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

Ohio State CSE Technical Report #OSU-CISRC-4/15-TR05, April 2015

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

Gagan Agrawal
Computer Science and Engineering
The Ohio State University, Columbus, OH 43210
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 which 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 from parallel (OpenMP and/or MPI) code region once each simulation output partition is ready, while the global analytics result can be directly obtained after the parallel code converges. We have developed both time sharing and space 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. We can achieve 93% parallel efficiency on average. Next, we show the efficiency of both time sharing and space sharing modes. Finally, we also show that our optimization for in-situ window-based analytics can achieve a speedup of up to 5.6.

Keywords

MapReduce, Scientific Analytics, In-Situ Processing

1. INTRODUCTION

A major challenge faced by data-driven discovery from scientific simulations [47, 54, 55] is a shift towards architectures where memory and I/O capacities are not keeping pace with the increasing computing power [27]. 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* [22]. Scientific simulations are being increasingly 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* [21, 23, 64, 72] has emerged 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 Motivation

The current in-situ analytics research can be very broadly classified into two areas: 1) in-situ algorithms at the *application level*, including indexing [20, 23], compression [24, 73], visualization [19, 61, 71], and other analytics [25, 50, 66]; and 2) in-situ resource scheduling platforms at the *system level*, which aim to enhance resource utilization and simplify the management of co-located analytics code [4,6,11,35,50,69]. These in-situ middleware systems mainly play the role of a *coordinator*, aiming to facilitate the underlying scheduling tasks, such as cycle stealing [69] and asynchronous I/O [50].

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. Moreover, because similar analytics may be applied in both in-situ and offline modes, another interesting question is "*can the offline and in-situ analytics codes be (almost) identical?*". Clearly, this is likely only if the implementation is in a high-level framework, where details like loading and staging data and complexity of parallelization are hidden from the application developer.

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 a high-level MapReduce-like API in in-situ scientific analytics. The system 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 supports 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 from parallel (OpenMP and/or MPI) code region once each simulation output partition is ready, while the global analytics result can be directly obtained after the parallel code converges. Further, we have developed both *time sharing* and *space 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 scientific simulations and analytics tasks on both multi-core and many-core clusters. We first show that our system can outperform Spark [63] 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 analytics programs written with low-level programming libraries (i.e., MPI and OpenMP). Next, by varying the number of nodes and threads, we demonstrate high scalability of our system. Moreover, by comparing with another implementation of our system that involves an extra copy of simulated data, we show the efficiency of our design (for time sharing mode). Further, we also evaluate how our space sharing mode is suitable for clusters with many-core nodes. Finally, we show our optimization for in-situ window-based analytics can achieve a speedup of up to 5.6, by comparing it with an implementation that disables early emission of the reduction object.

2. BACKGROUND AND CHALLENGES

In this section, we first provide background information on in-situ scientific analytics and MapReduce, then argue why the MapReduce API is suitable for in-situ analytics, and finally focus on the challenges in applying the MapReduce idea for efficient in-situ scientific analytics.

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 [20,24,25,50,61,71, 73]. 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 [2], as well as k-means clustering as the analytics program, to process 1 TB data on 64 cores in time 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.

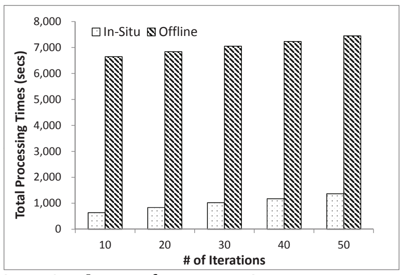


Figure 1: A Case Study: Performance Comparison Between In-Situ and Offline K-Means Clustering

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 [9] 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 [7, 29, 32, 40, 49, 51, 59, 67]. 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.

2.2 Opportunity and Feasibility

MapReduce [9] has been one of most widely adopted programming model for developing data analytics implementations – though it is perhaps not as widely accepted for science areas as it is for commercial areas. MapReduce API not only simplifies parallelization, but the framework implementation handles much of scheduling, task management, and data movement. However, none of its current implementations is directly suitable for in-situ scientific analytics.

We posit that MapReduce API is indeed suitable for a large set of analytics tasks one might perform in-situ on a scientific simulation. Now we give some specific use cases of in-situ analytics reported in various studies, and how these cases can potentially fit the MapReduce paradigm: 1) visualization algorithms [19, 61], where most steps are embarrassingly parallel and others involve reductions; 2) statistical analytics [71] and similarity analytics [44] where statistics like averages, histogram, and mutual information need to be calculated – steps that are very well suited for MapReduce [7], or even higher-level frameworks built on top of MapReduce, e.g., Pig [36] and Hive [48]; and 3) feature analytics [25] and clustering analytics [66], which have been efficiently implemented in MapReduce (though in an offline fashion), e.g., logistic regression and k-means clustering through Spark [63].

Besides the match between the target applications and the choice of programming model, another important issue tends to be that of programmer's expertise. In this respect, we argue that as MapReduce has been gaining great popularity in recent years, many scientists are now well-trained for writing MapReduce-style code for scientific analytics [7, 29, 32, 40, 49, 51, 59, 67]. Therefore, an in-situ MapReduce-like framework can be a promising approach to improve both productivity and maintainability of scientific analytics.

2.3 Challenges

We next discuss the the challenges of bridging the gap between in-situ scientific analytics and MapReduce, which we summarize as *four mismatches*.

2.3.1 Data Loading Mismatch

As the name 'in-situ' implies, the downstream analytics program is required to take the input directly from (distributed) 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 scenario. 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 [43] and SciHadoop [7], which load data from Hadoop Distributed File System (HDFS). Moreover, Hadoop actually mimics Google's MapReduce [9], 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 [14] and CGL-MapReduce [13] have adapted MapReduce to scientific analytics environment by loading data from a shared file system. Moreover, other MPI-based implementations like MapReduce-MPI [37] and MRO-MPI [34] can load data from shared file system and/or local disk.
3. **Loading Data from Memory**: Pthread-based MapReduce prototypes like Phoenix [38], Phoenix++ [46], and MATE [18], 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 a Data Stream**: Though MapReduce was originally designed for batch processing, systems like HOP [8], M3 [5], and iMR [30] have focused on stream processing.

Clearly, the first two categories, where data is loaded from file systems cannot support in-situ analytics. Similarly, the third class lacks native support for global synchronization required in a distributed environment. The fourth group seems more suitable, as one might consider the possibility of wrapping simulation output 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.

However, among all the MapReduce implementations we have examined, we find Spark [63] as an exception here. Its input data layout is defined as Resilient Distributed Dataset (RDD) [62], which can be derived from all the above data source options. However, Spark still has functionality and performance limitations, which will be demonstrated through a series of experiments we report in Section 5.

2.3.2 Programming View Mismatch

A critical gap between scientific simulation and MapReduce is caused by different programming views. On one hand, simulations are usually implemented in MPI (or a PGAS language) that is suitable for distributed memory environments (possibly in conjunction with a shared memory API like OpenMP/OpenCL). With these low-level parallel programming libraries, the programmers 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, or launch the execution of analytics from an SPMD 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 develop 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 input, a sequential programming view should follow, so parallelism details are hidden.

2.3.3 Memory Constraint Mismatch
As simulation programs normally execute with problem sizes that require all or almost all available main memory on each node, the *in-situ* analytics program can only take a very small amount of memory. However, nearly all existing MapReduce implementations are memory-intensive, and most are even disk-intensive. This is primarily because in the mapping phase, each element results in intermediate data in the form of one or more key-value pairs, which can have an even greater size than the original input data. Moreover, 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 need to avoid the intermediate key-value pairs.

2.3.4 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 HPC 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 (though 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 [14, 34, 37, 38], 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 generated by the simulation program – and in one of the in-situ modes (*time sharing*), does so without requiring an extra data copy; 2) to address the programming view mismatch, Smart offers a *hybrid programming view* – this exposes the data partitions to the analytics while launching the data processing, and can still hide parallelism during the execution; 3) to address the memory constraint mismatch, Smart achieves high memory efficiency by modifying the original MapReduce API (while still keeping programming effort very low), and more specifically, avoids the large number of key value pairs or the need for shuffling; and 4) to address the programming language mismatch, Smart is implemented in C++11, in conjunction with OpenMP and MPI.

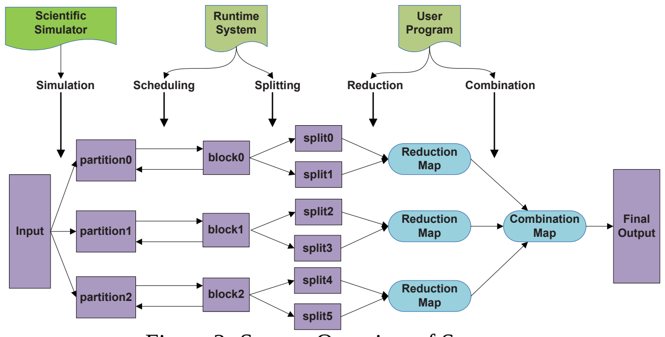
3.1 System Overview


Figure 2: System Overview of Smart

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 the disk, the memory resident data partitions are immediately taken as the input by the downstream Smart analytics job(s). Since the data partitions are generated from the SPMD region of the simulation program, the Smart jobs are also launched from the same code region. Unlike most distributed data processing systems, Smart can directly expose these partitions to the subsequent processing, rather than involve any explicit data partitioning among the compute nodes.

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 processing. Additionally, Smart binds each thread to a specific CPU core to maximize the performance.

In processing elements within a split, there are two key operations, *reduction* and *combination*, which are carried out on two core map structures, *reduction map* and *combination map*, respectively. To support these operations, the programmers need to define a *reduction object*, which represents the data structure of value in the key-value pairs of the two maps. This data structure maintains the accumulated (or reduced) value across all key-value pairs that have the same key. In the reduction operation, a key is first generated for each element in the split. With this key, the runtime next locates a reduction object in the reduction map, and then the corresponding element is accumulated on this 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.

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.

The above execution flow modifies the original MapReduce processing, but it is also the key to the high memory efficiency of S-mart. Specifically, explicit declaration of the reduction object eliminates the shuffling phase of MapReduce. In analytics program, besides self-defined reduction object, both reduction and combination operations can be customized via a simplified API. The specifics of API through examples later in Section 3.4. However, the key point is that besides the declaration of reduction object, the programming effort is not any higher than the one involved in using the original MapReduce API – 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.

3.2 Two In-Situ Modes

To maximize the performance in different scenarios, our system provides two in-situ modes – *time sharing* and *space sharing*. More specifically, we observe that: 1) for certain simulations and/or architectures, memory can be a significant constraint, and we must avoid unnecessary data copying, and 2) in many-core architectures, simulations may not be able to use all available cores effectively, and dedicating a certain number of cores for data analytics can be feasible and desirable. The two situations described above (which may not necessarily be exclusive), lead to the *time sharing* and *s-pace sharing* modes.

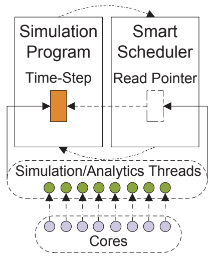


Figure 3: Time Sharing Mode

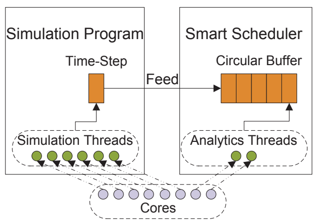


Figure 4: Space Sharing Mode

Time Sharing Mode: Time sharing mode aims to minimize the memory consumption of analytics, by avoiding extra data copy of simulation output. Note that although the memory copy itself is likely not an expensive operation, it can increase the total memory requirements, which can lead to performance degradation in certain cases.

As shown in Figure 3, to avoid an extra data copy, Smart sets a *read pointer* on the memory space corresponding to the output from a particular time-step (when the data is ready). Thus, this data can be now shared by both simulation and analytics programs. However, because this memory space is subject to being overwritten by the simulation program, the analytics logic must execute before the simulation resumes. As a result, in this mode simulation and analytics run in turns, and each makes full use of all the cores of each node (and hence the name time-sharing).

Space Sharing Mode: Consider a cluster where every node is an Intel Xeon Phi. Since each coprocessor has a much larger number of cores than the CPU, a simulation program written for a standard multi-core cluster is unlikely to use all cores of the Xeon Phi effectively. In this case, instead of stopping the progress of simulation periodically and performing the analytics, one can easily dedicate a certain number of the available cores for the analytics. More specifically, all the cores are divided into two separate groups – one is specifically used for simulation, and the other is dedicated to analytics. 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 allocate memory on demand and be used for caching the output from a time-step. In this mode, one can view simulation program and Smart as the producer and the consumer, respectively. Once a time-step's output is generated, if the circular buffer is not full, then this data can be fed to the Smart middleware by copying it to an empty cell. Otherwise, simulation program will be blocked until a cell in circular buffer becomes available.

3.3 Launching Smart and the API Provided by the Runtime

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). Now we show how to launch Smart in two different in-situ analytics modes. 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 - 4 in Table 1 are used in both modes, 5 and 6 are used in time sharing mode, and 7 - 9 are used in space sharing mode.

Listing 1: Launching Smart in Time 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 Time Sharing Mode: A distinguishing feature of Smart is the *ease of use*. In time sharing mode, Smart can minimize the modification of the original simulation code. As demonstrated in Listing 1, to run Smart in this mode, only 3 lines (lines 4 - 6) need to be added to the simulation program.

The example code shows the execution of processing a single time-step. 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 (i.e., unit element) $chunk\_size$, 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. $chunk\_size$

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) $const\ map{<}int,\ unique\_ptr{<}RedObj{>}{>}\&\ get\_combination\_map()\ const$ Retrieves the combination map
5) $void\ run(const\ In*\ in,\ size\_t\ in\_len,\ Out*\ out,\ size\_t\ out\_len)$ Runs the analytics by generating a single key given a unit chunk in time sharing mode
6) $void\ run2(const\ In*\ in,\ size\_t\ in\_len,\ Out*\ out,\ size\_t\ out\_len)$ Runs the analytics by generating multiple keys given a unit chunk in time sharing mode
7) $void\ feed(const\ In*\ in,\ size\_t\ in\_len)$ Feeds an input in space sharing mode
8) $void\ run(Out*\ out,\ size\_t\ out\_len)$ Runs the analytics by generating a single key given a unit chunk in space sharing mode
9) $void\ run2(Out*\ out,\ size\_t\ out\_len)$ Runs the analytics by generating multiple keys given a unit chunk in space sharing mode
Functions Implemented by the User
1) $virtual\ int\ gen\_key(const\ Chunk\&\ chunk,\ const\ In*\ data,\ const\ map{<}int,\ unique\_ptr{<}RedObj{>}{>}\&\ com\_map)\ const$ Generates a single key given the unit chunk (and combination map if necessary)
2) $virtual\ void\ gen\_keys(const\ Chunk\&\ chunk,\ const\ In*\ data,\ vector{<}int{>}\&\ keys,$ const $map{<}int,\ unique\_ptr{<}RedObj{>}{>}\&\ com\_map)\ const$ Generates multiple keys given the unit chunk (and combination map if necessary)
3) $virtual\ void\ accumulate(const\ Chunk\&\ chunk,\ const\ In*\ data,\ unique\_ptr{<}RedObj{>}\&\ red\_obj)\ =\ 0$ 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) $virtual\ void\ process\_extra\_data(const\ void*\ extra\_data,\ map{<}int,\ unique\_ptr{<}RedObj{>}{>}\&\ com\_map)$ Processes the extra input data to help initialize the combination map if necessary
6) $virtual\ void\ post\_combine(map{<}int,\ unique\_ptr{<}RedObj{>}{>}\&\ com\_map)$ Performs post-combination processing and update the combination map if necessary
7) $virtual\ void\ convert(const\ RedObj\&\ red\_obj,\ Out*\ out)\ const$ Converts a reduction object to an output result if necessary

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. 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).

Listing 2: Launching Smart in Space Sharing Mode

```
 1  void simulate(Out* out, size_t out_len, const Param&
         p) {
 2      /* Initialize both simulation and Smart. */
 3      #pragma omp parallel num_threads(2)
 4      #pragma omp single
 5      {
 6          #pragma omp task  // Simulation task.
 7          {
 8              omp_set_num_threads(num_sim_threads);
 9              for (int i = 0; i < num_steps; ++i) {
10                  /* Each process simulates an output
                       partition of length in_len. */
11                  smart->feed(partition, in_len);
12              }
13          }
14          #pragma omp task  // Analytics task.
15          for (int i = 0; i < num_steps; ++i)
16              smart->run(out, out_len);
17      }
18  }
```

Launching Smart in Space Sharing Mode: As shown in Listing 2, space sharing mode requires more code reorganization than time 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 S-mart, 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 the 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 [57]. In addition, the programmers need to define a specialized reduction object as a subclass of the interface class *RedObj*.

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

```
1:  process_extra_data($extra\_data\_$, $combination\_map\_$) {* 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($chunk$, $data\_$, $combination\_map\_$)
9:          accumulate($chunk$, $data\_$, $reduction\_map\_[key]$)
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($combination\_map\_$) {* 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 *}
```

The *run* function in Algorithm 1 is used in time sharing mode, and it shows the data processing mechanism in Smart. The space 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. In line 8, a key is generated for the unit element. Line 9 accumulates the derived data from the element into a *reduction object*, which can be located in the reduction map by the generated key. The reduction object is updated in place – no intermediate key-value pair is emitted or stored, and thus, no shuffling phase is needed during the reduction. This is a key difference between our alternate API and the conventional MapReduce paradigm.

Lines 11 - 17 show the combination phase consisting of 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. 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 global combination leverages the same merge operation used for the local combination. Line 18 can update reduction objects after the combination phase for each iteration, e.g., computing average based on sum and count. Finally, lines 20 - 23 convert all the reduction objects in the global combination map into the desired output.

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 In* data,
        const map<int, unique_ptr<RedObj>>&
        combination_map) const override {
        // Each chunk has a single element.
        return (data[chunk.start] - MIN)) /
            BUCKET_WIDTH;
    }
    // Accumulate chunk on red_obj.
    void accumulate(const Chunk& chunk, const In*
        data, 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. Two major steps are taken. 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 $count$.

In the second step, a derived system scheduler class should be defined, e.g., $Histogram$ here. 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 usually only needs to implement three functions – *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 first bucket, then 0 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 array element. Next 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 T* data, const map<int, unique_ptr<RedObj>>& combination_map) 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, const T* data, 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(const void* extra_data, map<int, unique_ptr<RedObj>>& combination_map) 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}(map<int, unique_ptr<RedObj>>& combination_map) 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. To make use of this function, a restriction is that, the integer key should start from 0.

From the above two examples, we can see that Smart provides a sequential programming view for application development, and the user only needs to write some sequential code based on the declared reduction object. Thus, like traditional MapReduce framework, our system makes parallelism entirely transparent to the application code. Note that unlike a MapReduce job optimized by a combiner function, our application code does not emit any key-value pair as intermediate result.

4. SYSTEM OPTIMIZATION FOR WINDOW-BASED ANALYTICS

4.1 Motivation

In practice, simulation output may contain some short-term volatility or undesired fine-scale structures. In such cases, it is important to perform analytics for specific ranges of time-steps, also referred to as *sliding windows*. In some other cases, in-situ analytics can involve certain preprocessing steps like denoising [17] and smoothing [16, 33], which also execute on a sliding window basis. A simple example of such window-based analytics is *moving average*, where the average of the elements within every window snapshot is computed. A critical challenge in the implementation of such window-based analytics is that of *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. Irrespective of the application, since N can often be too large to meet the memory constraints of in-situ scenarios, it is very desirable to reduce the space complexity, especially by reducing the maximal number of reduction objects.

4.2 Optimization: Early Emission of Reduction Objects

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($chunk$, $data\_$, $red\_obj$)
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
```

We develop the optimization based on the following observation. For most elements, all the associated window snapshots are entirely covered by their respective local split of data. As a result, most reduction object values have been finalized in in the (local) reduction phase, and they will not be involved in the subsequent combination phase. By capturing this observation, we design a mechanism that can support *early emission of reduction objects* in the reduction phase, which is in contrast to the original design that holds all the reduction objects until the combination phase ends.

Our optimization is implemented as follows. 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 emitted early 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 element 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). With such an optimization, the maximal number of reduction objects need to be maintained is reduced from the *input data size* to the *window size*.

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, const In* data,
          vector<int>& keys, const map<int,
         unique_ptr<RedObj>>& combination_map) const
         override {
        // Each chunk has a single element, which is
             the center of the window.
        for (int i = max(chunk.start - WIN_SIZE / 2,
             0); i <= min(chunk.start + WIN_SIZE / 2,
             total_len_); ++i) {
            keys.emplace_back(i);
        }
    }
    // Accumulate chunk on red_obj.
    void accumulate(const Chunk& chunk, const In*
         data, unique_ptr<RedObj>& red_obj) override
         {
        if (red_obj == nullptr) red_obj.reset(new
             WinObj);
        red_obj->sum += data[chunk.start];
        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 is equal to the window size. 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 [**?**, 28, 31, 41, 42, 52, 53, 56, 60]. A simple example can be matrix multiplication, where the number of element-wise multiplications that contribute to a single output element is a fixed number.

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 [63] – a popular MapReduce implementation (while also providing other functionality), which has been shown to outperform Hadoop by up to 100x. Second, we compare with analytics programs written with lower-level APIs (MPI and OpenMP), to measure both the programmability and overheads of our middleware approach. Third, we evaluate the scalability of Smart as the number of nodes and cores is increased. Next, we focus on understanding and comparing performance for time sharing and space sharing modes. Lastly, we evaluate the effect of the optimization for window-based analytics, by comparing the performance with an implementation that disables the trigger mechanism.

5.1 Applications and Environmental Setup

We experimented with nine applications that represent six different classes of in-situ analytics – these classes were previously described as in-situ use cases from the literature in Section 2.2. The classes of analytics and specific applications are: 1) *visualization*: *grid aggregation* [57] 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) *feature 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 [66]; 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 the Gaussian kernel, and *Savitzky-Golay filter* [39] is a well-known smoothing filter.

The above analytics programs can be applied on a variety of simulation programs. However, from a performance view-point, only two aspects of the simulation program are important for us – the memory requirements for the simulation, and relative to it, the amount of data that is either output or needs to be analyzed every time-step. Thus, we choose two open-source simulation programs that have very different amounts of output. Specifically, for every time-step in our experimental setup, Heat3D [2] generates large volumes of data, e.g., 400 MB per node, whereas Lulesh [3] has a moderate amount of output, which is typically smaller than 100 MB on each node.

Our experiments were conducted on two different clusters. The first cluster is a more traditional cluster with multi-core nodes – specifically, each node is an 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 experiment with time sharing mode only on this cluster, as the simulation program can be expected to scale with all available cores. We have used up to 512 cores (64 nodes).The second cluster has a many-core accelerator on each nodes, and both time sharing and space sharing modes are used and compared. Each node on this cluster has an Intel Xeon Phi SE10P coprocessor, with 61 cores and a clock frequency of 1.1 GHz (488 cores in total). The memory size of coprocessor is 8 GB.

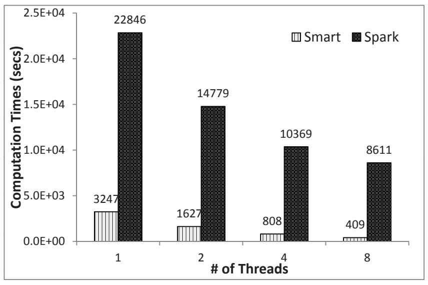
5.2 Performance Comparison with Spark

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 2.3. 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, the simulation program was replaced by a simple emulator – a sequential program that outputs double precision array elements that follow a normal distribution, and in addition, the experiments were only conducted on a single node with 8 cores, 2) the memory constraint mismatch was also addressed by the use of the emulator which hardly consumed any extra memory, and thus there was no tight memory bound for the analytics programs; and 3) to bypass the programming language mismatch, the emulator used by Spark was written in Java. 40 GB data was output from the simulation, over 800 time-steps, and the number of threads used for analytics was varied from 1 to 8. The version of Spark used was 1.1.1.

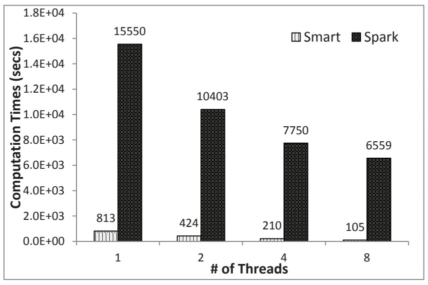
We used three applications for comparison, with the following parameters – 1) *logistic regression*: the number of iterations and the number of dimensions were 10 and 15, respectively; 2) *k-means*: the number of centroids, the number of iterations, and the number of dimensions were 8, 10, and 64, respectively; 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 emulation 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. The reason for such a large performance difference is three-fold. 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 emitting any key-value pairs, and thus, completely eliminates the need for grouping. Moreover, every Spark transformation operation makes a new RDD (Resilient Distributed Dataset) [62] 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.

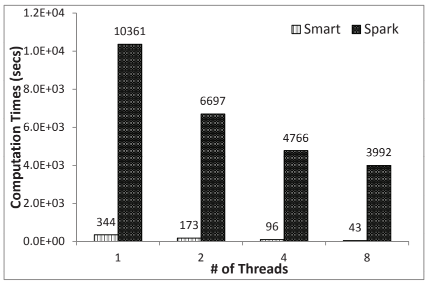
Besides the efficiency advantage, we can also see that Smart scales much better than Spark, at least in the shared-memory environment. Particularly, Smart can achieve a speedup of 7.95, 7.71, and 7.96, by using 8 threads on logistic regression, k-means, and histogram, respectively. This is because that, 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.



(a) Logistic Regression



(b) K-Means

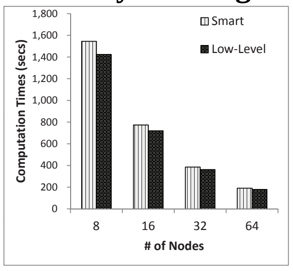


(c) Histogram

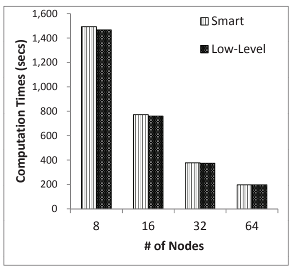
Figure 5: Performance Comparison with Spark

In addition, Smart can also achieve a much higher memory efficiency than Spark. It turns out that for all the three applications, Spark takes up constantly over 90% of the total memory (12 GB) whereas the memory consumption of Smart is only 4.3% (528 MB). Since the time-step size is already 512 MB, the analytics program run by Smart actually consumes only around 16 MB memory. Note that the time-step size is much smaller than the memory capacity, and hence Spark is very unlikely to spill the input to the disk.

5.3 Performance Comparison with Low-level Analytics Programs



(a) K-Means



(b) Logistic Regression

Figure 6: Performance Comparison with Low-Level Programs

In the second experiment, we compared both the programmability and performance of analytics programs written using Smart against the ones that were manually implemented in OpenMP and MPI. 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.

First, it turns out that Smart is effective in simplifying application development, by saving the efforts on implementing and debugging low-level parallelization details. Specifically, for k-means and logistic regression, 55% and 69%, respectively, of the lines of OpenMP/MPI codes in the low-level implementations are either eliminated or converted into sequential code by Smart. Note that these low-level codes are usually the most error-prone part for the programmers.

Second, we will like to understand performance overheads that arise as well. Figure 6 shows the results. First, we find that the low-level codes for k-means can outperform Smart version by up to 9%. Such performance difference is mainly due to the extra overheads involved in the global combination of Smart. In the manual 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. Note that we follow such a design for a better applicability and flexibility – the keys do not have to be continuous integers on each node, and early emission of reduction objects can be supported. 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, since in practice the total processing cost is mostly dominated by the simulation program, we do not expect noticeable overheads from our framework over hand-written low-level code.

5.4 Scalability Evaluation

The next set of experiments evaluate the scalability of Smart, by using both Heat3D and Lulesh simulations, and nine analytics programs: 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.

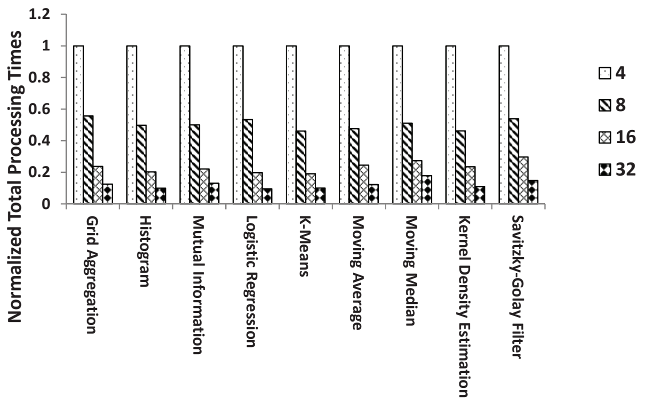


Figure 7: In-Situ Processing Times with Varying # of Nodes on Heat3D (Using 8 Cores per Node)

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

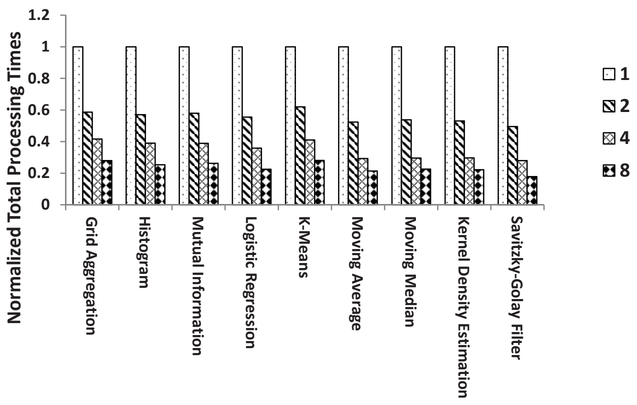


Figure 8: In-Situ Processing Times with Varying # of Threads on Lulesh (Using 64 Nodes)

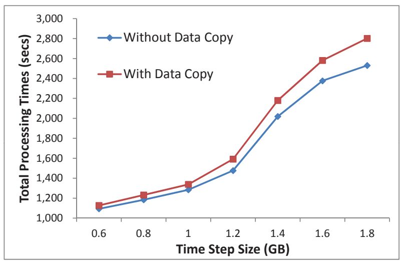
output by Heat3D over 100 time-steps. As Figure 7 shows, Smart can achieve 93% parallel efficiency on average for all the applications. Particularly, we can even see that, for some cases where 16 nodes are used, a super linear scalability can be achieved. Such an extra speedup is caused by the reduction 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. For example, compared with the first five applications, the window-based applications are more compute-intensive, and the synchronization overheads weigh much less in the total processing cost, leading to a better scalability.

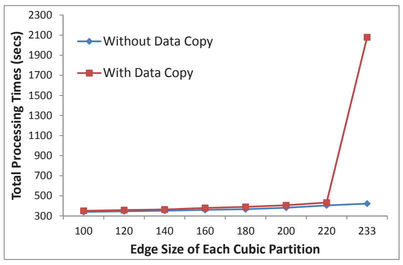
5.5 Evaluating Memory Efficiency

We next demonstrate a key advantage of Smart design (its time sharing mode implementation) – in-situ analytics can be supported even *without involving an extra copy* of the simulation output. Many simulation programs practically use almost all available memory on the machine, and unnecessary copying of data can lead to severe performance degradation – this is even more important as memory to flop ratio has decreased with many recent systems. 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 GB 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



(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 Time Sharing Mode

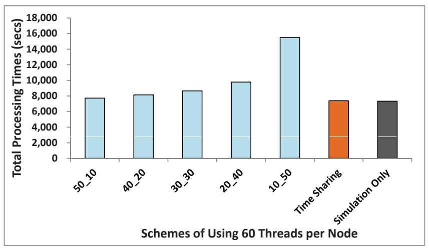
size was varied from 100 to 233, and the corresponding time-step size was varied from 1.5 GB to 18.3 GB.

As shown in Figure 9, we can see that when our system does not involve any data copy, there can be a notable performance improvement. 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 capacity, and hence its processing time increases substantially. For this case, our system can achieve a speedup of 5x.

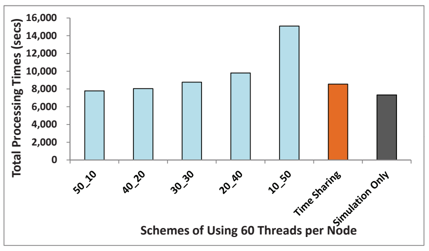
5.6 Comparing Time Sharing and Space Sharing Modes

Recall that in the space sharing mode, both simulation and analytics run concurrently, using two separate groups of cores on each node. All of our experiments so far have considered the time sharing mode only. Now we evaluate the efficiency of space sharing mode, by comparing against the performance in time sharing mode, as well as the performance of pure simulation as a baseline, on a many-core cluster.

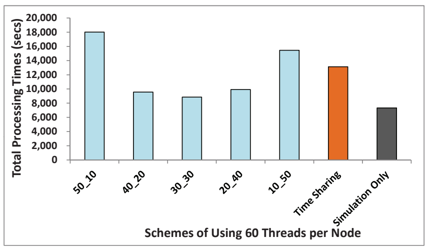
In this set of experiments, 1 TB data was output by Lulesh on 8 Xeon Phi nodes. Since it turns out that the simulation could not benefit from hyperthreading on the coprocessors, we only used 60 threads for computation in this mode, and 1 core was reserved for scheduling and communication. Histogram, k-means, and moving median were used as analytics programs, with the same parameters as for the experiments in Section 5.4. Besides time sharing and 'simulation-only' versions, to vary the number of cores used for simulation and analytics in space sharing mode, we used 5 dif-



(a) Histogram



(b) K-Means



(c) Moving Median

Figure 10: Evaluating the Efficiency of Space Sharing Mode

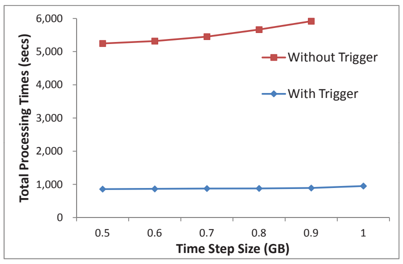
ferent versions, in which the number of cores used for simulation was varied from 50 to 10, and the remaining cores were used for analytics.

The results are shown in Figure 10. Here, "n_m" denotes a space sharing scheme with n threads for simulation and m threads for analytics. First, although the best performance in space 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 space sharing mode for k-means and moving median are achieved by "50_10" and "30_30", and reflect a performance improvement of over the time sharing mode by 10% and 48%, respectively. This is because space 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 space sharing mode – the best performance of histogram in space sharing mode (achieved by "50_10") is 4.4% lower than the performance from the time sharing mode. This is because the synchronization (or message passing) cost in histogram is relatively higher that in the other two applications, and space sharing mode can only execute the message passing in simulation and analytics sequentially, to avoid the potential data race in MPI, i.e., only a single thread can call MPI function at a time during concurrent execution. Thus, we conclude that space sharing mode can be advantageous when a simulation program does not scale well with increasing number of cores, but it is not a good fit for the applications involving frequent synchronization.

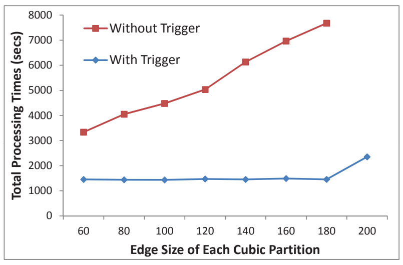
5.7 Evaluating the Optimization for Window-Based Analytics

The last set of experiments evaluate the effect of optimization for window-based analytics. Specifically, we compare the optimized version against an implementation that does not set a trigger function and hence cannot support early emission of the reduction object. In the first experiment, 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. In the second experiment, 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.

The results are shown by Figure 11. We can see that the optimization can lead to a speedup of up to 5.6 and 5.2 in the two experiments, respectively, which is because of the memory efficiency. For



(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

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 the moving average application. 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 the trigger mechanism, due to the extremely large memory consumption – we have not even reported the results for these two cases.

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 [7], 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 [57], e.g., grid aggregation and moving average. Moreover, we believe our system can well support a variety of scientific analytics, from both applicability and performance view-points. Unlike the scientific simulations that often require point-to-point communication and fail to fit into MapReduce, many complex analytics programs can still be expressed as MapReduce jobs or even nuanced MapReduce pipelines (e.g., mutual information). We have also demonstrated the high efficiency of our system compared with the manual implementations in low-level libraries.

6. RELATED WORK

As recent years have witnessed an increasing performance gap between I/O and compute capabilities, in-situ scientific analytics [21, 23, 64, 72] has attracted much attention. As we stated in Section 1, the research on in-situ scientific analytics has been mainly focused on two areas – *applications* and *platforms*. In-situ applications and algorithms have been extensively studied, with work on topics including indexing [20, 23], compression [24, 73], visualization [19, 61, 71], and other analytics like object tracking [66], feature extraction [25], and fractal dimension analysis [50]. On the other hand, in-situ resource scheduling research that offers platforms can be classified into *time sharing* and *space sharing* categories. An example of time sharing platform is GoldRush [69], 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 space sharing platforms, CPU utilization of simulation and analytics are decoupled, while the memory bound on analytics still holds. Examples of efforts include Functional Partitioning [26], and the systems Damaris [12] and CoDS [65]. By contrast, our work explores the opportunities in in-situ scientific analytics at the *programming model* level. 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 of the in-situ resource scheduling platforms.

In a broader context of online resource scheduling platforms, another two processing modes have been studied in addition to in-situ processing. The first is *in-transit* processing, where by leveraging extra resources, online analytics can be moved to dedicated *staging nodes* that are different from the nodes where simulation runs. Platforms supporting this mode include PreDatA [68], GLEAN [50], JITStager [4], and NESSIE [35]. Based on the observation that in-situ and in-transit modes can complement each other, the second mode is that of *hybrid* processing. This mode is supported on many platforms, including ActiveSpaces [10], DataSpace [11], FlexIO [70], and others [6]. Our system can be incorporated into these platforms to support in-transit or hybrid processing.

We had earlier compared the limitations of various MapReduce implementations for possible in-situ analytics, and have extensively compared our work against Spark. In addition, iMR [30] is specifically designed for in-situ log stream processing. To meet the in-situ resource constraints, iMR focuses on lossy processing and load shedding. Smart, in comparison, is based on a distinct API that reduces memory requirements. Further, integrating MapReduce with scientific analytics has been a topic of much interest recently [7, 29, 32, 40, 49, 51, 59, 67]. SciHadoop [7] integrates Hadoop with NetCDF library support to allow processing of NetCDF data with MapReduce API. SciMATE [59] is a MapReduce variant that can transparently process scientific data in multiple scientific formats. Zhao *et al.* [67] implement a parallel storage and access method for NetCDF data based on MapReduce. The Kepler+Hadoop project [51] integrates MapReduce with Kepler, which is a scientific workflow platform. Himach [49] extends MapReduce to support molecular dynamics trajectory data analysis. MARP [40] and KMR [32] are other two MapReduce-based frameworks that can support scientific analytics in MPI environment. SciHive [15] can support querying scientific data like some scientific data query processing engines [45, 58], and it is built on top of MapReduce. In contrast, Smart is designed for in-situ processing, and accordingly, the focus is on addressing the data loading mismatch, memory constraint, and other similar issues. 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 have developed and evaluated a system that applies MapReduce-style API for developing in-situ analytics programs. Our work has addressed a number of challenges in creating data analytics programs from a high-level API that is efficient and can share resources with an ongoing simulation program.

We have extensively evaluated our framework. 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 low-level implementations. Moreover, we have demonstrated both the functionality and scalability of our system by running different simulation and analytics programs in different in-situ modes on clusters with multi-core and many-core nodes. We can achieve 93% parallel efficiency on average. Finally, we show that our optimization for in-situ window-based analytics can achieve a speedup of up to 5.6. 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] Heat3D. http://dournac.org/info/parallel_heat3d.

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

[4] Hasan Abbasi, Greg Eisenhauer, Matthew Wolf, Karsten Schwan, and Scott Klasky. Just in time: adding value to the IO pipelines of high performance applications with JITStaging. In *HPDC*, pages 27–36. ACM, 2011.

[5] Ahmed M Aly, Asmaa Sallam, Bala M Gnanasekaran, L Nguyen-Dinh, Walid G Aref, Mourad Ouzzani, and Arif Ghafoor. M3: Stream processing on main-memory mapreduce. In *ICDE*, pages 1253–1256. IEEE, 2012.

[6] David A Boyuka, Sriram Lakshminarasimham, Xiaocheng Zou, Zhenhuan Gong, John Jenkins, Eric R Schendel, Norbert Podhorszki, Qing Liu, Scott Klasky, and Nagiza F Samatova. Transparent In Situ Data Transformations in ADIOS. In *CCGRID*, pages 256–266. IEEE, 2014.

[7] Joe B. Buck, Noah Watkins, Jeff LeFevre, Kleoni Ioannidou, Carlos Maltzahn, Neoklis Polyzotis, and Scott Brandt. SciHadoop: Array-based Query Processing in Hadoop. In *SC*, 2011.

[8] Tyson Condie, Neil Conway, Peter Alvaro, Joseph M Hellerstein, Khaled Elmeleegy, and Russell Sears. MapReduce Online. In *NSDI*, volume 10, page 20, 2010.

[9] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. In *OSDI*, pages 137–150, 2004.

[10] Ciprian Docan, Manish Parashar, Julian Cummings, and Scott Klasky. Moving the code to the data-dynamic code deployment using activespaces. In *IPDPS*, pages 758–769. IEEE, 2011.

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

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

[13] Jaliya Ekanayake, Shrideep Pallickara, and Geoffrey Fox. Mapreduce for data intensive scientific analyses. In *eScience*, pages 277–284. IEEE, 2008.

[14] Zacharia Fadika, Elif Dede, Madhusudhan Govindaraju, and Lavanya Ramakrishnan. Mariane: Mapreduce implementation adapted for hpc environments. In *GRID*, pages 82–89. IEEE, 2011.

[15] Yifeng Geng, Xiaomeng Huang, Meiqi Zhu, Huabin Ruan, and Guangwen Yang. SciHive: Array-based query processing with HiveQL. In *TrustCom*, pages 887–894. IEEE, 2013.

[16] Clemens Heitzinger, A Hossinger, and Siegfried Selberherr. On smoothing three-dimensional monte carlo ion implantation simulation results. *TCAD*, 22(7):879–883, 2003.

[17] LI Hsu, Steven G Self, Douglas Grove, Tim Randolph, Kai Wang, Jeffrey J Delrow, Lenora Loo, and Peggy Porter. Denoising array-based comparative genomic hybridization data using wavelets. *Biostatistics*, 6(2):211–226, 2005.

[18] Wei Jiang, Vignesh T. Ravi, and Gagan Agrawal. A Map-Reduce System with an Alternate API for Multi-core Environments. In *CCGRID*, pages 84–93, 2010.

[19] Homa 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.

[20] Jinoh Kim, Hasan Abbasi, L. Chacon, C. Docan, S. Klasky, Qing Liu, Norbert Podhorszki, A. Shoshani, and Kesheng Wu. Parallel in situ indexing for data-intensive computing. In *LDAV*, pages 65–72. IEEE, 2011.

[21] Scott Klasky, Hasan Abbasi, Jeremy Logan, Manish Parashar, Karsten Schwan, Arie Shoshani, Matthew Wolf, Sean Ahern, Ilkay Altintas, Wes Bethel, et al. In situ data processing for extreme-scale computing. *SciDAC*, 2011.

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

[23] Sriram Lakshminarasimhan, David A Boyuka, Saurabh V Pendse, Xiaocheng Zou, John Jenkins, Venkatram Vishwanath, Michael E Papka, and Nagiza F Samatova. Scalable in situ scientific data encoding for analytical query processing. In *HPDC*, pages 1–12. ACM, 2013.

[24] Sriram Lakshminarasimhan, Neil Shah, Stephane Ethier, Scott Klasky, Rob Latham, Rob Ross, and Nagiza F Samatova. Compressing the incompressible with ISABELA: In-situ reduction of spatio-temporal data. In *Euro-Par*, pages 366–379. Springer, 2011.

[25] Aaditya G Landge, Valerio Pascucci, Attila Gyulassy, Janine C Bennett, Hemanth Kolla, Jacqueline Chen, and Peer-Timo Bremer. In-situ feature extraction of large scale combustion simulations using segmented merge trees. In *SC*, pages 1020–1031. IEEE, 2014.

[26] Min Li, Sudharshan S Vazhkudai, Ali R Butt, Fei Meng, Xiaosong Ma, Youngjae Kim, Christian Engelmann, and Galen Shipman. Functional partitioning to optimize end-to-end performance on many-core architectures. In *SC*, pages 1–12. IEEE, 2010.

[27] Qing Liu, Jeremy Logan, Yuan Tian, Hasan Abbasi, Norbert Podhorszki, Jong Youl Choi, Scott Klasky, Roselyne Tchoua, Jay Lofstead, Ron 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.

[28] Zhi Liu, Ahmed Abbas, Bing-Yi Jing, and Xin Gao. Wavpeak: picking nmr peaks through wavelet-based smoothing and volume-based filtering. *Bioinformatics*, 28(7):914–920, 2012.

[29] Sarah Loebman, Dylan Nunley, Yong-Chul Kwon, Bill Howe, Magdalena Balazinska, and Jeffrey P Gardner. Analyzing massive astrophysical datasets: Can Pig/Hadoop or a relational DBMS help? In *CLUSTER*, pages 1–10. IEEE, 2009.

[30] Dionysios Logothetis, Chris Trezzo, Kevin C Webb, and Kenneth Yocum. In-situ MapReduce for log processing. In *USENIX ATC*, page 115, 2011.

[31] Zhihan Lv, Alex Tek, Franck Da Silva, Charly Empereur-mot, Matthieu Chavent, and Marc Baaden. Game on, science-how video game technology may help biologists tackle visualization challenges. *PloS one*, 8(3):e57990, 2013.

[32] Motohiko Matsuda, Naoya Maruyama, and Shin'ichiro 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] William J McCausland, Shirley Miller, and Denis Pelletier. Simulation smoothing for state–space models: A computational efficiency analysis. *Computational Statistics & Data Analysis*, 55(1):199–212, 2011.

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

[35] Ron A Oldfield, Gregory D Sjaardema, Gerald F Lofstead II, and Todd Kordenbrock. Trilinos i/o support (trios). *Scientific Programming*, 20(2):181–196, 2012.

[36] Christopher Olston, Benjamin Reed, Utkarsh Srivastava, Ravi Kumar, and Andrew Tomkins. Pig Latin: A Not-So-Foreign Language for Data Processing. In *SIGMOD*, pages 1099–1110. ACM, 2008.

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

[38] Colby Ranger, Ramanan Raghuraman, Arun Penmetsa, Gary Bradski, and Christos Kozyrakis. Evaluating mapreduce for multi-core and multiprocessor systems. In *HPCA*, pages 13–24. IEEE, 2007.

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

[40] Saba Sehrish, Grant Mackey, Jun Wang, and John Bent. MRAP: A Novel MapReduce-based Framework to Support HPC Analytics Applications with Access Patterns. In *HPDC*, pages 107–118, 2010.

[41] Ju Shen and S Cheung. Layer depth denoising and completion for structured-light rgb-d cameras. In *CVPR*, pages 1187–1194. IEEE, 2013.

[42] Ju Shen, Po-Chang Su, Sen-ching Samson Cheung, and Jian Zhao. Virtual mirror rendering with stationary rgb-d cameras and stored 3-d background. *Image Processing, IEEE Transactions on*, 22(9):3433–3448, 2013.

[43] Avraham Shinnar, David Cunningham, Vijay Saraswat, and Benjamin Herta. M3R: increased performance for in-memory Hadoop jobs. *VLDB*, 5(12):1736–1747, 2012.

[44] Yu Su, Yi Wang, and Gagan Agrawal. In-situ bitmaps generation and efficient data analysis based on bitmaps. In *HPDC*. ACM, 2015.

[45] Yu Su, Yi Wang, Gagan Agrawal, and Rajkumar Kettimuthu. SDQuery DSI: integrating data management support with a wide area data transfer protocol. In *SC*, page 47. ACM, 2013.

[46] Justin Talbot, Richard M Yoo, and Christos Kozyrakis. Phoenix++: modular MapReduce for shared-memory systems. In *MapReduce'11*, pages 9–16. ACM, 2011.

[47] Alex Tek, Benoist Laurent, Marc Piuzzi, Zhihan Lu, Matthieu Chavent, Marc Baaden, Olivier Delalande, Christine Martin, Lorenzo Piccinali, Brian Katz, et al. Advances in human-protein interaction-interactive and immersive molecular simulations. *Biochemistry, Genetics and Molecular Biology"Protein-Protein Interactions-Computational and Experimental Tools'*, pages 27–65, 2012.

[48] Ashish Thusoo, Joydeep Sen Sarma, Namit Jain, Zheng Shao, Prasad Chakka, Suresh Anthony, Hao Liu, Pete Wyckoff, and Raghotham Murthy. Hive - A Warehousing Solution Over a Map-Reduce Framework. *PVLDB*, 2(2):1626–1629, 2009.

[49] Tiankai Tu, Charles A Rendleman, David W Borhani, Ron O Dror, Justin Gullingsrud, MO Jensen, John L Klepeis, Paul Maragakis, Patrick Miller, Kate A Stafford, et al. A scalable parallel framework for analyzing terascale molecular dynamics simulation trajectories. In *SC*, pages 1–12. IEEE, 2008.

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

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

[52] Jim Jing-Yan Wang, Xiaolei Wang, and Xin Gao. Non-negative matrix factorization by maximizing correntropy for cancer clustering. *BMC Bioinformatics*, 14:107, 2013.

[53] Jim Jing-Yan Wang, Yi Wang, Shiguang Zhao, and Xin Gao. Maximum mutual information regularized classification. *Engineering Applications of Artificial Intelligence*, 37:1–8, 2015.

[54] Jingjing Wang, Nael Abu-Ghazaleh, and Dmitry Ponomarev. Interference resilient pdes on multi-core systems: towards proportional slowdown. In *SIGSIM*, pages 115–126. ACM, 2013.

[55] Jingjing Wang, Deepak Jagtap, Nael Abu-Ghazaleh, and Dmitry Ponomarev. Parallel discrete event simulation for multi-core systems: Analysis and optimization. *IEEE Trans. Parallel Distrib. Syst.*, 25(6):1574–1584, 2014.

[56] Yi Wang, Gagan Agrawal, Gulcin Ozer, and Kun Huang. Removing sequential bottlenecks in analysis of next-generation sequencing data. In *HiCOMB*, pages 508–517. IEEE, 2014.

[57] Yi Wang, Arnab Nandi, and Gagan Agrawal. SAGA: Array Storage as a DB with Support for Structural Aggregations. In *SSDBM*, page 9. ACM, 2014.

[58] Yi Wang, Yu Su, and Gagan Agrawal. Supporting a Light-Weight Data Management Layer Over HDF5. In *CCGrid*, pages 335–342. IEEE, 2013.

[59] Yi Wang, Jiang Wei, and Gagan Agrawal. SciMATE: A Novel MapReduce-Like Framework for Multiple Scientific Data Formats. In *CCGRID*, pages 443–450, may 2012.

[60] Jianjun Yang and Zongming Fei. Broadcasting with prediction and selective forwarding in vehicular networks. *International journal of distributed sensor networks*, 2013, 2013.

[61] Hongfeng Yu, Chaoli Wang, Ray W Grout, Jacqueline H Chen, and Kwan-Liu Ma. In situ visualization for large-scale combustion simulations. *IEEE Computer Graphics and Applications*, 30(3):45–57, 2010.

[62] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J Franklin, Scott Shenker, and Ion Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *NSDI*, pages 2–2. USENIX Association, 2012.

[63] Matei Zaharia, Mosharaf Chowdhury, Michael J Franklin, Scott Shenker, and Ion Stoica. Spark: cluster computing with working sets. In *HotCloud*, pages 10–10, 2010.

[64] Boyu Zhang, Trilce Estrada, Pietro Cicotti, and Michela Taufer. Enabling in-situ data analysis for large protein-folding trajectory datasets. In *IPDPS*, pages 221–230. IEEE, 2014.

[65] Fan Zhang, Ciprian Docan, Manish Parashar, Scott Klasky, Norbert Podhorszki, and Hasan Abbasi. Enabling in-situ execution of coupled scientific workflow on multi-core platform. In *IPDPS*, pages 1352–1363. IEEE, 2012.

[66] Fan Zhang, Solomon Lasluisa, Tong Jin, Ivan Rodero, Hoang Bui, and Manish Parashar. In-situ Feature-Based Objects Tracking for Large-Scale Scientific Simulations. In *SCC*, pages 736–740. IEEE, 2012.

[67] Hui Zhao, SiYun Ai, ZhenHua Lv, and Bo Li. Parallel Accessing Massive NetCDF Data Based on MapReduce. In *WISM*, pages 425–431, Berlin, Heidelberg, 2010. Springer-Verlag.

[68] Fang Zheng, Hasan Abbasi, Ciprian Docan, Jay Lofstead, Qing Liu, Scott Klasky, Manish Parashar, Norbert Podhorszki, Karsten Schwan, and Matthew Wolf. PreDatA–preparatory data analytics on peta-scale machines. In *IPDPS*, pages 1–12. IEEE, 2010.

[69] Fang Zheng, Hongfeng Yu, Can Hantas, Matthew Wolf, Greg Eisenhauer, Karsten Schwan, Hasan Abbasi, and Scott Klasky. GoldRush: resource efficient in situ scientific data analytics using fine-grained interference aware execution. In *SC*, page 78. ACM, 2013.

[70] Fang Zheng, Hongbo Zou, Greg Eisenhauer, Karsten Schwan, Matthew Wolf, Jai Dayal, Tuan-Anh Nguyen, Jianting Cao, Hasan Abbasi, Scott Klasky, et al. FlexIO: I/O Middleware for Location-Flexible Scientific Data Analytics. In *IPDPS*, pages 320–331. IEEE, 2013.

[71] Hongbo Zou, Karsten Schwan, Magdalena Slawinska, Matt Wolf, Greg Eisenhauer, Fang Zheng, Jai Dayal, Jeremy Logan, Qing Liu, Scott Klasky, et al. FlexQuery: An online query system for interactive remote visual data exploration at large scale. In *CLUSTER*, pages 1–8. IEEE, 2013.

[72] Hongbo Zou, Yongen Yu, Wei Tang, and Hsuan-Wei Michelle Chen. FlexAnalytics: a flexible data analytics framework for big data applications with I/O performance improvement. *Big Data Research*, 1:4–13, 2014.

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