Smart: A MapReduce-Like Framework for In-Situ Scientific Analytics

Yi Wang Gagan Agrawal
Computer Science and Engineering
The Ohio State University,
Columbus, OH 43210
{wayi,agrawal}@cse.ohio-state.edu

Tekin Bicer
Mathematics and Computer Science Division
Argonne National Laboratory,
Lemont, IL 60439
bicer@anl.gov

Wei Jiang
Quantcast Corp.
wjiang@quantcast.com

ABSTRACT

Recent years have witnessed an increasing performance gap between I/O and compute capabilities. In-situ analytics, which can avoid expensive data movement of simulation output data by co-locating both simulation and analytics programs, has lately been shown to be an effective approach to reduce both I/O and storage costs. Developing an efficient *in-situ* implementation involves many challenges, including parallelizing the analysis, performing required data movement or sharing, and allocating appropriate resources for the execution of the analytics program(s). MapReduce has been a widely adopted programming model for parallelizing data analytics. However, despite the popularity of MapReduce, there are several obstacles to applying its API and implementing it for in-situ scientific analytics.

In this paper, we present a novel MapReduce-like framework that supports efficient in-situ scientific analytics. Our system is referred to as in-Situ MApReduce liTe (Smart). Smart can load simulated data directly from memory in each node of a cluster. It leverages a MapReduce-like API to parallelize the analytics, while meeting the strict memory constraints on the analytics code when it is co-located with simulation. Using Smart for in-situ analytics requires only minimal changes to the simulation program itself. Smart can be launched in parallel (OpenMP and/or MPI) code region once each simulation output partition is ready, while the global analytics result can be directly fetched after the parallel code converges. We have developed both space sharing and time sharing modes for maximizing the performance in different scenarios. Moreover, Smart also incorporates an optimization for efficient in-situ window-based analytics.

Performance comparison with Spark shows that our system can achieve high efficiency in in-situ analytics, by outperforming Spark by at least an order of magnitude. We also show that our middleware does not add much overhead (typically less than 10%) compared with handcrafted analytics programs. By using different simulation and analytics programs on both multi-core and many-core clusters, we have demonstrated both the functionality and scalability of our system on nine applications. We can achieve 93% parallel efficiency on average. Next, we show the efficiency of both space sharing and time sharing modes. Finally, we also show that our optimization for in-situ window-based analytics can achieve a speedup of up to 5.6.

1. INTRODUCTION

A major challenge faced by data-driven discovery from scientific simulations is a shift towards architectures where memory and I/O capacities are not keeping pace with the increasing computing power [29]. There are many reasons for this constraint on HPC machines. Most critically, the need for providing high performance in a cost and power effective fashion is driving architectures with two critical bottlenecks — *memory bound* and *data movement costs* [24]. Scientific simulations are being increasing executed on systems with coprocessors and accelerators, including GPUs and the Intel MIC, which have a large number of cores but only a small amount of memory per core. As power considerations are driving both the design and operation of HPC machines, power costs associated with data movement must be avoided.

In response to this unprecedented challenge, *in-situ analytics* [23, 25, 57, 50] emerges as a promising data processing paradigm, and is beginning to be adopted by the HPC community. This approach co-locates the upstream simulations with the downstream analytics on same compute nodes, and hence it can launch analytics as soon as simulated data becomes available. Compared with traditional scientific analytics that processes simulated data offline, in-situ analytics can avoid, either completely or to a very large extent, the expensive data movement of massive simulation output to persistent storage. This translates to saving in execution times, power, and storage costs.

1.1 Positioning and Motivation

The landscape of the current in-situ analytics research mainly falls into two levels: 1) in-situ algorithms at the *application level*, including indexing [22, 25], compression [26, 58], visualization [47, 56, 21], and other analytics [42, 27]; and 2) in-situ resource scheduling platforms at the *system level*, which aims to enhance resource utilization and simplify the management of co-located analytics code [54, 42, 34, 10, 15, 8]. These in-situ middleware systems mainly play the role of a *coordinator*, aiming to facilitate the underlying scheduling tasks, such as cycle stealing [54] and asynchronous I/O [42].

Despite a large volume of recent work in this area, an important question remains almost completely unexplored: "*can the applications be mapped more easily to the platforms for in-situ analytics*?". In other words, we posit that *programming model* research on *in-situ* analytics is needed. Particularly, in-situ algorithms are currently implemented with low-level parallel programming libraries such as MPI, OpenMP, and Pthread, which offer high performance but require that programmers manually handle all the parallelization complexities. As a result, scientists often have to spend plenty of efforts coping with many nitty-gritty and error-prone parallelization details, leading to unpleasant distraction from the core analytics as well as low productivity. Moreover, given that most in-situ algorithms already have the offline implementations, another interesting question can be "*can the offline analytics code be the same as the in-situ analytics code*?". Since the change of processing mode often leads to nontrivial modifications of the low-level implementation of parallelization, if we can decouple the parallelization from the analytics logic, then it is very likely to support processing mode switch base on the same analytics code.

Most in-situ applications perform statistical analysis, feature extraction, data mining, preprocessing, or other closely related tasks. For this class of applications, and from the programmability (and not necessarily performance) view-point, MapReduce is the most widely adopted programming model [13]. Not only the MapReduce API simplifies parallelization of an application, but also MapReduce implementations handle much of scheduling, task management, and data movement. As MapReduce has been gaining great popularity in recent years, many scientists are now well-trained for writing clean MapReduce-style code for scientific analytics [43, 52, 30, 11, 46, 41, 38, 32]. Therefore, an in-situ based MapReduce-like framework can be a promising approach to improve both productivity and maintainability of scientific analytics.

1.2 Our Contributions

In this paper, we describe a novel MapReduce-like framework for in-situ scientific analytics. To the best of our knowledge, this framework is the first work to exploit MapReduce in in-situ scientific analytics, and it is referred to as in-Situ MApReduce liTe (Smart). Our system can support a variety of scientific analytics on simulation nodes, with minimal modification of simulation code and without any specialized deployment (such as installing HDFS). Compared with traditional MapReduce frameworks, Smart can support efficient in-situ processing by accessing simulated data directly from memory in each node of a cluster or a distributed memory parallel machine. Moreover, unlike the traditional implementations, we base our work on a variant of the MapReduce API, which avoids outputting *key-value* pairs and thus keeps the memory consumption of analytics programs low. To address the mismatch between parallel programming view of simulation code and sequential programming view of MapReduce, Smart can be launched in parallel (OpenMP and/or MPI) code region once each simulation output partition is ready, while the global analytics result can be directly fetched after the parallel code converges. Further, we have developed both *space sharing* and *time sharing* modes for maximizing the performance in different scenarios. Additionally, for memory-intensive window-based analytics, we improve the in-situ efficiency by supporting early emission of *reduction object*.

We have extensively evaluated both the functionality and efficiency of our system, by using different real-life simulation and analytics programs on both multi-core and many-core clusters. We first show that our system can outperform Spark [7] by at least an order of magnitude for three applications. Second, we show that our middleware does not add much overhead (typically less than 10%) compared with handcrafted analytics programs. Next, by varying the number of nodes and threads, we demonstrate a good scalability of our system on nine applications. Moreover, by comparing with another implementation of our system that involves an extra copy of simulated data, we show the efficiency of space sharing mode that can avoid extra data copy. Further, we also evaluate the efficacy of time sharing mode that can support concurrent simulation and analytics. Finally, we compare with an implementation that disables early emission of reduction object, and show our optimization for in-situ window-based analytics can achieve a speedup of up to 5.6.

2. BACKGROUND AND CHALLENGES

In this section, we first provide background information on in-situ scientific analytics and MapReduce, and then discuss the challenges of bridging the gap between the two.

2.1 Background: In-Situ Scientific Analytics and MapReduce

As the performance gap between I/O and compute capabilities has been increasing unprecedentedly, *in-situ* data processing has been widely used in a variety of scientific analytics [22, 26, 58, 47, 56, 42, 27]. This emerging data processing paradigm can avoid expensive data movement of simulation output by co-locating both simulation and analytics programs, leading to a significant reduction in both I/O and storage costs. As a case study, we compare the performance of in-situ analytics, against the traditional offline analytics which runs in a store-first-analyze-after manner – first outputs simulated data to disk and then loads the data into analytics programs. We used a real-life simulation program Heat3D [3], as well as k-means clustering as the analytics program, to process 1 TB data on 64 cores in space sharing mode. To vary the amount of computation, we used different number of iterations (before convergence) in the k-means algorithm. Figure 1 shows the total processing times including both simulation and analytics time, as well as the I/O overheads involved by offline analytics. It turns out that even with a moderate amount of computation, in-situ analytics can still outperform offline analytics by up to 10.4x.

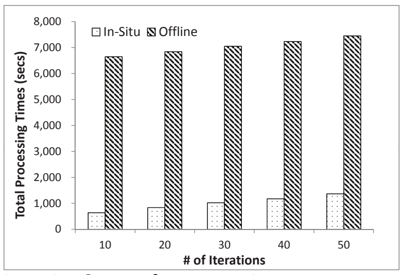


Figure 1: A Case Study: Performance Comparison Between In-Situ and Offline K-Means Clustering

Despite the performance gain compared with traditional offline analytics, to minimize the impact on simulation performance, in-situ analytics requires that some strict resource constraints be met. Particularly, since co-located simulation and analytics share the same memory resource, and simulations are often memory-bound, it is very desirable that the in-situ analytics program operates with only a small amount of memory.

On the other hand, MapReduce [13] was proposed by Google for scalable application development for data-centers. With a simple interface of two functions, *map* and *reduce*, this model has a great suitability for the parallel implementations of a variety of applications, including large-scale scientific analytics [43, 52, 30, 11, 46, 41, 38, 32]. The *map* function takes a set of input instances and generates a set of corresponding intermediate output *(key, value)* pairs. The MapReduce library groups together all of the intermediate values associated with the same key and shuffles them to the *reduce* function. The *reduce* function, also written by the users, accepts a key and a set of values associated with that key. It merges together these values to form a possibly smaller set of values.

As we stated in the previous section, MapReduce or a similar API can greatly simplify in-situ analytics implementations – not only by making specification of parallelism easier, but also by hiding details of data movement, scheduling, and resource allocation. However, there are many challenges in taking the MapReduce idea and implementing it for in-situ scenarios, as we discuss below.

2.2 Challenges: Four Mismatches

The implementation of a MapReduce framework for in-situ scientific analytics imposes several challenges, which we summarize as *four mismatches*.

2.2.1 Data Loading Mismatch

As the name 'in-situ' implies, the downstream analytics program is required to take the input directly from memory rather than from a file system, as soon as the simulated data becomes available. However, existing MapReduce implementations are not designed for such a model. To further elaborate on this data loading mismatch, we first categorize all the MapReduce implementations into four types according to the data loading mechanism.

1. **Loading Data from Distributed File Systems**: A prominent example is Hadoop, as well as its variants like M3R [39] and SciHadoop [11], which load data from Hadoop Distributed File System (HDFS). Moreover, Hadoop actually mimics Google's MapReduce [13], which loads data from the Google File System (GFS). Additionally, Disco [1], a MapReduce implementation in Erlang, loads data from Disco Distributed File System (DDFS).

2. **Loading Data from Shared and/or Local File Systems**: Systems like MARIANE [18] and CGL-MapReduce [17] have adapted MapReduce to scientific analytics environment by loading data from shared file system. Moreover, other MPI-based implementations like MapReduce-MPI [35] and MRO-MPI [33] can load data from shared file system and/or local disk. Note that the aforementioned MapReduce implementations on distributed file systems can still be deployed on shared/local file systems, but installing an additional distributed file system layer is often required.

3. **Loading Data from Memory**: Pthread-based MapReduce prototypes like Phoenix [36], Phoenix++ [40] and MATE [20], can load data from memory. However, these prototypes are restricted to shared-memory environment, and hence currently they are not available for distributed computing.

4. **Loading Data from Data Stream**: To extend MapReduce which is originally designed for batch processing, some systems like HOP [12], M3 [9], and iMR [31] have focused on stream processing.

Clearly, the first two categories loading input from file systems are unable to support in-situ analytics, and the third class lacks native support for global synchronization required in distributed environment. One might think an alternate solution is to wrap up simulated data as a data stream. However, this approach still imposes several obstacles. First, as data is simulated in the form of potentially large time steps, it is simply unnatural to cast time steps into a data stream. Such casting often not only requires nontrivial extra overhead, but also results in periodical stream spiking that can severely degrade the performance of stream processing, due to the sudden arrival of large volumes of simulated data. Second, since only a single pass can be allowed over data stream, such casting also loses the capability of iterative processing, which can be required by many analytics programs, e.g., regression and clustering. Nevertheless, among all the MapReduce implementations that we have examined, we find Spark [49] as an exception here. Its input data layout is defined as Resilient Distributed Dataset (RDD) [48], which can be derived from all the above data source options. However, as will be described later, Spark is still unable to meet other requirements, and we have also extensively experimentally compared our system against Spark in Section 5.

2.2.2 Programming View Mismatch

A critical gap between scientific simulation and MapReduce is caused by different programming views. On one hand, simulations are mostly implemented in MPI in distributed environments, possibly in conjunction of OpenMP on each compute node. Thus, the programmers must explicitly express parallelism in a *parallel programming view*, i.e., all the parallelization details like data partitioning, message passing, and synchronization, must be manually handled. On the other hand, the simplified interface of MapReduce presents a *sequential programming view*, which hides all the parallelization complexities. In such a sequential programming view, all parallelization details are transparent to the programmers. Thus, traditional MapReduce implementations cannot explicitly take partitioned simulation output as the input in parallel, and then launch the execution in a parallel code region. Without any change at the downstream MapReduce side, this mismatch cannot be addressed in a realistic way.

For example, one might consider rewriting all the simulation code based on MapReduce. This option is clearly impractical due to four obstacles. First, scientists not only spend many years on writing, debugging, and tuning existing simulation programs, but those programs also have long lifetimes. Second, simulation programs are mostly written in Fortran or C/C++, and translating them into a programming language like Java or Scala that is used by MapReduce will likely result in a large performance loss. Third, almost all simulation programs require point-to-point data exchange between different partitions, a pattern that does not match MapReduce. Lastly, simulation programs manipulate array slabs and need to be aware of element positional information, whereas conventional MapReduce is designed to process lines of record and does not preserve any record positional information. Yet another possibility might be to gather all partitioned simulated data on a single compute node and feed it to MapReduce. This option is also clearly prohibitively expensive.

An elegant option will be to seek a new MapReduce implementation, which can present a *hybrid programming view*. Particularly, at the beginning, a parallel programming view should be presented, to allow the programmers to be aware of all the partitions during the parallel execution. After the partitioned data are taken by MapReduce, a sequential programming view should follow, to hide all the parallelism and finally present the result in a sequential code region.

2.2.3 Memory Constraint Mismatch

As also mentioned earlier, to minimize the impact on simulation performance, a memory constraint is typically imposed on analytics program. Thus, MapReduce jobs should take up only a very limited amount of memory. However, nearly all existing MapReduce implementations are memory-intensive, and most are even disk-intensive. The main reasons for this are as follows. First, in the mapping phase, each element results in intermediate data in the form of one or more key-value pairs, which often have a greater size than the original element. Second, the subsequent operations like sorting, shuffling, and grouping only reorganize intermediate data, while not reducing its size. Thus, the memory consumption cannot be decreased until the actual reduce operation is executed. Note that although a combiner function at the mapper side can significantly reduce the size of intermediate data in the shuffling phase, it will not help reduce the peak memory consumption in the mapping phase. The memory constraint mismatch cannot be addressed unless we redesign the MapReduce execution flow – particularly, we should avoid the source of intermediate data.

2.2.4 Programming Language Mismatch

The last mismatch is from the programming languages that are used to implement simulation and analytics programs with MapReduce. On one hand, almost all the simulations in use are written in Fortran or C/C++, and it is impractical to rewrite simulation code in other programming languages like Java or Scala. On the other hand, both Hadoop and Spark, which are the most widely adopted MapReduce implementations (although Spark also provides other functionality), cannot natively support Fortran or C/C++. Although this mismatch can be alleviated by using alternate C/C++ based MapReduce implementations [36, 18, 35, 33], these systems are not widely adopted.

3. SYSTEM DESIGN

In this section, we discuss the design and implementation of our system. Overall, Smart design addresses all the challenges we described in the last section, specifically: 1) to address the data loading mismatch, Smart supports processing data from memory, and even no extra data copy is required in space sharing mode; 2) to address the programming view mismatch, Smart presents a hybrid programming view, which can expose each input data partition while launching the data processing, and can still hide parallelism during the execution; 3) to address the memory constraint mismatch, Smart can achieve high memory efficiency with a variant of the MapReduce API, which can avoid massive amounts of intermediate data in the mapping phase and eliminate the shuffling phase; and 4) to address the programming language mismatch, Smart is implemented in C++11, in conjunction with OpenMP and MPI.

3.1 System Overview

Figure 2 gives an overview of the execution flow of a typical application using Smart in a distributed environment. First, given a simulation program, each compute node generates a data partition at each time step. Instead of the data being output to disk, the memory resident data partitions are immediately taken as the input by the downstream Smart analytics job(s). Smart bridges the simulation program by providing a *parallel programming view* – since the data partitions are generated in the parallel code region of simulation programs, the subsequent Smart jobs are also launched in the same parallel code region. Unlike most distributed data processing systems, since the input has already been partitioned by the upstream simulation, Smart can directly expose those partitions to the subsequent processing, rather than involve any explicit data

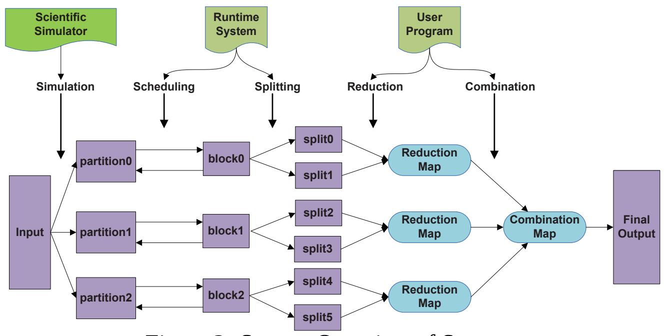


Figure 2: System Overview of Smart

partitioning among the compute nodes. By contrast, all the other MapReduce implementations execute in a sequential code region, take the input as a whole, and partition the data internally by themselves.

Next, the Smart runtime scheduler processes partitioned data block by block. For each data block, the Smart runtime scheduler equally divides it into multiple splits, where each split is assigned to a thread for multi-threading. Additionally, Smart binds each thread to a specific CPU core to maximize the performance.

Afterwards, each thread processes a split by executing two key operations, *reduction* and *combination*, on two core map structures, *reduction map* and *combination map*. To support these operations the programmers need to define a *reduction object*, which acts as the value in the key-value pairs of the two maps. In the reduction process, each data processing unit in the split locates a key in reduction map, and then the data is accumulated on the corresponding reduction object. In the combination process, all the reduction maps are combined into a single combination map locally, and then all the combination maps on each node are further merged on the master node. The key to the high memory efficiency of Smart is the explicit declaration of reduction object, in conjunction of the two maps, which can eliminate the shuffling phase of MapReduce. One can view the sorting executed by conventional MapReduce as a distributed counting sort in Smart.

In analytics program, besides self-defined reduction object, both reduction and combination operations can be customized via a simplified API. The details are discussed in Section 3.4. Since this API does not involve any parallelization detail, the programmers only need to write sequential code, leading to a good programmability like the conventional MapReduce.

Finally, as the parallel code converges, the final output can be retrieved in sequential code region. Thus, a sequential programming view is presented to the user. Alternatively in many cases where the in-situ analytics tasks are deployed as a MapReduce pipeline, some preprocessing steps like smoothing, filtering, and reorganization, only have a local output on each partition. For this case, by turning off the global combination process, the user can retrieve the output directly in the parallel code region, and then feed the output to the next Smart job.

3.2 Two In-Situ Modes

To maximize the performance in different scenarios, our system provides two in-situ modes – *space sharing* and *time sharing*.

Space Sharing Mode: Space sharing mode aims to minimize the memory consumption of analytics, by avoiding extra data copy of simulation output, i.e., time step. Note that although the memory copy itself is often not a costly operation, needing an extra copy of the data can lead to performance degradation. This is because that,

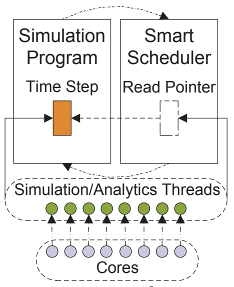


Figure 3: Space Sharing Mode

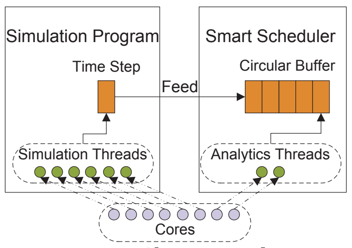


Figure 4: Time Sharing Mode

to maximize the problem size that can be simulated on a platform, simulation programs often only leave very limited unused memory space, which has to be used for in-situ analytics.

As shown in Figure 3, to avoid an extra data copy, Smart sets a *read pointer* on the time step. Thus, time step can be shared by both simulation and analytics programs. Since an early overwrite of time step during simulation may kill the analytics running on it, a constraint of this mode is that, the simulation program cannot output the next time step until the analytics on the previous time step is completed. As a result, in this mode simulation and analytics run in turns, and each makes full use of all the cores of each node.

Time Sharing Mode: In practice, many simulation programs cannot always scale out with an increasing number of cores, due to some nontrivial sequential operations. For these simulation programs, after a certain threshold with regard to the number of cores used, investing more resources may not offer a satisfying speedup. This problem is especially severe for the Intel MIC cluster, since each coprocessor has much more cores than CPU.

To maximize the performance in this scenario, it is more desirable to divide all the cores into two separate groups – one is specifically used for simulation, and the other is dedicated to analytics. We design time sharing mode to meet such need, by supporting concurrent simulation and analytics on two separate groups of cores of each node. In this mode, besides the parallelism of multi-threading as well as the parallelism on multiple nodes, another task-level parallelism is placed on top of these two parallelism levels.

As shown in Figure 4, Smart maintains a *circular buffer* internally, in which each cell can be used for caching a time step. In this mode, one can view simulation program and Smart as producer and consumer, respectively. Once a time step is generated, if the circular buffer is not full, then this time step can be feeded to Smart by copying it to an empty cell. Otherwise, simulation program will be blocked until a cell in circular buffer becomes available. In this example, on an 8-core machine, 6 cores are dedicated to simulation, and 2 cores to analytics. The best division of the two groups of cores is determined by both hardware and specific application run, and currently we find the best division by sample runs.

3.3 Launching Smart and the API Provided by the Runtime

Now we show how to launch Smart in two different modes using System API. Smart is written in C++11, using OpenMP and MPI to achieve parallelism and to also be compatible with a scientific simulation environment. Thus, launching Smart does not require installing additional libraries (e.g., HDFS). As summarized in Table 1, a set of functions are provided by the runtime, which are simply invoked in the application code to initialize the system and run the analytics. These functions are transparent to the programmers. Specifically, functions 1 - 3 in Table 1 are used in both modes, 4 and 5 are used in space sharing mode, and 6 - 8 are used in time sharing mode.

Listing 1: Launch Smart in Space Sharing Mode

```
1 void simulate(Out* out, size_t out_len, const Param&
       p) {
2     /* Each process simulates an output partition of
          data type In and length in_len. */
3     // Launch Smart after simulation in the parallel
          code region.
4     SchedArgs args(num_threads, chunk_size,
          extra_data, num_iters);
5     unique_ptr<Scheduler<In, Out>> smart(new
          DerivedScheduler<In, Out>(args));
6     smart->run(partition, in_len, out, out_len);
7 }
```

Launching Smart in Space Sharing Mode: A distinguishing feature of Smart is the *ease of use*. In space sharing mode, Smart can minimize the modification of the original simulation code with the simplified system API. As demonstrated in Listing 1, to run Smart in this mode, only 3 lines (lines 4 - 6) need to be added when the data is ready. The example code shows the execution of processing a single time step. Note that the definition of reduction object, as well as the derived Smart scheduler class, are implemented in a separate file based on another API set, which does not add any complexity of the original simulation code (see Section 3.4).

After each data partition is simulated given a set of simulation parameters p (line 2), line 4 constructs a scheduler argument $args$, which specifies the number of threads per process $num\_threads$, the size of a unit chunk $chunk\_size$, the extra data for analytics $extra\_data$, and the number of iterations num_iters. To maximize the analytics performance, $num\_threads$ should be equal to the number of threads used for simulation. $chunk\_size$ is the size of processing unit, and it can often be viewed as the length of *feature vector* in analytics applications. $extra\_data$ is used when some additional input is required, e.g., the initial k centroids are required in k-means clustering. num_iters can be specified for iterative processing. By default, $extra\_data$ and num_iters are initialized as a null pointer and 1, respectively. Line 5 constructs a derived Smart scheduler instance $smart$ with the scheduler argument $args$. Note that Smart scheduler class is defined as a template class, and hence Smart can be utilized for taking any array type as input or output, without complicating the application code. In line 6, Smart launches analytics by taking the partitioned data as the input, and the final result will be output to the given destination. During the entire process, all the parallelization details are hidden in a sequential programming view.

Listing 2: Launch Smart in Time Sharing Mode

```
void simulate(Out* out, size_t out_len, const Param&
     p) {
    /* Initialize both simulation and Smart. */
    #pragma omp parallel num_threads(2)
    #pragma omp single
    {
        #pragma omp task  // Simulation task.
        {
            omp_set_num_threads(num_sim_threads);
            for (int i = 0; i < num_steps; ++i) {
                /* Each process simulates an output
                      partition of length in_len. */
                smart->feed(partition, in_len);
            }
        }
        #pragma omp task  // Analytics task.
        for (int i = 0; i < num_steps; ++i)
            smart->run(out, out_len);
    }
}
```

Table 1: Descriptions of the Functions in the System API

Functions Provided by the Runtime	
1) *SchedArgs(int num_threads, size_t chunk_size, const void* extra_data, int num_iters)*	Initializes the Smart scheduler argument by specifying the # of threads, the size of a unit chunk, the extra data, and the # of iterations
2) *explicit Scheduler(const SchedArgs& args)*	Initializes the Smart runtime system
3) *void set_global_combination(bool flag)*	Enable or disable global combination, which is enabled by default
4) *void run(const In* in, size_t in_len, Out* out, size_t out_len)*	Runs the analytics by generating a single key given a unit chunk in space sharing mode
5) *void run2(const In* in, size_t in_len, Out* out, size_t out_len)*	Runs the analytics by generating multiple keys given a unit chunk in space sharing mode
6) *void feed(const In* in, size_t in_len)*	Feeds an input in time sharing mode
7) *void run(Out* out, size_t out_len)*	Runs the analytics by generating a single key given a unit chunk in time sharing mode
8) *void run2(Out* out, size_t out_len)*	Runs the analytics by generating multiple keys given a unit chunk in time sharing mode
Functions Implemented by the User	
1) *virtual int gen_key(const Chunk& chunk) const*	Generates a single key given the unit chunk
2) *virtual void gen_keys(const Chunk& chunk, vector<int>& keys) const*	Generates multiple keys given the unit chunk
3) *virtual void accumulate(const Chunk& chunk, unique_ptr<RedObj>& red_obj) = 0*	Accumulates the unit chunk on a reduction object
4) *virtual void merge(const RedObj& red_obj, unique_ptr<RedObj>& com_obj) = 0*	Merges the first reduction object into the second reduction object, i.e., a combination object
5) *virtual void process_extra_data()*	Processes the extra input data to help initialize the combination map if necessary
6) *virtual void post_combine()*	Performs post-combination processing if necessary
7) *virtual void convert(const RedObj& red_obj, Out* out) const*	Converts a reduction object to an output result if necessary

Launching Smart in Time Sharing Mode: Time sharing mode requires more code reorganization than space sharing mode, since an extra task-level parallelism has to be deployed. Particularly, two OpenMP tasks are created for concurrent execution. After the initialization of both simulation and Smart, one task encapsulates the simulation code and then feeds its output to Smart (lines 6 - 13), and the other task runs analytics (lines 14 - 16). The number of threads used for simulation is specified within the simulation task. and the number of threads used for analytics is specified when Smart is initialized. Note that MPI codes are hidden in both simulation task and analytics task, and in this mode MPI functions may be called concurrently by different threads. Thus, to avoid potential data race, the level of thread support should be upgraded to *MPI_THREAD_MULTIPLE* when MPI environment is initialized.

3.4 Data Processing Mechanism and the API Implemented by the User

Next, we introduce the data processing mechanism, as well as the API potentially to be implemented by the programmers, specific to an application. This set of API is used for implementing the *run* (or *run2*) function in the previous API set, and the same implementation can be used for both in-situ modes, as well as offline processing. This API set mainly includes three functions – *gen_key* or *gen_keys*, *accumulate*, and *merge*. *gen_key* or *gen_keys*, as well as *accumulate* are invoked in the reduction phase, and *merge* is called in the combination phase. Particularly, the *run* function invokes *gen_key* to generate a single key given a unit chunk for most applications, and *run2* function calls *gen_keys* (similar to the *flatmap* function in Scala) to generate multiple keys given a unit chunk for other analytics such as window-based applications [45]. In addition, the programmers need to define a specialized reduction object as a subclass of the interface class *RedObj*.

The *run* function in Algorithm 1 is used in space sharing mode, and it shows the data processing mechanism in Smart. The time sharing mode uses the same mechanism, with a minor difference in the function signature. In the reduction phase, as a data block is divided into multiple splits, each thread processes a data split chunk by chunk. First, for each chunk, a key can be generated by the *gen_key* function. Next, by calling the *accumulate* function, Smart transforms each chunk into a *reduction object*, which serves as the value in the key-value pair of a reduction map. With this reduction object, the corresponding key in the reduction map is updated accumulatively. Thus, no key-value pair with a duplicate key will be generated, and no shuffling phase is needed during the reduction. This is a key difference between our alternate API and the conventional MapReduce paradigm.

The combination phase includes two steps – *local combination* and *global combination*. In the local combination, the reduction maps maintained by all the threads on a process are combined into a local combination map. Particularly, the two reduction objects associated with the same key are merged into one, according to the definition of the *merge* function. In the global combination, the local combination maps on all compute nodes are further combined into a global combination map that holds the global result. This

Algorithm 1: run(const In* in, size_t in_len, Out* out, size_t out_len)

```
 1: process_extra_data()  {* Process the extra data if needed *}
 2: for each iteration $iter$ do
 3:   if $iter > 1$ then
 4:     Distribute the global combination map to each local combination map
 5:   end if
 6:   Distribute the local combination map to each reduction map
 7:   for each processing unit $chunk \in in$ do
 8:     $key \leftarrow$ gen_key($chunk$)
 9:     accumulate($chunk$, $reduction\_map\_[key]$)
10:   end for {* Reduction *}
11:   for each $(key, red\_obj) \in reduction\_map\_$ do
12:     if $key$ exists in $combination\_map\_$ then
13:       merge($red\_obj$, $combination\_map\_[key]$)
14:     else
15:       move $red\_obj$ to $combination\_map\_[key]$
16:     end if
17:   end for {* Local combination and global combination *}
18:   post_combine()  {* Perform post-combine operations if needed *}
19: end for
20: if $out \neq NULL$ and $out\_len > 0$ then
21:   for each $(key, red\_obj) \in combination\_map\_$ do
22:     convert($red\_obj$, $out[key]$)
23:   end for
24: end if {* Output results from the combination map *}
```

global combination leverages the same *merge* function.

Moreover, the *process_extra_data* and *post_combine* functions are often used for the analytics involving iterative processing. Particularly, the *process_extra_data* function can help initialize combination map with the extra input. For example, k-means clustering requires some initial centroids as the extra data besides the input points, and those centroids can be used to initialize the reduction objects that represent clusters. After the combination map is initialized, it is then distributed to each reduction map (lines 3 - 6). After the combination phase, the *post_combine* function can help update reduction objects. For instance, two fields *sum* and *size* in a reduction object can be used to compute the *average* in this function. Additionally, for non-iterative applications, the two functions actually involve no computation by default, leading to an empty initial combination map. Finally, all the reduction objects in the combination map are converted into the desired output, according to the *convert* function.

Additionally, the only difference between the *run* and *run2* functions is in lines 8 and 9. In the *run2* function, given a chunk, multiple keys will be generated, and the chunk will be accumulated in a loop that iterates over all the generated keys.

3.5 Smart Analytics Examples

We now illustrate the use of our system API by creating two example applications, *histogram* and *k-means clustering*, as an instance of non-iterative and iterative application, respectively. Note that the application-specific analytics code is written in a separate file, and it does not differ in different in-situ modes.

Listing 3: Histogram as a Non-Iterative Example Application

```
Derive a reduction object:
struct Bucket : public RedObj {
    size_t count = 0;
};
Derive a system scheduler:
template <class In>
class Histogram : public Scheduler<In, size_t> {
    // Compute the bucket ID as the key.
    int gen_key(const Chunk& chunk) const override {
        // Each chunk has a single element.
        return (chunk[0] - MIN)) / BUCKET_WIDTH;
    }
    // Accumulate chunk on red_obj.
    void accumulate(const Chunk& chunk, unique_ptr<
         RedObj>& red_obj) override {
        if (red_obj == nullptr) red_obj.reset(new
             Bucket);
        red_obj->count++;
    }
    // Merge red_obj into com_obj.
    void merge(const RedObj& red_obj, unique_ptr<
         RedObj>& com_obj) override {
        com_obj->count += red_obj->count;
    }
};
```

As the first example, Listing 3 shows the pseudo code of equi-width histogram construction. To begin with, the user needs to define a derived reduction object class. In this example, the class $Bucket$ represents a histogram bucket, consisting of a single field $count$. Next, a derived system scheduler class $Histogram$ is defined. Note that to facilitate the manipulation on the datasets of different types, in our system the derived class can be defined as either a template class or a class specific to an input and/or output array type. For this kind of non-iterative application, the user only needs to implement three functions in Table 1 – *gen_key*, *accumulate*, and *merge*. First, the *gen_key* function computes the bucket ID based on the element value in the input data $chunk$, and the bucket ID serves as the returned key. For example, if the element value is located within the value range of the second bucket, then 2 will be returned. For simplicity, we assume that the minimum element value can be taken as priori knowledge or be retrieved by an earlier Smart analytics job. Note that in this application, since each element should be examined individually, each $chunk$ as a processing unit only contains a single element. Second, in the reduction phase, the *accumulate* function accumulates $count$ of the bucket that corresponds to the key returned by the gen_key function. Lastly, given two reduction objects, where the first one red_obj is from the reduction map, and the second one com_obj is from the combination map, the *merge* function merges $count$ on com_obj in the combination phase.

Listing 4: K-Means Clustering as an Iterative Example Application

```
Derive a reduction object:
template <class T>
struct ClusterObj<T> : public RedObj {
    T centroid[NUM_DIMS];
    T sum[NUM_DIMS];
    size_t size = 0;
    void update(); // Update centroid by sum and
         size, which are then reset.
};
Derive a system scheduler:
template <class T>
class KMeans : public Scheduler<T, T*> {
    // Compute the ID of the nearest centroid as the
         key.
    int gen_key(const Chunk& chunk) const override {
        /* Let C be the a set of centroids from the
             reduction objects in combination_map_.
             */
        /* Find the centroid c nearest to the point
             represented by chunk from C. */
        /* Return the key associated with c in
             combination_map_. */
    }
    // Accumulate chunk on sum and size of red_obj.
    void accumulate(const Chunk& chunk, unique_ptr<
         RedObj>& red_obj) override {
        red_obj->sum += chunk;  // Vector addition.
        red_obj->size++;
    }
    // Merge red_obj into com_obj on sum and size.
    void merge(const RedObj& red_obj, unique_ptr<
          RedObj>& com_obj) override {
        com_obj->sum += red_obj->sum;  // Vector
              addition.
        com_obj->size += red_obj->size;
    }
    // Process extra_data_ to set up the initial
          centroids in combination_map_.
    void process_extra_data() override {
        /* Transform extra_data_ into a set of
              cluster objects C. */
        /* Load C into combination_map_. */
    }
    // Update the clusters for the next iteration.
    void post_combine}() override {
        for (auto& pair : combination_map_) {
            RedObj* red_obj = pair->second.get();
            red_obj->update();
        }
    }
    // Extract the centroid from red_obj as the
          output.
    void convert(const RedObj& red_obj, T** out)
          const override {
        memcpy(*out, red_obj->centroid, sizeof(T) *
              NUM_DIMS);
    }
};
```

As shown by Listing 4, the second example is k-means clustering, which represents a set of applications involving iterative processing. First of all, the class $ClusterObj$ is defined as a derived reduction object class, indicating a cluster in a multi-dimensional space. In this class, $centroid$, sum and $size$ represent the centroid coordinate, the sum of the distances from each point to the centroid, and the number of points in the cluster, respectively. Next, $KMeans$ is defined as a derived system scheduler class. For this kind of iterative application, usually most virtual functions should be overwritten. First, given a point represented by the input data $chunk$, the *gen_key* function finds the closest centroid and returns the centroid ID as the key. Second, similar to the previous example, the *accumulate* function accumulates the two distributive (or associative and commutative) fields sum and $size$ on the reduction object in reduction map, and the *merge* function accumulates reduction objects in combination map. Next, the *process_extra_data* function initializes the combination map with the extra data that indicates some initial centroids, and the *post_combine* function prepares for the next iteration, by updating all the clusters. Specifically, the centroid coordinates are computed by sum and $size$, which are then reset as zeros. Lastly, the *convert* function extracts the centroid coordinate from each reduction object as an output result.

From the above two examples, we can see that our system API that needs to be implemented by the user, provides a sequential programming view, which only requires some sequential code based on a self-defined reduction object. Thus, like traditional MapReduce framework, our system makes parallelism entirely transparent to the application code.

4. SYSTEM OPTIMIZATION FOR WINDOW-BASED ANALYTICS

This section first describes an optimization for window-based analytics [45], and then discusses the extension on our system design.

4.1 Motivation

In practice, in-situ analytics can be some preprocessing steps like smoothing and density estimation, which execute in a sliding window. A simple example of such window-based analytics is moving average, which replaces each array element with the average of all the elements within a sliding window. A critical challenge in the implementation is the *high memory consumption*, as we will elaborate on the space complexity below.

The space complexity of window-based analytics implemented by using MapReduce is determined by two factors, which are *the maximal number of key-value pairs* and *the size of key-value pair*. Formally, let the input size and window size be N and W, respectively. In terms of the first factor, since each input element corresponds to an output result, there are totally N output results, which are transformed from N key-value pairs after reduction. Moreover, since each element typically appears W times in the sliding window, W key-value pairs are generated by each element. Thus, in a conventional MapReduce implementation, totally $N \times W$ key-value pairs with N distinct keys are generated (at least in the mapping phase). Smart can reduce the maximal number of key-value pairs to N, because each distinct key corresponds to a single reduction object. On the other hand, the size of key-value pair or reduction object is dependent on the specific application, and it is typically varied from $\Theta(1)$ to $\Theta(W)$. For example, since average is *algebraic* and can be computed by sum and count, the size of reduction object for moving average can be only $\Theta(1)$, while median is *holistic* and can only be computed by preserving all elements, the size of reduction object for moving median is $\Theta(W)$. Another example is K nearest neighbor smoother, where the size of reduction object is $\Theta(K), 1 \leq K \leq W$.

Overall, given a window-based application implemented by Smart, the space complexity is $\Theta(N \times R)$, where N and R denote the maximal number of reduction objects and the size of reduction object, respectively. Since N can often be too large to meet the memory constraint of in-situ scenarios, it is very desirable to reduce the space complexity, especially by reducing the maximal number of reduction objects maintained by Smart.

4.2 Optimization: Early Emission of Reduction Object

We develop the optimization based on the following observation. When an array element is completely passed by a sliding window, the corresponding reduction object will no longer be updated. More specifically, although each input data partition is divided into multiple splits for multi-threading, for most elements, all the associated window snapshots are entirely covered by their belonged local split. As a result, most reduction objects finish the update in the (local) reduction phase, and they will not be involved in the subsequent combination phase.

Algorithm 2: reduce(Split $split$)

```
 1: for each data $chunk \in split$ do
 2:     for each key $k$ generated by $chunk$ do
 3:         Let the reduction object $red\_obj$ be $reduction\_map\_[key]$
 4:         accumulate($chunk$, $red\_obj$)
 5:         if $red\_obj.trigger()$ then
 6:             convert($red\_obj$, $out\_[key]$)
 7:             $reduction\_map\_$.erase($key$)
 8:         end if {* Optimization for early emission *}
 9:     end for
10: end for
```

By capturing this observation, we design a mechanism that can support *early emission of reduction object* in the reduction phase, which is in contrast to the original design that holds all the reduction object until the combination phase ends. Our optimization is implemented in two aspects. First, we extend the reduction object class by adding a *trigger function*. This trigger evaluates a self-defined *emission condition*, and determines if the reduction object should be early emitted from the reduction map. By default, the function returns *false*, and hence no early emission is triggered. Second, we extend the implementation of reduce operation, which is an internal step in Smart scheduling. Lines 5 - 7 in Algorithm 2 show the extension. Once a data chunk is accumulated on a reduction object (line 4), the added trigger function evaluates a user-defined emission condition (line 5). If this condition is satisfied, the reduction object will be immediately converted into an output result, and then be erased from the reduction map (lines 6 and 7).

Listing 5: Moving Average as a Window-Based Example Application

```
Derive a reduction object:
struct WinObj : public RedObj {
    double sum = 0;
    size_t count = 0;
    bool trigger() const override {
        return count == WIN_SIZE;
    }
};
Derive a system scheduler:
template <class In>
class MovingAverage : public Scheduler<In, double> {
    // Take all the element positions covered by the
         window as the keys.
    void gen_keys(const Chunk& chunk, vector<int>&
         keys) const override {
        // Each chunk has a single element, which is
             the center of the window.
        for (int i = max(chunk.start_pos - WIN_SIZE /
             2, 0); i <= min(chunk.start_pos +
            WIN_SIZE / 2, total_len_); ++i) {
            keys.emplace_back(i);
        }
    }
    // Accumulate chunk on red_obj.
    void accumulate(const Chunk& chunk, unique_ptr<
         RedObj>& red_obj) override {
        if (red_obj == nullptr) red_obj.reset(new
            WinObj);
        red_obj->sum += chunk[0];
        red_obj->count++;
    }
    // Merge red_obj into com_obj.
    void merge(const RedObj& red_obj, unique_ptr<
         RedObj>& com_obj) override {
        com_obj->sum += red_obj->sum;
        com_obj->count += red_obj->count;
    }
    // Transform red_obj into average as the output.
    void convert(const RedObj& red_obj, double* out)
         const override {
        *out = red_obj->sum / red_obj->count;
    }
};
```

To support such an optimization, the user only needs to overwrite the trigger function when deriving the reduction object class. Listing 5 shows the implementation of moving average as a window-based application example. In this example, the reduction object counts the number of elements covered by a window, and the emission condition can be whether the count equals the window size. This way, the maximal number of reduction objects is reduced from the input size to the window size, leading to a potentially significant improvement of memory efficiency. Note that since each input element contributes to multiple window snapshots, here we use the gen_keys function instead of gen_key in Table 1, to map each element to multiple keys.

It should be noted that, this optimization is not only specific to in-situ window-based analytics, but also can be broadly applied to other applications, even for offline analytics. Take matrix multiplication as an example, the number of elementary multiplications that contribute to a single output element is a fixed number, which is similar to the window size in window-based analytics. Another example is time series, which can be viewed be as a window-based application in time dimension.

5. EXPERIMENTAL RESULTS

In this section, we evaluate both efficiency and scalability of our system on both multi-core and many-core clusters. First, we compare with Spark [7] – a popular MapReduce implementation (while also providing other functionality), which has been shown to outperform Hadoop by up to 100x. Second, we compare with handcrafted analytics programs (as the baseline performance). Third, we evaluate the scalability of Smart by using two real-life simulation programs, Heat3D [3] and Lulesh [5], and a number of different analytics tasks. Next, we evaluate the efficiency of both space sharing and time sharing modes separately. Lastly, we evaluate the effect of the optimization for window-based analytics, by comparing the performance with an implementation without trigger.

5.1 Applications and Environmental Setup

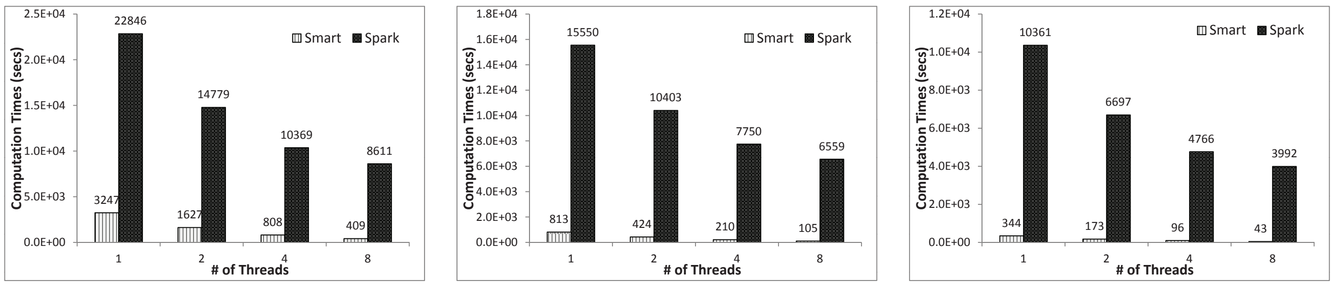
We experimented on totally nine applications that represent six different kinds of in-situ analytics. These applications include: 1) *visualization*: *grid aggregation* [45] groups the elements within a grid into a single element for multi-resolution visualization; 2) *statistical analytics*: *histogram* renders data distribution with equi-width buckets; 3) *similarity analytics*: *mutual information* reflects the similarity or correlation between two variables; 4) *regression analytics*: *logistic regression* measures the relationship between a dependent variable and multiple independent variables; 5) *clustering analytics*: *k-means* tracks the movement of centroids in different time steps; and 6) *window-based analytics*: *moving average* and *moving median* compute average and median in a sliding window, respectively, *Gaussian kernel density estimation* plots data density with Gaussian kernel, and *Savitzky-Golay filter* [37] is often used for rendering absorption peaks in noisy spectrometric data.

The two simulation programs that we chose represent two types of simulations, in terms of the size of the simulation output on each node per time step compared with the memory capacity of simulation machines. Heat3D represents the simulations like GTC [2] and LAMMPS [4], which can generate large data, and Lulesh represents the simulations like miniMD [6] and PDES [44], of which output has a small or medium size.

Our experiments were conducted on two different clusters. We experiment in space sharing mode on a multi-core cluster, and in time sharing mode on a many-core cluster. In the first cluster, the system uses Intel(R) Xeon(R) Processor with 4 dual-core CPUs (8 cores in all). The clock frequency of each core is 2.53 GHz, and the system has a 12 GB main memory. We have used up to 512 cores (64 nodes).In the second cluster, the system uses 8 Intel Xeon Phi SE10P coprocessors, each of which has 61 cores with a clock frequency of 1.1 GHz (488 cores in total). The memory size of coprocessor is 8 GB. Unless specified otherwise, space sharing mode is used in our experiments.

5.2 Performance Comparison with Spark

Although Spark can directly load data from memory and hence can address the data loading mismatch, it cannot overcome the other three mismatches mentioned in Section 2.2. Thus, to make a fair comparison, we let Spark bypass all the other mismatches with the following setup: 1) to bypass the programming view mismatch, we used a sequential program to simulate a double floating-point array with normal distribution, on an 8-core machine (single shared-memory node), 2) to bypass the memory constraint mismatch, the simulation program consumed little memory besides the simulated



(a) Logistic Regression (b) K-Means (c) Histogram

Figure 5: Performance Comparison with Spark

data, and we did not set a memory bound for analytics programs; and 3) to bypass the programming language mismatch, the simulation code was rewritten in Java. 40 GB data was output from the simulation, and the time step size was 0.5 GB. The number of threads used for analytics was varied from 1 to 8. The version of Spark used was 1.1.1.

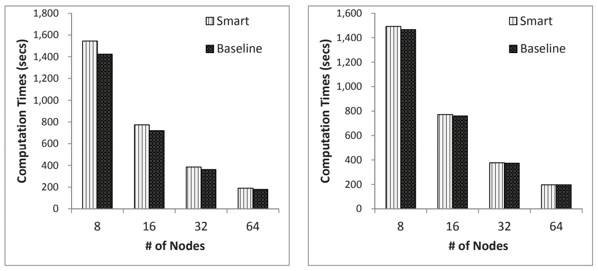
We used three applications for comparison, with the following parameters – 1) *logistic regression*: the number of iterations and the number of dimensions were 10 and 15, respectively; 2) *k-means*: the number of centroids, the number of iterations, and the number of dimensions were 8, 10, and 64, respectively; and 3) *histogram*: 100 buckets were generated. Particularly, both logistic regression and k-means were implemented based on the example codes provided by Spark. Since the simulation code was not parallelized, here we only report the computation times of analytics.

The comparison results are shown in Figure 5. Smart can outperform Spark by up to 21x, 62x, and 92x, on logistic regression, k-means, and histogram, respectively. There can be at least three reasons. First, like other MapReduce implementations, Spark emits massive amounts of intermediate data after the map operation, and grouping is required before reduction. By contrast, Smart performs all reduction in place of reduction maps, avoids storing any key-value pair with a duplicate key, and completely eliminates the need for grouping. Moreover, every Spark transformation operation makes a new RDD (Resilient Distributed Dataset) [48] due to its immutability. In comparison, all Smart operations are carried out on reduction maps and combination maps, and these maps can be reused even for iterative processing. Further, Spark serializes RDDs and send them through network even in local mode, whereas Smart avoids copying any reduction object from reduction map to combination map, by taking advantage of the shared-memory environment within each compute node.

Besides the efficiency advantage, we can also see that Smart scales much better than Spark, at least in the shared-memory environment. Particularly, Smart can achieve a speedup of 7.95, 7.71, and 7.96, by using 8 threads on logistic regression, k-means, and histogram, respectively. This is because that, Spark can only allow the number of worker threads to be controlled by the user, while it still launches extra threads for other tasks, e.g., communication and driver's user interface. Particularly, we can see that, when 8 worker threads were used for Spark execution, the speedup becomes relatively small, because not all 8 cores are being used for computation. By contrast, Smart does not launch any extra threads, and the analytics is efficiently parallelized on all threads.

In addition, Smart can also achieve a much higher memory efficiency than Spark. It turns out that Spark takes up constantly over 90% of the total memory (12 GB) for all the three applications, whereas the memory consumption of Smart is much lower and mainly dominated by the nature of analytics. Take histogram as an example, Smart only consumes 13% of the total memory. Note that the time step size (0.5 GB) is much smaller than the memory capacity, and hence Spark is very unlikely to spill the input to disk.

5.3 Performance Comparison with Handcrafted Analytics Programs



(a) K-Means (b) Logistic Regression

Figure 6: Performance Comparison with Handcrafted Programs

In the second experiment, we compared the performance with handcrafted analytics programs, which were implemented in OpenMP and MPI, and were taken as the baseline performance. We used logistic regression and k-means with the same parameters as in Section 5.2. 1 TB data were processed on a varying number of nodes, ranging from 8 to 64.

Figure 6 shows the results. First, we find that the baseline performance on k-means can outperform Smart by up to 9%. Such performance difference is mainly due to the extra overheads involved in the global combination of Smart. In the baseline implementation, the synchronized data is stored in contiguous arrays, and the global synchronization can be done by a single MPI function call (MPI_Allreduce). By comparison, Smart stores reduction objects in a map structure noncontiguously, and hence an extra serialization of these objects is required by global combination. Second, it turns out that the performance difference on logistic regression is unnoticeable, because only a single key-value pair is maintained in this application and trivial serialization is needed. Overall, our system does not add much overhead compared with the baseline performance. Particularly, since in practice the total processing cost is mostly dominated by simulation, such overhead often does not make a difference in the overall performance.

Note that the container of reduction objects in Smart can be designed as either an array or a map. Our map-based design can have a much better applicability – the keys do not have to be continuous integers, each node can only maintain a subset of keys, and early emission of reduction objects can be supported.

5.4 Scalability Evaluation

The next set of experiments evaluate the scalability of Smart, by using both Heat3D and Lulesh. Totally nine applications were used, with the following parameters – 1) *grid aggregation*: the grid size was 1,000; 2) *histogram*: the number of buckets was 1,200; 3) *mutual information*: the number of buckets for each variable was 100, and hence the 2-dimensional space was divided into up to 10,000 cells; 4) *logistic regression*: the number of iterations and the number of dimensions were 3 and 15, respectively; 5) *k-means*: the number of centroids, the number of iterations, and the number of dimensions were 8, 10, and 4, respectively; and 6) the four window-based applications, including *moving average*, *moving median*, *(Gaussian) kernel density estimation*, as well as *Savitzky-Golay filter*: the window sizes were all 25.

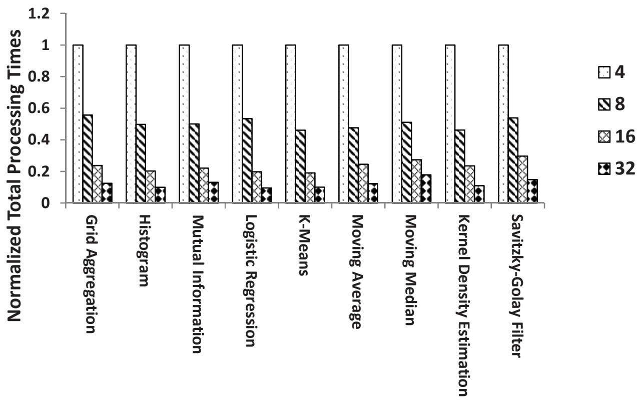


Figure 7: In-Situ Processing Times with Varying # of Nodes on Heat3D (Using 8 Cores per Node)

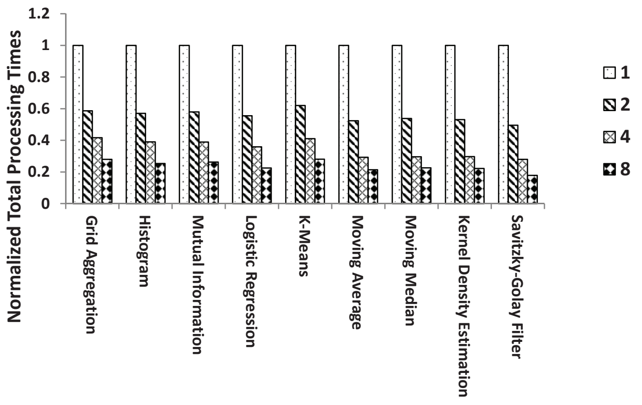
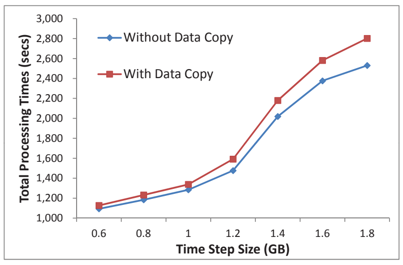


Figure 8: In-Situ Processing Times with Varying # of Threads on Lulesh (Using 64 Nodes)

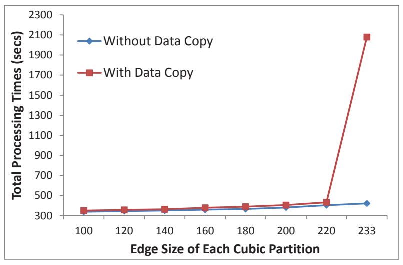
First, we evaluate the total processing times on Heat3D, as we scale the number of compute nodes from 4 to 32, with 8 threads on each node being used for both simulation and analytics. 1 TB data was output by Heat3D over 100 time steps. As Figure 7 shows, Smart can achieve 93% parallel efficiency on average for all the applications. Particularly, we can even see that, for some cases where 16 nodes are used, a super linear scalability can be achieved. Such an extra speedup is caused by the reduction in memory requirements per node as more compute nodes are used.

Second, we evaluate the performance of scaling the number of threads on 64 nodes by using Lulesh. Lulesh output 1 TB data over 93 time steps. The number of threads used for both simulation and analytics per node was up to 8. Figure 8 shows the results. Smart can achieve 59% and 79% parallel efficiency on average for the first five applications, and the other four window-based applications, respectively. The difference in parallel efficiency is related to the nature of these applications.

5.5 Evaluating the Efficiency of Space Sharing Mode



(a) In-Situ Processing Times of Logistic Regression on Heat3D (Using 4 Nodes)



(b) In-Situ Processing Times of Mutual Information on Lulesh (Using 64 Nodes)

Figure 9: Evaluating the Efficiency of Space Sharing Mode

We next demonstrate the advantage of space sharing mode – in-situ processing can be supported even *without involving an extra copy* of the data output from simulation. Recall that in-situ analytics is often required to meet strict memory constraints, this advantage can help speed up in-situ analytics, especially when the setup nearly reaches the memory bound. We evaluate such impact by comparing the performance with an implementation involving data copy.

In this set of experiments, 1 TB data was output by Heat3D on 4 nodes, and by Lulesh on 64 nodes. Logistic regression and mutual information were used as analytics programs on Heat3D and Lulesh, respectively, with the same parameters as for the experiments in Section 5.4. To vary the memory consumption in the simulation program. we varied the time step size for each simulation run as follows. For Heat3D, we could vary the length of one dimension of the 3D problem size, and hence we varied the time step size as well as the memory consumption linearly. Particularly, the time step size was varied from 0.6 to 1.8 GB. For Lulesh, we could vary the edge size of a 3D array cube simulated on each node, and hence by varying the edge size linearly, we could result in a cubic growth of memory consumption. Particularly, the edge size was varied from 100 to 233, and the corresponding time step size was varied from 1.5 to 18.3 GB.

As shown by Figure 9, we can see that when our system does not involve any data copy, there can be a notable speedup. For Heat3D, with a time step greater than 1.4 GB, our system can outperform the other implementation by up to 11%. Note that a time step of

1.8 GB makes the system reach the memory bound in our setup, since a time step of 2 GB can result in a crash. For Lulesh, with an edge size smaller than 220, only a performance gain of up to 7% is achieved. This is because the size of simulated data on each node is only 247 MB, which is very small compared with the memory capacity (12 GB). However, when the edge size reaches 233, the memory consumption of the implementation involving data copy becomes very close to the physical bound, and hence its processing time increases substantially. For this case, our system can achieve a speedup of 5x.

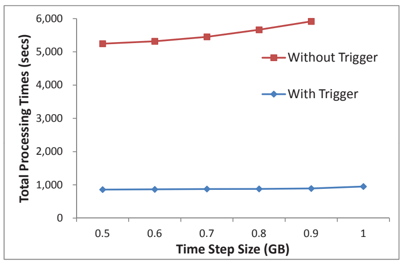
5.6 Evaluating the Efficiency of Time Sharing Mode

Now we evaluate the efficiency of time sharing mode, where both simulation and analytics run concurrently, using two separate groups of cores on each node. This mode can be useful for certain applications, especially when the simulation reaches a scalability bottleneck with the given resources. We evaluate the efficiency of time sharing mode by comparing against the performance in space sharing mode, as well as the performance of pure simulation as a baseline.

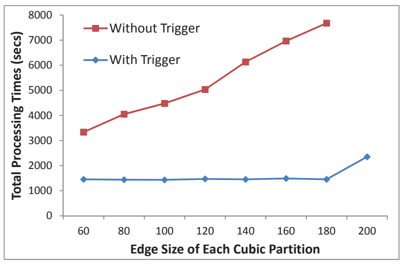
In this set of experiments, 1 TB data was output by Lulesh on 8 MIC nodes. Since it turns out that simulation could not benefit from hyperthreading on the coprocessors, we only used 60 threads for computation in this mode, and spared 1 core for scheduling and communication. Histogram, k-means, and moving median were used as analytics programs, with the same parameters as for the experiments in Section 5.4. Besides space sharing and simulation-only schemes, to vary the number of cores used for simulation and analytics in time sharing mode, we used 5 different schemes, in which the number of cores used for simulation was varied from 50 to 10, and the remaining cores were used for analytics. Three observations can be made. The results are shown in Figure 10, where "50_10" denotes a time sharing scheme with 50 threads for simulation and 10 threads for analytics, and the other time sharing schemes are notated analogically. Three First, although the best performance in time sharing mode is achieved by different schemes for different applications, it does not incur too much overhead compared with the simulation-only performance, even with a moderate amount of computation (as shown in Figure 10(c)). Second, the best performances in time sharing mode for k-means and moving median are achieved by "50_10" and "30_30", and higher than s-pace sharing mode by 10% and 48%, respectively. This is because time sharing mode can make better use of some cores, when simulation reaches its scalability bottleneck. In addition, we also notice that not all applications can benefit from time sharing mode – the best performance of histogram in time sharing mode (achieved by "50_10") is 4.4% lower than space sharing mode. This is because that, the synchronization (or message passing) cost in histogram weighs much more than the other two applications in the total analytics cost, and time sharing mode can only execute the message passing in simulation and analytics sequentially, to avoid the potential data race in MPI. (Only a single thread can call MPI function at a time during concurrent execution.) Thus, we believe time sharing mode does not fit for the applications involving frequent synchronization.

5.7 Evaluating the Optimization for Window-Based Analytics

The last set of experiments evaluate the effect of optimization for window-based analytics, by comparing with an implementation that does not set a trigger function and hence cannot support early emission of reduction object. First, we used Heat3D to simulate 300 GB data, and used moving average as the analytics program on 4 nodes. Similar to the previous experiment, we varied the time step size from 0.5 to 1 GB in Heat3D, and the window size of moving average was 7. Second, 1 TB data was output by Lulesh, and then analyzed by moving median on 64 nodes. We also varied the size of simulated data on each node from 5.2 to 186 MB in Lulesh, by varying the edge size of array cube from 60 to 200, and the window size of moving median was 11.



(a) In-Situ Processing Times of Moving Average on Heat3D (Using 4 Nodes)



(b) In-Situ Processing Times of Moving Median on Lulesh (Using 64 Nodes)

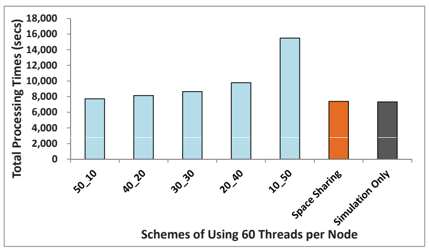
Figure 11: Evaluating the Effect of Optimization for Window-Based Analytics

The results are shown by Figure 11. First, the optimization can lead to a speedup of up to 5.6 and 5.2 in the two experiments, respectively. This is because our optimization with trigger can enable early emission of reduction object, leading to a high memory efficiency. For instance, with such an optimization, it turns out that the maximal number of reduction objects maintained by Smart can be decreased by 1,000,000 times for the case of moving average. Moreover, a time step of 1 GB in Heat3D, or an edge size of 200 in Lulesh, can result in a crash from the implementation without trigger, due to the huge memory consumption. Thus, we do not report the results for these two cases. Overall, we can see that our optimization can significantly improve both compute and memory efficiency of Smart.

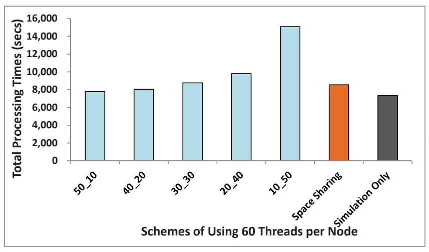
5.8 Discussion

It is worth noting that our system is specifically designed for scientific analytics. Thus, although the *byte stream* data model in conventional MapReduce does not match the array data model prevalent in scientific analytics [11], our system does not have such a mismatch. In our system, the data chunk for unit processing natively preserves array positional information, and hence it can support ad-hoc structural analytics [45], e.g., grid aggregation and moving average. Moreover, in practice many complex analytics can still be can be adapted to nuanced MapReduce pipelines (e.g., mutual information), and our system can still support such application in an in-situ manner.

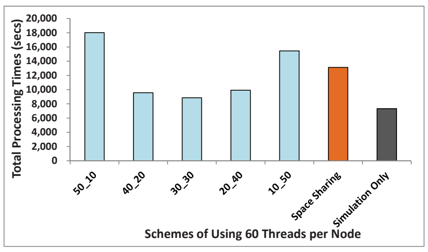
6. RELATED WORK



(a) Histogram



(b) K-Means



(c) Moving Median

Figure 10: Evaluating the Efficiency of Time Sharing Mode

As recent years have witnessed an increasing performance gap between I/O and compute capabilities, in-situ scientific analytics [23, 25, 57, 50] has attracted a lot of attentions in the past few years.

The research on in-situ scientific analytics has been mainly focused on two areas – *applications* and *platforms*. On one hand, in-situ applications and algorithms have been extensively studied, including indexing [22, 25], compression [26, 58], visualization [47, 56, 21], and other analytics like feature extraction [27] and fractal dimension analysis [42]. On the other hand, in-situ resource scheduling platforms can be classified into *space sharing* and *time sharing* two categories. An example of space sharing platform is GoldRush [54] which runs analytics on the same simulation cores. Since simulation and analytics are tightly coupled, cycle stealing becomes critical for performance optimization. For the case of time sharing platforms, CPU utilization of simulation and analytics are decoupled, while the memory bound on analytics still holds. Examples include Functional Partitioning [28], Damaris [16] and CoDS [51]. By contrast, our work explores the opportunities in in-situ scientific analytics at the *programming model* level, which is orthogonal to the dimensions of both in-situ algorithms and platforms. Broadly, in-situ applications can benefit from Smart by adapting the system API and abstracting parallelization, while Smart can be deployed on top of any in-situ resource scheduling platform.

In a broader context of online resource scheduling platforms, another two processing modes have been studied in addition to in-situ processing. One is *in-transit* processing. By leveraging extra resources, this mode moves online analytics to dedicated *staging nodes* that are different from the nodes where simulation runs. Platforms including PreDatA [53], GLEAN [42], JITStager [8], and NESSIE [34], are mainly designed for in-transit processing. Moreover, in-situ and in-transit modes can complement each other. Thus, another mode is *hybrid* processing, which combines in-situ and in-transit modes. Such combination can be supported on the platforms including ActiveSpaces [14], DataSpace [15], FlexIO [55], and others [10]. Our system can be incorporated into these platforms to support in-transit or hybrid processing.

Among existing MapReduce implementations, iMR [31] is specifically designed for in-situ log stream processing. To meet the in-situ resource constraints, iMR focuses on lossy processing and load shedding, while Smart uses a nuanced MapReduce-like API. Further, integrating MapReduce with scientific analytics has been a topic of much interest recently [43, 52, 30, 11, 46, 41, 38, 32]. SciHadoop [11] integrates Hadoop with NetCDF library support to allow processing of NetCDF data with MapReduce API. SciMATE [46] is a MapReduce variant that can transparently process scientific data in multiple scientific formats. Zhao *et al.* [52] implement a parallel storage and access method for NetCDF data based on MapReduce. The Kepler+Hadoop project [43] integrates MapReduce with Kepler, which is a scientific workflow platform. Himach [41] extends MapReduce to support molecular dynamics trajectory data analysis. MARP [38] and KMR [32] are other two MapReduce-based frameworks that can support scientific analytics in MPI environment. SciHive [19] can support querying scientific data based on MapReduce. By contrast, Smart is designed for in-situ processing, and hence it focuses on addressing other challenges such as data loading mismatch and memory constraint mismatch. Moreover, Smart is not bound to any specific scientific data format, since its input is considered to be resident in (distributed) memory.

7. CONCLUSIONS

In this paper, we present a novel MapReduce-like framework, in-Situ MApReduce liTe (Smart), which can support efficient in-situ scientific analytics. Smart can load simulated data directly from memory in distributed environments, and it can also leverage a nuanced MapReduce-like API to meet strict memory constraints during simulation. Performance comparisons with Spark and handcrafted implementations show that our system can achieve high efficiency in in-situ analytics. We have demonstrated both the functionality and scalability of our system by running different real-life simulation and analytics programs on up to 512 cores. Finally, we also show the efficiency of different in-situ modes and the optimization for window-based analytics. Smart is an open-source software, and the source code can be accessed at `https://github.com/SciPioneer/Smart.git`.

8. REFERENCES

[1] Disco Project. http://discoproject.org/.

[2] GTC. http://phoenix.ps.uci.edu/GTC/.

[3] Heat3D. http://dournac.org/info/parallel_heat3d.

[4] LAMMPS. http://lammps.sandia.gov/.

[5] LULESH. https://codesign.llnl.gov/lulesh.php.

[6] MiniMD. https://github.com/akohlmey/miniMD-omp.

[7] Spark. https://spark.apache.org/.

[8] H. Abbasi, G. Eisenhauer, M. Wolf, K. Schwan, and S. Klasky. Just in time: adding value to the IO pipelines of high performance applications with JITStaging. In *HPDC*, pages 27–36. ACM, 2011.

[9] A. M. Aly, A. Sallam, B. M. Gnanasekaran, L. Nguyen-Dinh, W. G. Aref, M. Ouzzani, and A. Ghafoor. M3: Stream processing on main-memory mapreduce. In *ICDE*, pages 1253–1256. IEEE, 2012.

[10] D. A. Boyuka, S. Lakshminarasimham, X. Zou, Z. Gong, J. Jenkins, E. R. Schendel, N. Podhorszki, Q. Liu, S. Klasky, and N. F. Samatova. Transparent In Situ Data Transformations in ADIOS. In *CCGRID*, pages 256–266. IEEE, 2014.

[11] J. B. Buck, N. Watkins, J. LeFevre, K. Ioannidou, C. Maltzahn, N. Polyzotis, and S. Brandt. SciHadoop: Array-based Query Processing in Hadoop. In *SC*, 2011.

[12] T. Condie, N. Conway, P. Alvaro, J. M. Hellerstein, K. Elmeleegy, and R. Sears. MapReduce Online. In *NSDI*, volume 10, page 20, 2010.

[13] J. Dean and S. Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. In *OSDI*, pages 137–150, 2004.

[14] C. Docan, M. Parashar, J. Cummings, and S. Klasky. Moving the code to the data-dynamic code deployment using activespaces. In *IPDPS*, pages 758–769. IEEE, 2011.

[15] C. Docan, M. Parashar, and S. Klasky. DataSpaces: an interaction and coordination framework for coupled simulation workflows. *Cluster Computing*, 15(2):163–181, 2012.

[16] M. Dorier. Src: Damaris-using dedicated i/o cores for scalable post-petascale hpc simulations. In *ICS*, pages 370–370. ACM, 2011.

[17] J. Ekanayake, S. Pallickara, and G. Fox. Mapreduce for data intensive scientific analyses. In *eScience*, pages 277–284. IEEE, 2008.

[18] Z. Fadika, E. Dede, M. Govindaraju, and L. Ramakrishnan. Mariane: Mapreduce implementation adapted for hpc environments. In *GRID*, pages 82–89. IEEE, 2011.

[19] Y. Geng, X. Huang, M. Zhu, H. Ruan, and G. Yang. SciHive: Array-based query processing with HiveQL. In *TrustCom*, pages 887–894. IEEE, 2013.

[20] W. Jiang, V. T. Ravi, and G. Agrawal. A Map-Reduce System with an Alternate API for Multi-core Environments. In *CCGRID*, pages 84–93, 2010.

[21] H. Karimabadi, B. Loring, P. O'Leary, A. Majumdar, M. Tatineni, and B. Geveci. In-situ visualization for global hybrid simulations. In *XSEDE*, page 57. ACM, 2013.

[22] J. Kim, H. Abbasi, L. Chacon, C. Docan, S. Klasky, Q. Liu, N. Podhorszki, A. Shoshani, and K. Wu. Parallel in situ indexing for data-intensive computing. In *LDAV*, pages 65–72. IEEE, 2011.

[23] S. Klasky, H. Abbasi, J. Logan, M. Parashar, K. Schwan, A. Shoshani, M. Wolf, S. Ahern, I. Altintas, W. Bethel, et al. In situ data processing for extreme-scale computing. *SciDAC*, 2011.

[24] P. M. Kogge and T. J. Dysart. Using the top500 to trace and project technology and architecture trends. In *SC*, page 28. ACM, 2011.

[25] S. Lakshminarasimhan, D. A. Boyuka, S. V. Pendse, X. Zou, J. Jenkins, V. Vishwanath, M. E. Papka, and N. F. Samatova. Scalable in situ scientific data encoding for analytical query processing. In *HPDC*, pages 1–12. ACM, 2013.

[26] 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 *Euro-Par*, pages 366–379. Springer, 2011.

[27] A. G. Landge, V. Pascucci, A. Gyulassy, J. C. Bennett, H. Kolla, J. Chen, and P.-T. Bremer. In-situ feature extraction of large scale combustion simulations using segmented merge trees. In *SC*, pages 1020–1031. IEEE, 2014.

[28] M. Li, 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. In *SC*, pages 1–12. IEEE, 2010.

[29] Q. Liu, J. Logan, Y. Tian, H. Abbasi, N. Podhorszki, J. Y. Choi, S. Klasky, R. Tchoua, J. Lofstead, R. Oldfield, et al. Hello ADIOS: the challenges and lessons of developing leadership class I/O frameworks. *Concurrency and Computation: Practice and Experience*, 26(7):1453–1473, 2014.

[30] S. Loebman, D. Nunley, Y.-C. Kwon, B. Howe, M. Balazinska, and J. P. Gardner. Analyzing massive astrophysical datasets: Can Pig/Hadoop or a relational DBMS help? In *CLUSTER*, pages 1–10. IEEE, 2009.

[31] D. Logothetis, C. Trezzo, K. C. Webb, and K. Yocum. In-situ MapReduce for log processing. In *USENIX ATC*, page 115, 2011.

[32] M. Matsuda, N. Maruyama, and S. Takizawa. K MapReduce: A scalable tool for data-processing and search/ensemble applications on large-scale supercomputers. In *CLUSTER*, pages 1–8. IEEE, 2013.

[33] H. Mohamed and S. Marchand-Maillet. MRO-MPI: MapReduce overlapping using MPI and an optimized data exchange policy. *Parallel Computing*, 39(12):851–866, 2013.

[34] R. A. Oldfield, G. D. Sjaardema, G. F. Lofstead II, and T. Kordenbrock. Trilinos i/o support (trios). *Scientific Programming*, 20(2):181–196, 2012.

[35] S. J. Plimpton and K. D. Devine. MapReduce in MPI for large-scale graph algorithms. *Parallel Computing*, 37(9):610–632, 2011.

[36] C. Ranger, R. Raghuraman, A. Penmetsa, G. Bradski, and C. Kozyrakis. Evaluating mapreduce for multi-core and multiprocessor systems. In *HPCA*, pages 13–24. IEEE, 2007.

[37] R. W. Schafer. What is a Savitzky-Golay filter?[lecture notes]. *Signal Processing Magazine, IEEE*, 28(4):111–117, 2011.

[38] S. Sehrish, G. Mackey, J. Wang, and J. Bent. MRAP: A Novel MapReduce-based Framework to Support HPC Analytics Applications with Access Patterns. In *HPDC*, pages 107–118, 2010.

[39] A. Shinnar, D. Cunningham, V. Saraswat, and B. Herta. M3R: increased performance for in-memory Hadoop jobs. *VLDB*, 5(12):1736–1747, 2012.

[40] J. Talbot, R. M. Yoo, and C. Kozyrakis. Phoenix++: modular MapReduce for shared-memory systems. In *MapReduce'11*, pages 9–16. ACM, 2011.

[41] T. Tu, C. A. Rendleman, D. W. Borhani, R. O. Dror, J. Gullingsrud, M. Jensen, J. L. Klepeis, P. Maragakis, P. Miller, K. A. Stafford, et al. A scalable parallel framework for analyzing terascale molecular dynamics simulation trajectories. In *SC*, pages 1–12. IEEE, 2008.

[42] V. Vishwanath, M. Hereld, and M. E. Papka. Toward simulation-time data analysis and i/o acceleration on leadership-class systems. In *LDAV*, pages 9–14. IEEE, 2011.

[43] J. Wang, D. Crawl, and I. Altintas. Kepler + Hadoop: A General Architecture Facilitating Data-Intensive Applications in Scientific Workflow Systems. In *SC-WORKS*, pages –1–1, 2009.

[44] J. Wang, D. Jagtap, N. Abu-Ghazaleh, and D. Ponomarev. Parallel discrete event simulation for multi-core systems: Analysis and optimization. *IEEE Trans. Parallel Distrib. Syst.*, 25(6):1574–1584, 2014.

[45] Y. Wang, A. Nandi, and G. Agrawal. SAGA: Array Storage as a DB with Support for Structural Aggregations. In *SSDBM*, page 9. ACM, 2014.

[46] Y. Wang, J. Wei, and G. Agrawal. SciMATE: A Novel MapReduce-Like Framework for Multiple Scientific Data Formats. In *CCGRID*, pages 443–450, may 2012.

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

[48] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *NSDI*, pages 2–2. USENIX Association, 2012.

[49] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica. Spark: cluster computing with working sets. In *USENIX*, pages 10–10, 2010.

[50] B. Zhang, T. Estrada, P. Cicotti, and M. Taufer. Enabling in-situ data analysis for large protein-folding trajectory datasets. In *IPDPS*, pages 221–230. IEEE, 2014.

[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 *IPDPS*, pages 1352–1363. IEEE, 2012.

[52] H. Zhao, S. Ai, Z. Lv, and B. Li. Parallel Accessing Massive NetCDF Data Based on MapReduce. In *WISM*, pages 425–431, Berlin, Heidelberg, 2010. Springer-Verlag.

[53] F. Zheng, H. Abbasi, C. Docan, J. Lofstead, Q. Liu, S. Klasky, M. Parashar, N. Podhorszki, K. Schwan, and M. Wolf. PreDatA–preparatory data analytics on peta-scale machines. In *IPDPS*, pages 1–12. IEEE, 2010.

[54] F. Zheng, H. Yu, C. Hantas, M. Wolf, G. Eisenhauer, K. Schwan, H. Abbasi, and S. Klasky. GoldRush: resource efficient in situ scientific data analytics using fine-grained interference aware execution. In *SC*, page 78. ACM, 2013.

[55] F. Zheng, H. Zou, G. Eisenhauer, K. Schwan, M. Wolf, J. Dayal, T.-A. Nguyen, J. Cao, H. Abbasi, S. Klasky, et al. FlexIO: I/O Middleware for Location-Flexible Scientific Data Analytics. In *IPDPS*, pages 320–331. IEEE, 2013.

[56] H. Zou, K. Schwan, M. Slawinska, M. Wolf, G. Eisenhauer, F. Zheng, J. Dayal, J. Logan, Q. Liu, S. Klasky, et al. FlexQuery: An online query system for interactive remote visual data exploration at large scale. In *CLUSTER*, pages 1–8. IEEE, 2013.

[57] H. Zou, Y. Yu, W. Tang, and H.-W. M. Chen. FlexAnalytics: a flexible data analytics framework for big data applications with I/O performance improvement. *Big Data Research*, 1:4–13, 2014.

[58] H. Zou, F. Zheng, M. Wolf, G. Eisenhauer, K. Schwan, H. Abbasi, Q. Liu, N. Podhorszki, and S. Klasky. Quality-Aware Data Management for Large Scale Scientific Applications. In *SCC*, pages 816–820, 2012.