of data (SIGMOD '08), 2008.

[16] 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," in Proc. of ACM/IEEE Conference on Supercomputing (SC04), 2004.

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

[18] ADIOS, http://www.olcf.ornl.gov/center-projects/adios/, 2012.

[19] L. Huston, R. Sukthankar, R. Wickremesinghe, M. Satyanarayanan, G. Ganger, E. Riedel, A. Ailamaki, "Diamond: A Storage Architecture for Early Discard in Interactive Search," in Proc. of the 3rd USENIX Conference on File and Storage Technologies (FAST'04), 2004.

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

[21] R. Khan, K. Hildrum, S. Parekh, D. Rajan, J. Wolf, K. Wu, H. Andrade, B. Gedik, "Cola: Optimizing Stream Processing Application via Graph Partitioning," In Proc. of the 10th ACM/IFIP/USENIX International Conference on Middleware (Middleware'09), 2009.

[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," Proc. ACM/IEEE Conference on Supercomputing (SC03), 2003.

[23] S. Lakshminarasimhan, N. Shah, S. Ethier, S. Klasky, R. Latham, R. Ross, and N. F. Samatova, "Compressing the incompressible with isabela: In-situ reduction of spatio-temporal data," In Proc. of Euro-Par, 2011.

[24] 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.

[25] 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," Special issue: Workflow in Grid Systems, 2006.

[26] J. F. Lofstead, J. Dayal, R. Oldfield, K. Schwan, "D2T: Doubly Distributed Transactions for High Performance and Distributed Computing," In Proc. of IEEE Cluster Computing Conference, 2012.

[27] R. Oldfield, T. Kordenbrock, J. Lofstead. "Developing Integrated Data Services for Cray Systems with a Gemini Interconnect," Cray User Group Meeting 2012.

[28] J. F. Lofstead, F. Zheng, S. Klasky, and K. Schwan, "Adaptable, Metadata Rick I/O Methods for Portable High Performance I/O," In Proc. of IEEE International Parallel and Distributed Processing Symp (IPDPS'09), 2009.

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

[30] R. Oldfield, and D. Kotz, "Improving Data Access for Computational Grid Application," In Proc. of IEEE International Conference on Cluster Computing (Cluster'10), 2010.

[31] PAPI: Performance Application Programming Interface, http://icl.cs.utk.edu/papi/, 2012.

[32] P. Pietzuch, etc. "Network-aware Operator Placement for Stream-processing Systems," In Proc. of the 22nd International Conference on Data Engineering (ICDE'06), 2006.

[33] 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.

[34] H. Pritchard, I. Gorodetsky, D. Buntinas, "A uGNI-based MPICH2 nemesis network module for the cray XE," Proc. EuroMPI'11, 2011.

[35] A. Singh, P. Balaji, and W. Feng, "GePSeA: A General-Purpose Software Acceleration Framework for Lightweight Task Offloading," In Proc. of the 38th International Conference on Parallel Processing (ICPP), 2009.

[36] SCOTCH library. http://www.labri.fr/perso/pelegrin/scotch/, 2012.

[37] E. Soroush, M. Balazinska, D. Wang. "ArrayStore: a storage manager for complex parallel array processing", In Proc. of the 2011 ACM SIGMOD International Conference on Management of data (SIGMOD '11), 2011.

[38] Y. Sun, G. Zheng, L. V. Kale, T. Jones, R. Olson, "A uGNI-based Asynchronous Message-driven Runtime System for Cray Supercomputers with Gemini Interconnect," Proc. 2012 International Parallel and Distributed Processing Symposium (IPDPS'12), 2012.

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

[40] T. Tu, C. A. Rendleman, D. W. Borhani, R. O. Dror, J. Gullingsrud, M. Ø. Jensen, et al.. "A scalable parallel framework for analyzing terascale molecular dynamics simulation trajectories", In Proc. of the 2008 ACM/IEEE conference on Supercomputing (SC '08), 2008.

[41] 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," In Proc. of ACM/IEEE Conference on Supercomputing (SC06), 2006.

[42] V. Vishwanath, M. Hereld, M. E. Papka, "Toward Simulation-Time Data Analysis and I/O Acceleration on Leadership-Class Systems," In Proc. of IEEE Symp. on Large-Scale Data Analysis and Visualization (LDAV2011), 2011.

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

[44] Y. Wang, W. Jiang, G. Agrawal. "SciMATE: A Novel MapReduce-Like Framework for Multiple Scientific Data Formats", In Proc. of the 12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID '12), 2012.

[45] K. Wu, S. Ahern, E.W. Bethel, J. Chen, H. Childs, E. Cormier-Michel, et al., "FastBit: Interactively Searching Massive Data," Proc. SciDAC, Journal of Physics: Conference Series, 2009.

[46] R. Wick, J. S. Chase, and J. S. Vitter, "Distributed Computing with Load-managed Active Storage," In Proc. of ACM Symp. on High-Performance Parallel and Distributed Computing (HPDC'02), 2002.

[47] 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," Physics of Plasmas, 2006, pp 59-64.

[48] Xpmem, http://code.google.com/p/xpmem/.

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

[50] F. Zhang, C. Docan, M. Parashar and S. Kalasky. "Enabling Multi-Physics Coupled Simulations within the PGAS Programming Framework". In Proc. of 11th IEEE/ACM International Symposium on Cluster, Cloud, and Grid Computing (CCGrid 2011), 2011.

[51] F. Zhang, C. Docan, M. Parashar, S. Klasky, N. Podhorszki and H. Abbasi. "Enabling In-situ Execution of Coupled Scientific Workflow on Multi-core Platform". In Proc. of the 26th IEEE International Parallel & Distributed Processing Symposium (IPDPS'12), 2012.

[52] 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," In Proc. of 6th Parallel Data Storage Workshop (PDSW 2011), 2011.

[53] 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", In Proc. of IEEE International Parallel and Distributed Processing Symp (IPDPS'10), 2010.

[54] F. Zheng, J. Cao, J. Dayal, G. Eisenhauer, K. Schwan, M. Wolf, H. Abbasi, S. Klasky, N. Podhorszki. "High End Scientific Codes with Computational I/O Pipelines: Improving their End-to-End Performance". In Proc. of 2nd International Workshop on Petascale Data Analytics: Challenges and Opportunites (PDAC-11) 2011.