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

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

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

Wei Jiang
Quantcast Corp.
wjiang@quantcast.com

ABSTRACT

In-situ analytics has lately been shown to be an effective approach to reduce both I/O and storage costs for scientific analytics. Developing an efficient in-situ implementation, however, involves many challenges, including parallelization, data movement or sharing, and resource allocation. Based on the premise that MapReduce can be an appropriate API for specifying scientific analytics applications, we present a novel MapReduce-like framework that supports efficient in-situ scientific analytics, and address several challenges that arise in applying the MapReduce idea for in-situ processing. Specifically, our implementation can load simulated data directly from distributed memory, and it uses a modified API that helps meet the strict memory constraints of in-situ analytics. The framework is designed so that analytics can be launched from the parallel code region of a simulation program. We have developed both *time sharing* and *space sharing* modes for maximizing the performance in different scenarios, with the former even avoiding any copying of data from simulation to the analytics program. We demonstrate the functionality, efficiency, and scalability of our system, by using different simulation and analytics programs, executed on clusters with multi-core and many-core nodes.

1. INTRODUCTION

A major challenge faced by data-driven discovery from scientific simulations is a shift towards architectures where memory and I/O capacities are not keeping pace with the increasing computing power [26]. 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* [21]. 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* [20,22,51,59] 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 [19, 22], compression [23, 60], visualization [18, 48, 58], and other analytics [24, 43, 53]; 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 [4,6,11,32,43,56]. These in-situ middleware systems mainly play the role of a *coordinator*, aiming to facilitate the underlying scheduling tasks, such as cycle stealing [56] and asynchronous I/O [43].

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 [50] 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. MAPREDUCE AND IN-SITU ANALYTICS

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

2.1 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 [18, 48], where most steps are embarrassingly parallel and others involve reductions; 2) statistical analytics [58] and similarity analytics [39] 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 [33] and Hive [41]; and 3) feature analytics [24] and clustering analytics [53], which have been efficiently implemented in MapReduce (though in an offline fashion), e.g., logistic regression and k-means clustering through Spark [50].

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, 27, 29, 37, 42, 44, 47, 54]. Therefore, an in-situ MapReduce-like framework can be a promising approach to improve both productivity and maintainability of scientific analytics.

2.2 Challenges

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

2.2.1 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 [38] 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 [34] and MRO-MPI [31] can load data from shared file system and/or local disk.

3. **Loading Data from Memory**: Pthread-based MapReduce prototypes like Phoenix [35], Phoenix++ [40], and MATE [17], 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 [28] 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 is still problematic, as simulation outputs are sporadic, and more critically, stream systems do not normally support iterative processing [45].

However, among all the MapReduce implementations we have examined, we find Spark [50] as an exception here. Its input data layout is defined as Resilient Distributed Dataset (RDD) [49], 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.

2.2.2 *Programming View Mismatch*

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*. On the other hand, the simplified interface of MapReduce presents a *sequential programming view*, which hides all the parallelization complexities. 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.

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.

2.2.3 *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. 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. 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.

2.2.4 *Programming Language Mismatch*

The last mismatch is from the programming languages that are used to implement simulation and analytics programs with MapReduce. Almost all of the HPC simulations in use are written in Fortran or C/C++, whereas 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, 31, 34, 35], 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

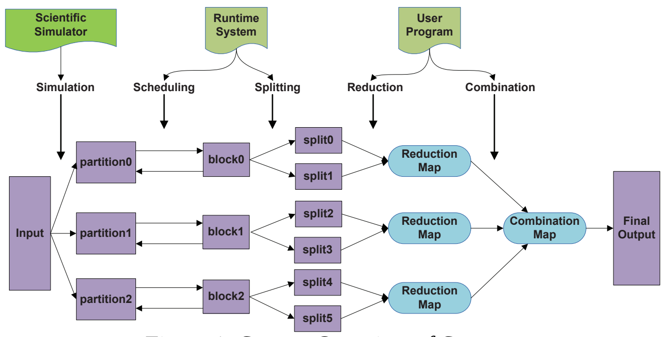


Figure 1: System Overview of Smart

Figure 1 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.

The above execution flow modifies the original MapReduce processing, but it is also the key to the high memory efficiency of Smart. Specifically, explicit declaration of the reduction object eliminates the shuffling phase of MapReduce. We will show the specifics of API through examples later in this section, but 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.

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 *space sharing* modes.

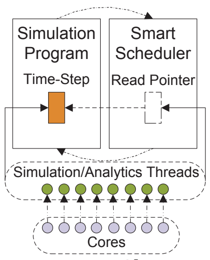


Figure 2: Time Sharing Mode

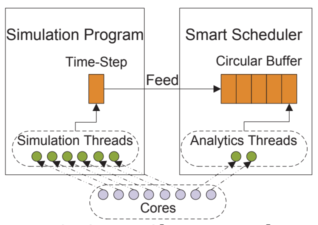


Figure 3: Space Sharing Mode

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 2, 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.

As shown in Figure 3, 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 Runtime System

3.3.1 Launching Smart 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.

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: 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. Lines 4 and 5 construct a derived Smart scheduler by specifying the number of threads, the size of a unit data chunk (i.e., unit element), the extra data besides the input array (e.g., the initial k centroids are required in k-means clustering), and the number of iterations. 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. 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.

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 somewhat larger amount of work than the 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 Smart, one task encapsulates the simulation code and then feeds its output to Smart (lines 6 - 13), and the other task runs analytics (lines 14 - 16). The number of threads used for simulation is specified within the simulation task. and the number of threads used for analytics is specified when Smart is initialized. Note that MPI codes are hidden in both simulation task and analytics task, and in this mode MPI functions may be called concurrently by different threads. Thus, to avoid the potential data race, the level of thread support should be upgraded to *MPI_THREAD_MULTIPLE* when MPI environment is initialized.

3.3.2 Data Processing Mechanism

Algorithm 1 shows the data processing mechanism in Smart. Note that all the functions called in the pseudo-code are from our

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 *}
```

API and either have to be or can be implemented by the programmers. Due to the space limit, the full description of API is available in a separate extended version [45]. At the beginning, some extra input can be processed to help initialize the combination maps as well as reduction maps (lines 1 - 6). In the reduction phase (lines 7 - 10), as a data block is divided into multiple splits, each thread processes a data split one unit element at a time. 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.

3.3.3 A Smart Analytics Example: Histogram

Listing 3: Histogram as an 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;
    }
};
```

We now illustrate how to develop an analytics program by using Smart. The analytics code can be reused in any (even offline) analytics mode. As an example, Listing 3 shows equi-width histogram construction. Two major steps are taken. To begin with, the user needs to define a derived reduction object class. Here 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.

From the above example, 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. More code examples can be found in our extended technical report [45].

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 [16] and smoothing [15, 30], 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*. More broadly, suppose we are calculating a derived quantity corresponding to each separate window snapshot centered by each input element (e.g., average for each distinct window of length W). The space complexity using Smart's processing is $\Theta(R \times N)$, where R and N denote two factors – the size of each reduction object and the maximal number of reduction objects maintained by Smart, respectively. On one hand, the size of the 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 will only be $\Theta(1)$, 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 $\Theta(K), 1 \leq K \leq W$.

On the other hand, the total number of reduction objects in window-based analytics is equal to the input data size, since each input element corresponds to a window snapshot in a form of reduction object. 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*.

To support such an optimization, the user only needs to overwrite the trigger function when deriving the reduction object class. Particularly for window-based applications, the emission condition can be the number of elements that have so far contributed to a reduction object equal to the window size, which indicates the reduction object value has been finalized. 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. 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 [50] – 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 2.1. The classes of analytics and specific applications are: 1) *visualization*: *grid aggregation* [46] 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 [53]; 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* [36] 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 2.2. Thus, to make a fair comparison, we let Spark bypass all the other mismatches with the following setup: 1) to bypass the programming view mismatch, 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.

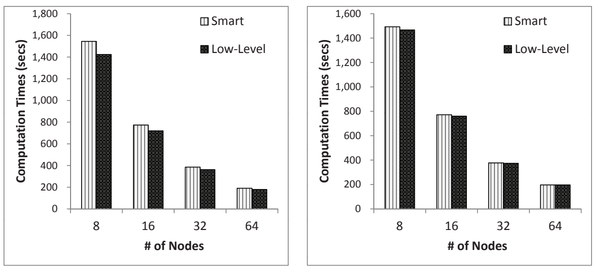
We used three applications for comparison, with the following parameters – 1) *logistic regression*: the number of iterations and the number of dimensions were 10 and 15, respectively; 2) *k-means*: the number of centroids, the number of iterations, and the number of dimensions were 8, 10, and 64, respectively; 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 4. 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) [49] due to its immutability. In comparison, all Smart operations are carried out on reduction maps and combination maps, and these maps can be reused even for iterative processing. Further, Spark serializes RDDs and send them through network even in local mode, whereas Smart avoids copying any reduction object from reduction map to combination map, by taking advantage of the shared-memory environment within each compute node.

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

In addition, Smart can also achieve a much higher memory efficiency than Spark. It turns out that 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

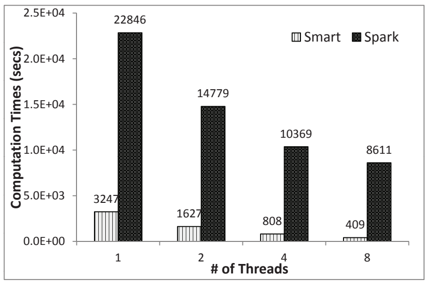
Figure 5: Performance Comparison with Low-Level Programs

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.

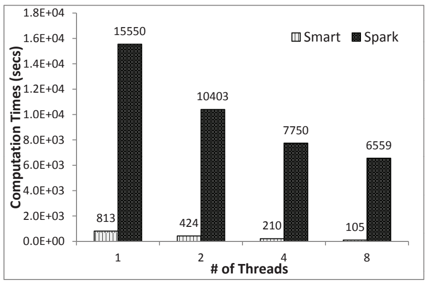
First, it turns out that Smart is effective in simplifying application development, by saving the efforts on implementing and debugging low-level parallelization details. Specifically, for k-means and logistic regression, 55% and 69%, respectively, of the lines of OpenMP/MPI codes in the low-level implementations 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 5 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.

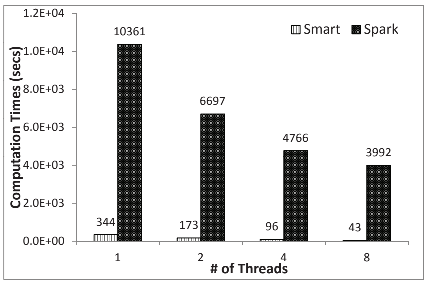
5.4 Scalability Evaluation



(a) Logistic Regression



(b) K-Means



(c) Histogram

Figure 4: Performance Comparison with Spark

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.

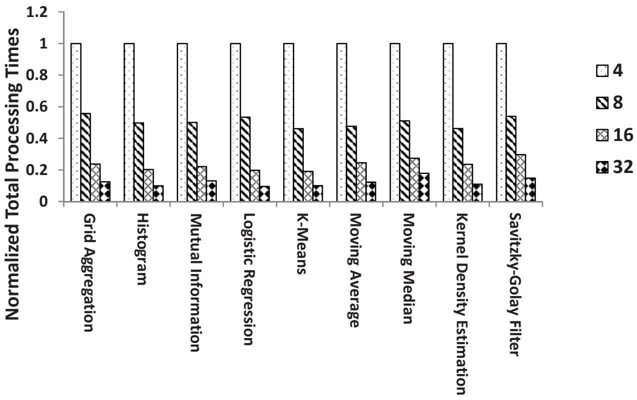


Figure 6: In-Situ Processing Times with Varying # of Nodes on Heat3D (Using 8 Cores per Node)

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 output by Heat3D over 100 time-steps. As Figure 6 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 7 shows the results. Smart can achieve 59% and 79% parallel efficiency on average for the first five applications, and the other four window-based appli-

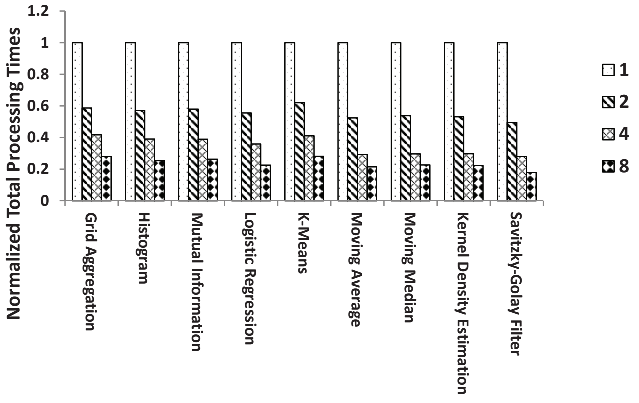


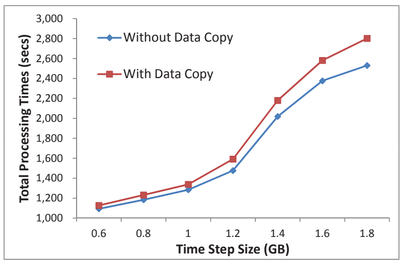
Figure 7: In-Situ Processing Times with Varying # of Threads on Lulesh (Using 64 Nodes)

cations, 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.

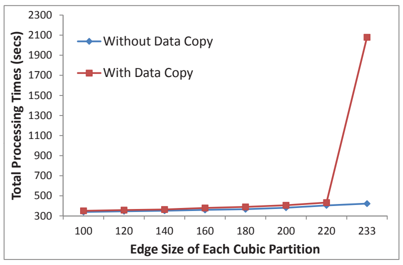
5.5 Evaluating Memory Efficiency

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 8: 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 8, 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.

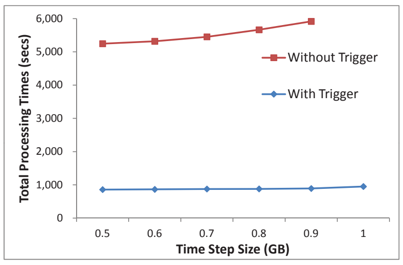
5.6 Comparing Time Sharing and Space Sharing Modes

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.

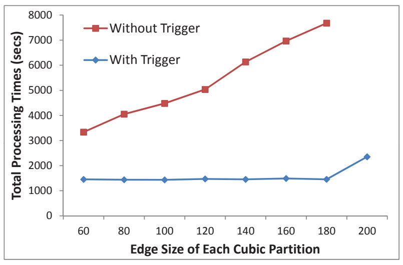
In this set of experiments, 1 TB data was output by Lulesh on 8 Xeon Phi nodes. Since it turns out that the simulation could not benefit from hyperthreading on the coprocessors, we only used 60 threads for computation in this mode, and 1 core was reserved for scheduling and communication. Histogram, 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 different 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 9. 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 9(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



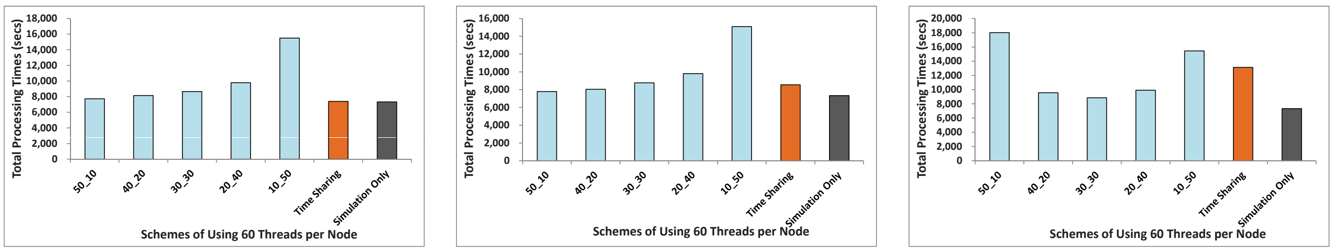
(a) In-Situ Processing Times of Moving Average on Heat3D (Using 4 Nodes)



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

Figure 10: Evaluating the Effect of 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



(a) Histogram (b) K-Means (c) Moving Median

Figure 9: Evaluating the Efficiency of Space Sharing Mode

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 10. 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 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.

6. RELATED WORK

As recent years have witnessed an increasing performance gap between I/O and compute capabilities, in-situ scientific analytics [20, 22, 51, 59] 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 [19, 22], compression [23, 60], visualization [18, 48, 58], and other analytics like object tracking [53], feature extraction [24], and fractal dimension analysis [43]. 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 [56], 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 [25], and the systems Damaris [12] and CoDS [52]. 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 [55], GLEAN [43], JITStager [4], and NESSIE [32]. 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 [57], 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 [28] 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, 27, 29, 37, 42, 44, 47, 54]. SciHadoop [7] integrates Hadoop with NetCDF library support to allow processing of NetCDF data with MapReduce API. SciMATE [47] is a MapReduce variant that can transparently process scientific data in multiple scientific formats. Zhao *et al.* [54] implement a parallel storage and access method for NetCDF data based on MapReduce. The Kepler+Hadoop project [44] integrates MapReduce with Kepler, which is a scientific workflow platform. Other MapReduce frameworks [29, 37, 42] are also developed for scientific analytics. 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] H. Abbasi, G. Eisenhauer, M. Wolf, K. Schwan, and S. Klasky. Just in time: adding value to the IO pipelines of high performance applications with JITStaging. In *HPDC*, pages 27–36. ACM, 2011.

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

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

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

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

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

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

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

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

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

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

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

[16] L. Hsu, S. G. Self, D. Grove, T. Randolph, K. Wang, J. J. Delrow, L. Loo, and P. Porter. Denoising array-based comparative genomic hybridization data using wavelets. *Biostatistics*, 6(2):211–226, 2005.

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

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

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

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

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

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

[23] S. Lakshminarasimhan, N. Shah, S. Ethier, S. Klasky, R. Latham, R. Ross, and N. F. Samatova. Compressing the incompressible with ISABELA: In-situ reduction of spatio-temporal data. In *Euro-Par*, pages 366–379. Springer, 2011.

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

[25] M. Li, S. S. Vazhkudai, A. R. Butt, F. Meng, X. Ma, Y. Kim, C. Engelmann, and G. Shipman. Functional partitioning to optimize end-to-end performance on many-core architectures. In *SC*, pages 1–12. IEEE, 2010.

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

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

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

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

[30] W. J. McCausland, S. Miller, and D. Pelletier. Simulation smoothing for state–space models: A computational efficiency analysis. *Computational Statistics & Data Analysis*, 55(1):199–212, 2011.

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

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

[33] C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins. Pig Latin: A Not-So-Foreign Language for Data Processing. In *SIGMOD*, pages 1099–1110. ACM, 2008.

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

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

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

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

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

[39] Y. Su, Y. Wang, and G. Agrawal. In-situ bitmaps generation and efficient data analysis based on bitmaps. In *HPDC*. ACM, 2015.

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

[41] A. Thusoo, J. S. Sarma, N. Jain, Z. Shao, P. Chakka, S. Anthony, H. Liu, P. Wyckoff, and R. Murthy. Hive - A Warehousing Solution Over a Map-Reduce Framework. *PVLDB*, 2(2):1626–1629, 2009.

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

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

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

[45] Y. Wang, G. Agrawal, T. Bicer, and W. Jiang. Smart: A MapReduce-Like Framework for In-Situ Scientific Analytics. Technical report, OSU-CISRC-4/15-TR05, Ohio State University, 2015.

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

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

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

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

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

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

[52] F. Zhang, C. Docan, M. Parashar, S. Klasky, N. Podhorszki, and H. Abbasi. Enabling in-situ execution of coupled scientific workflow on multi-core platform. In *IPDPS*, pages 1352–1363. IEEE, 2012.

[53] F. Zhang, S. Lasluisa, T. Jin, I. Rodero, H. Bui, and M. Parashar. In-situ Feature-Based Objects Tracking for Large-Scale Scientific Simulations. In *SCC*, pages 736–740. IEEE, 2012.

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

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

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

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

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

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

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