Smart User's Guide
Version 2.0

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

April 2015

Contents

Chapter 1

Introduction to Smart

Smart (in-Situ MApReduce liTe) is a MapReduce-like framework originally designed for in-situ scientific analytics. Unlike the conventional MapReduce implementations that mostly load data from file systems, Smart can load simulated data directly from memory in each node of a cluster. It leverages a MapReduce-like API to parallelize the analytics, while meeting the strict memory constraints on the analytics code when it is co-located with simulation. Using Smart for in-situ analytics requires only minimal changes to the simulation program itself. Smart can be launched in parallel (OpenMP and/or MPI) code region once each simulation output partition is ready, while the global analytics result can be directly fetched after the parallel code converges.

This document is a user's guide, written for scientists and developers who use Smart as a middleware for efficient scientific analytics.

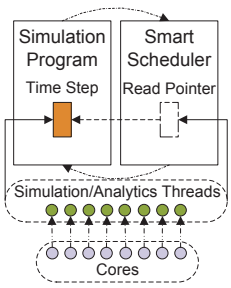

Figure 1.1: Space Sharing Mode

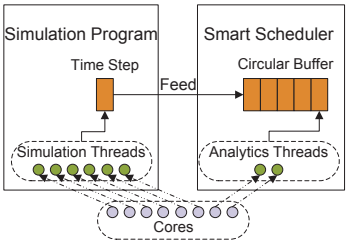

Figure 1.2: Time Sharing Mode

For the purpose of in-situ analytics, Smart provides both *space sharing* and *time sharing* modes for maximizing the performance in different scenarios. Space sharing mode aims to minimize the memory consumption of analytics, by avoiding extra data copy of simulation output, i.e., time step. This mode is generally preferred. Time sharing mode can support concurrent simulation and analytics on

two separate groups of cores of each node. This mode can be more useful when simulation task reaches a scalability bottleneck with the given resources. In addition, *offline analytics* for disk-resident data (in NetCDF/HDF5 format) is also supported. All these modes can be switched flexibly with the same analytics code (similar to a MapReduce job).

1.1 Space Sharing Mode for In-Situ Analytics

Space sharing mode can minimize the memory consumption of analytics, by avoiding extra data copy of simulation output. Note that although the memory copy itself is often not a costly operation, needing an extra copy of the data can lead to performance degradation. This is because that, to maximize the problem size that can be simulated on a platform, simulation programs often only leave very limited unused memory space, which has to be used for in-situ analytics.

As shown in Figure 1.1, to avoid an extra data copy, Smart sets a *read pointer* on the time step. Thus, time step can be shared by both simulation and analytics programs. Since an early overwrite of time step during simulation may kill the analytics running on it, a constraint of this mode is that, the simulation program cannot output the next time step until the analytics on the previous time step is completed. As a result, in this mode simulation and analytics run in turns, and each makes full use of all the cores of each node.

1.2 Time Sharing Mode for In-Situ Analytics

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

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

As shown in Figure 1.2, Smart maintains a *circular buffer* internally, in which each cell can be used for caching a time step. In this mode, one can view sim-

ulation program and Smart as producer and consumer, respectively. Once a time step is generated, if the circular buffer is not full, then this time step can be feeded to Smart by copying it to an empty cell. Otherwise, simulation program will be blocked until a cell in circular buffer becomes available. In this example, on an 8-core machine, 6 cores are dedicated to simulation, and 2 cores to analytics. The optimal division of the two groups of cores is determined by both hardware and specific application run, and such optimal division usually can be found by some sample runs.

1.3 Offline Analytics

Offline analytics serve as a complementary feature to analyze disk-resident data. Particularly, the current version can support the input data in NetCDF [2] or HDF5 [1] format. In addition, Smart can allow the user to perform analytics on other data formats, by implementing some format-specific data loading interfaces in Smart.

Chapter 2

Getting Started with Smart

This chapter discusses the setup of Smart environment, as well as the commands to run the programs in different analytics modes.

2.1 Smart Installation

2.1.1 Libraries to Install

To be adapted to scientific computing environment, Smart uses OpenMP for shard-memory computing within each compute node, and uses MPI for distributed computing among multiple compute nodes. Thus, both OpenMP and MPI should be installed. Particularly, we suggest that the OpenMP version should be at least 3.1.

Before running any application, we strongly recommend the user to set CPU affinity for OpenMP. It turns out that a 10X speedup can be achieved simply by setting an environment variable for the latest OpenMP version:

```
export OMP_PROC_BIND=true
```

Moreover, if offline analytics is required, to read the input data in NetCDF or HDF5 format, the corresponding libraries should be installed.

Smart is a lightweight MapReduce-like framework, and the current version almost does not require any specific configuration.

2.1.2 A Compiler to Support C++11

To compile the source code, C++11 should be supported. The version of GNU compiler should be at least 4.8.

If the Intel compiler is used, the version should be at least 14.0. Since some new C++11 features (e.g., constructor inheritance based on using-declaration)

currently are not supported by the Intel compiler, some minor modifications on C++11 syntax may be required. Particularly, as an example of running Smart compiled by the Intel compiler, an implementation of time sharing mode on the Intel MIC cluster can be found at [1], and some necessary C++ syntax modifications are marked in the source code.

2.2 Quick Start

2.2.1 Compiling and Linking

To support three different analytics modes, three sets of examples are provided under the directory *examples*, which consists of four subdirectories:

1. **common_app_headers**: This directory contains some application-specific analytics code.
2. **in_situ_space_sharing_analytics**: This directory contains the examples that run in-situ analytics in space sharing mode.
3. **in_situ_time_sharing_analytics**: This directory contains the examples that run in-situ analytics in time sharing mode.
4. **offline_analytics**: This directory contains the examples that run offline analytics over NetCDF/HDF5 data.

Since all the three modes can use same application-specific analytics code, all the analytics code is provided under the directory common_app_headers. Since these files are template classes, they are in the form of C++ header files.

For all the other three directories, each has a separate Makefile, which enables the user to compile the code on demand. We assume that all the libraries required are installed in */usr/local*. Otherwise, the user needs to either export the library paths to the environment variable, or manually add these paths in the Makefile.

Note that since all the application-specific analytics codes are written as template classes, if any header file is modified, "make clean" is required before recompilation.

To support Smart execution on the Intel MIC cluster, the user needs to make a MIC binary. In this case, the compiling flag "*-mmic*" should be added. To compile on an Intel MIC cluster, the user should use "-openmp" rather than "-fopenmp" to enable OpenMP. Otherwise, the thread level in OpenMP cannot be controlled on MIC nodes.

[1] `https://github.com/SciPioneer/Smart_TimeSharingMode_MIC`

2.2.2 Running Programs with Smart

Running In-Situ Analytics in the Space Sharing Mode

To use mvapich2, the user can use the following command to execute a program on 8 processors:

```
mpiexec -np 8 -f hostfile ./program
```

To use impi and run the program on an Intel MIC cluster, the user can use the following command:

```
ibrun.symm -m program
```

The number of allocated MIC nodes should be specified in a job file. Note that although in this case the program runs in a special symmetric mode on the MIC cluster. Since no CPU binary is loaded, only coprocessors are explicitly involved in the execution.

To maximize the performance, the number of processors used by the MPI environment should be equal to the number of compute nodes in the distributed environment. This is equivalent to setting the environment variable *PPN* (for the CPU cluster) or *MIC_PPN* (for the Intel MIC cluster) as 1. The number of threads used per node will be specified at the time launching Smart, as will be described in Chapter 3.

Running In-Situ Analytics in the Time Sharing Mode

To use mvapich2, unless global combination is not required, the user needs to make sure that *MV2_ENABLE_AFFINITY* is set as 0. Otherwise, the thread level cannot be controlled. The command to run on 32 processors can be:

```
mpirun_rsh -np 32 -hostfile hostfile \
MV2_ENABLE_AFFINITY=0 ./program
```

To use impi and run the program on the Intel MIC cluster, use the same command as in the space sharing mode.

Running Offline Analytics

The commands for offline analytics are exactly same as in the space sharing mode. In addition, two dummy data generators are provided to generate NetCDF and HDF5 files. To generate a simple NetCDF file “data.nc”, a simple command can be:

```
./netcdf_data_generator
```

Similarly, To generate a simple HDF5 file “data.h5”, a simple command can be:

```
./hdf5_data_generator
```

These two data generator files are only used for creating some sample input data. The user can provide their own input data files. In the current version, a Smart scheduler can only process a single variable in a file at a time. However, the user can create as many Smart schedulers as needed.

Chapter 3

Launching Smart

This chapter shows how to launch Smart in three different analytics modes. Smart runtime can be simply invoked in the application code to initialize the system and run the analytics, and all the details of data movement, scheduling, and resource allocation are transparent to the user. The APIs to launch Smart at runtime is shown by Table 3.1

3.1 Launching Smart for In-Situ Analytics in Space Sharing Mode

Listing 3.1: Launch Smart for In-Situ Analytics in Space Sharing Mode

```
1 void simulate(Out* out, size_t out_len, const Param& p) {
2     /* Each process simulates an output partition of data type In and length in_len.
          */
3     // Launch Smart after simulation in the parallel code region.
4     SchedArgs args(num_threads, chunk_size, extra_data, num_iters);
5     unique_ptr<Scheduler<In, Out>> smart(new DerivedScheduler<In, Out>(args));
6     smart->set_red_obj_size(sizeof(RedObj));
7     smart->run(partition, in_len, out, out_len);
8 }
```

A distinguishing feature of Smart is the *ease of use*. In space sharing mode, Smart can minimize the modification of the original simulation code. As demonstrated in Listing 3.1, to run Smart in this mode, only 3 lines (lines 4 - 7) need to be added when the simulation data is ready. The example code shows the execution of processing a single time step. Note that the definition of reduction object, as well as the derived Smart scheduler class, are implemented in a separate file for Smart analytics, which does not add any complexity of the original simulation code. Developing Smart analytics code will be discussed in Chapter 4.

Table 3.1: Descriptions of the Functions to Launch Smart

Functions Provided by the Scheduler
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 space 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 space sharing mode
7) $void\ feed(const\ In*\ in, size\_t\ in\_len)$ Feeds an input in time sharing mode
8) $void\ run(Out*\ out, size\_t\ out\_len)$ Runs the analytics by generating a single key given a unit chunk in time sharing mode
9) $void\ run2(Out*\ out, size\_t\ out\_len)$ Runs the analytics by generating multiple keys given a unit chunk in time sharing mode
Functions Provided by the Partitioner
1) $Partitioner(const\ string\&\ filename,\ const\ string\&\ varname,\ size\_t\ chunk\_size)$ Initializes the Smart partitioner by specifying the file name, the variable name, and the size of a unit chunk
2) $load\_partition()$ Loads a data partition
3) $get\_len()\ const$ Retrieves the partition length
4) $get\_data()\ const$ Retrieves the partitioned data

After each data partition is simulated given a set of simulation parameters p (line 2), line 4 constructs a scheduler argument $args$, which specifies the number of threads per process $num\_threads$, the size of a unit chunk $chunk\_size$, the extra data for analytics $extra\_data$, and the number of iterations num_iters. To maximize the analytics performance, $num\_threads$ should be equal to the number of threads used for simulation. $chunk\_size$ is the size of processing unit, and it can often be viewed as the length of *feature vector* in analytics applications. $extra\_data$ is used when some additional input is required, e.g., the initial k centroids are required in k-means clustering. num_iters can be specified for iterative processing. By default, $extra\_data$ and num_iters are initialized as a null pointer and 1, respectively. Line 5 constructs a derived Smart scheduler instance $smart$ with the scheduler argument $args$. Note that Smart scheduler class is defined as a template class, and hence Smart can be utilized for taking any array type as input or output, without complicating the application code. In line 6, the size of reduction object $RedObj$ is set for its deserialization required by message passing. In the current implementation, the reduction object should have a fixed size. In line 7, Smart launches analytics by taking the partitioned data as the input, and the final result will be output to the given destination. Alternatively, the user can also call $get\_combination\_map$ to retrieve the combination map, and then manually transform it into a desired output. During the entire process, all the parallelization details are hidden in a sequential programming view.

3.2 Launching Smart for In-Situ Analytics in Time Sharing Mode

Listing 3.2: Launch Smart for In-Situ Analytics in Time 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 }
```

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

3.3 Launching Smart for Offline Analytics

Listing 3.3: Launch Smart for Offline Analytics

```
1  void offlineRun(Out* out, size_t out_len, const string& filename, const string&
       varname, const SchedArgs& args) {
2      // Data partitioning and data loading.
3      unique_ptr<Partitioner> p(new DerivedPartitioner(filename, varname, args.
           chunk_size));  // Both NetCDF and HDF5 formats are natively supported.
4      p->load_partition();
5
6      // Launch Smart after a data partition is loaded.
7      unique_ptr<Scheduler<In, Out>> smart(new DerivedScheduler<In, Out>(args));
8      smart->set_red_obj_size(sizeof(RedObj));
9      smart->run((const In*)p->get_data(), p->get_len(), out, out_len);
10 }
```

Compared with in-situ analytics, to launch Smart for offline analytics, some extra effort is required for loading a specified input file and partitioning the input data. Specifically, a derived partitioner $DerivedPartitioner$ is created (line 3). The current implementation natively support loading data in NetCDF or HDF5 format, by providing both $NetCDFPartitioner$ and $HDF5Partitioner$ as two specific partitioner instances. To partition the data, the user only needs to specify the input file name $filename$, the variable name $varname$, and the unit chunk size $chunk\_size$ in the Scheduler argument $args$. Line 4 loads a data partition as an 1D array for the current computed node. To maximize the I/O performance, the implementation partitions the array data in the highest dimension by default. Lines 7 - 9 create a derived Smart scheduler. Particularly, line 9 passes both partitioned data and partition size to the constructor, by calling the partitioner's function get_data and get_len, respectively.

The offline analytics currently cannot work for the case of massive arrays, where the number of array elements may overflow the range of size_t (unsigned long integer). Besides, the partitioning process and data processing process are

decoupled in the implementation, and each partition is fed to Smart scheduler in one time. Thus, this simple implementation requires that each partition should be smaller than the memory size. We do not intend to overcomplicate the design of offline analytics here, because it turns out that in practice a partitioned array in a single data file usually can be fit into the memory.

In addition, if there is a need to support loading data in other formats, the user can develop a customized partitioner specific to the data format. Specifically, Smart allows the user to provide a derived partitioner, by overriding some pure virtual functions in the file *include/partitioner.h*.

Chapter 4

Developing Smart Analytics Code

This chapter guides the user to develop Smart analytics code via a set of simplified Smart API, which is potentially to be implemented by the programmers and shown by Table 4.1.

Table 4.1: Descriptions of the Functions to Develop Smart Analytics Code

Functions Implemented by the Scheduler
1) $virtual\ int\ gen\_key(const\ Chunk\&\ chunk)\ const$ Generates a single key given the unit chunk
2) $virtual\ void\ gen\_keys(const\ Chunk\&\ chunk,\ vector < int > \&\ keys)\ const$ Generates multiple keys given the unit chunk
3) $virtual\ void\ accumulate(const\ Chunk\&\ chunk,\ unique\_ptr < RedObj > \&\ red\_obj) = 0$ Accumulates the unit chunk on a reduction object
4) $virtual\ void\ merge(const\ RedObj\&\ red\_obj,\ unique\_ptr < RedObj > \&\ com\_obj) = 0$ Merges the first reduction object into the second reduction object, i.e., a combination object
5) $virtual\ void\ process\_extra\_data()$ Processes the extra input data to help initialize the combination map if necessary
6) $virtual\ void\ post\_combine()$ Performs post-combination processing if necessary
7) $virtual\ void\ convert(const\ RedObj\&\ red\_obj,\ Out*\ out)\ const$ Converts a reduction object to an output result if necessary
Functions Implemented by the RedObj
1) $virtual\ void\ reset()$ Reset the reduction object
2) $virtual\ bool\ trigger()\ const$ Set a trigger function for early emission

4.1 Introduction

This set of API is specific to an application, and is unrelated to any analytics mode. Thus, the same application developed based on Smart can run in different modes without any modification of the application-specific analytics code. Particularly, the directory *examples/common_app_headers* has provided six different example applications. This set of API is used for implementing the *run* (or *run2*) function in the previous API set shown by Table 3.1. 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. In addition, the programmers need to define a specialized reduction object as a subclass of the interface class *RedObj*.

4.2 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. In addition, to demonstrate the optimization of early emission of reduction object, we also provide another window-based example application – *moving average*.

4.2.1 Histogram as a Non-Iterative Example Application

Listing 4.1: Histogram as a Non-Iterative Example Application

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

```
18      // Merge red_obj into com_obj.
19      void merge(const RedObj& red_obj, unique_ptr<RedObj>& com_obj) override {
20          com_obj->count += red_obj->count;
21      }
22  };
```

As the first example, Listing 4.1 shows the pseudo code of equi-width histogram construction. To begin with, the user needs to define a derived reduction object class. In this example, the class $Bucket$ represents a histogram bucket, consisting of a single field $count$. Next, a derived system scheduler class $Histogram$ is defined. Note that to facilitate the manipulation on the datasets of different types, in our system the derived class can be defined as either a template class or a class specific to an input and/or output array type. For this kind of non-iterative application, the user usually only needs to implement three functions in Table 4.1 – *gen_key*, *accumulate*, and *merge*. First, the *gen_key* function computes the bucket ID based on the element value in the input data $chunk$, and the bucket ID serves as the returned key. For example, if the element value is located within the value range of the second bucket, then 2 will be returned. For simplicity, we assume that the minimum element value can be taken as priori knowledge or be retrieved by an earlier Smart analytics job. Note that in this application, since each element should be examined individually, each $chunk$ as a processing unit only contains a single element. Second, in the reduction phase, the *accumulate* function accumulates $count$ of the bucket that corresponds to the key returned by the gen_key function. Lastly, given two reduction objects, where the first one red_obj is from the reduction map, and the second one com_obj is from the combination map, the *merge* function merges $count$ on com_obj in the combination phase.

4.2.2 K-Means as an Iterative Example Application

Listing 4.2: K-Means Clustering as an Iterative Example Application

```
1   Derive a reduction object:
2   template <class T>
3   struct ClusterObj<T> : public RedObj {
4       T centroid[NUM_DIMS];
5       T sum[NUM_DIMS];
6       size_t size = 0;
7       void update(); // Update centroid by sum and size, which are then reset.
8   };
9   Derive a system scheduler:
10  template <class T>
11  class KMeans : public Scheduler<T, T*> {
12      // Compute the ID of the nearest centroid as the key.
13      int gen_key(const Chunk& chunk) const override {
14          /* Let C be the a set of centroids from the reduction objects in
                combination_map_. */
15          /* Find the centroid c nearest to the point represented by chunk from C.
                */
16          /* Return the key associated with c in combination_map_. */
```

```
17      }
18      // Accumulate chunk on sum and size of red_obj.
19      void accumulate(const Chunk& chunk, unique_ptr<RedObj>& red_obj) override {
20          red_obj->sum += chunk;  // Vector addition.
21          red_obj->size++;
22      }
23      // Merge red_obj into com_obj on sum and size.
24      void merge(const RedObj& red_obj, unique_ptr<RedObj>& com_obj) override {
25          com_obj->sum += red_obj->sum;  // Vector addition.
26          com_obj->size += red_obj->size;
27      }
28      // Process extra_data_ to set up the initial centroids in combination_map_.
29      void process_extra_data() override {
30          /* Transform extra_data_ into a set of cluster objects C. */
31          /* Load C into combination_map_. */
32      }
33      // Update the clusters for the next iteration.
34      void post_combine}() override {
35          for (auto& pair : combination_map_) {
36              RedObj* red_obj = pair->second.get();
37              red_obj->update();
38          }
39      }
40      // Extract the centroid from red_obj as the output.
41      void convert(const RedObj& red_obj, T** out) const override {
42          memcpy(*out, red_obj->centroid, sizeof(T) * NUM_DIMS);
43      }
44  };
```

As shown by Listing 4.2, 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.

4.2.3 Moving Average as a Window-Based Example Application

Listing 4.3: Moving Average as a Window-Based Example Application

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

In practice, (in-situ) analytics can be some preprocessing steps like smoothing and density estimation, which execute in a sliding window. A simple example of such window-based analytics is moving average, which replaces each array element with the average of all the elements within a sliding window.

As an optimization for memory efficiency, Smart can support *early emission of reduction object*. To fulfill such optimization, the user only needs to overwrite the *trigger function* when deriving the reduction object class. This trigger evaluates a self-defined *emission condition*, and determines if the reduction object should be early emitted from the reduction map. By default, the function returns *false*, and hence no early emission is triggered.

Listing 4.3 shows the implementation of moving average as a window-based application example. In this example, the reduction object counts the number of elements covered by a window, and the emission condition can be whether

the count equals the window size. This way, the maximal number of reduction objects maintained by Smart can be massively reduced, leading to a potentially significant improvement of memory efficiency. Note that since each input element contributes to multiple window snapshots, here we use the gen_keys function instead of gen_key in Table 4.1, to map each element to multiple keys.

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

Bibliography

[1] HDF5. http:// hdf.ncsa.uiuc.edu/HDF5/.

[2] NETCDF. http://www.unidata.ucar.edu/software/netcdf/.