

# iNFORMER Manual

## Table of Contents

Introduction .....	3
Dependencies .....	3
License Management.....	3
Hadoop Cluster Settings .....	3
Introduction to Smart.....	4
Time Sharing Mode for In-Situ Analytics.....	5
Space Sharing Mode for In-Situ Analytics.....	6
Offline Analytics.....	7
Getting Started with Smart .....	7
MPICH2-Yarn Installation .....	7
Smart Installation.....	7
Obtaining the License .....	7
Licensing Setup .....	7
Required Dependencies.....	8
Compilation.....	8
Building Examples .....	9
Running Smart Binaries.....	9
Developing with Smart code.....	11
Smart System Overview .....	11
Best Practices .....	12
Examining Output.....	13
Smart Initialization API .....	14
Initializing Smart in Various Modes.....	15
Time Sharing Mode.....	15
Space Sharing Mode .....	16
Offline Analytics .....	17
Explicit Instantiations of Smart.....	18
Smart Runtime and Execution API .....	18
Histogram as a Non-Iterative Example Application.....	20
K-Means as an Iterative Example Application .....	21

Moving Average as a Window-Based Example Application.....	23
Known Issues .....	24
Frequently Asked Questions (FAQs).....	25
Support Details .....	25
Works Cited .....	25

# Introduction

iNFORMER Smart is a framework that is aimed at supporting large scale data processing for in its native format through a low overhead MapReduce API. This document acts as a manual with introductory code samples and an overview on how to use the Smart system. Many of the figures, code samples and text are taken from the paper [1] and from the Smart UserGuide [2], licensed by OSU to RNET for commercial purposes.

## Dependencies

iNFORMER Smart has the following dependencies which must be present for successful compilation. These are:

- GCC 4.8.4 - Used for C++11 features.
  - If the Intel Compiler is being used
    - Should be at least version 14.0
    - May require modifications to the inheritance of constructors.
  - OpenMP should be newer than 3.1
- MPICH-3.2 - <http://www.mpich.org/downloads/>
- NetCDF 4.1.3
- HDF5 1.8.11
- Zlib 1.2.8

## License Management

To support the license management, the following dependencies must be installed

- Python 2.7 interpreter and library files.
- OpenSSL package. Version 1.0.1f supported. Earlier versions not tested.

## Hadoop Cluster Settings

iNFORMER Smart is certified for Yarn integration in Hadoop. To execute in a cluster (tested on Linux clusters), iNFORMER Smart will need the following YARN settings to be configured:

Property	Setting
yarn.nodemanager.linux-container-executor.resources-handler.class	org.apache.hadoop.yarn.server.nodemanager.util.CgroupsLCEResourcesHandler
yarn.nodemanager.container-executor.class	org.apache.hadoop.yarn.server.nodemanager

## 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 from a 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.

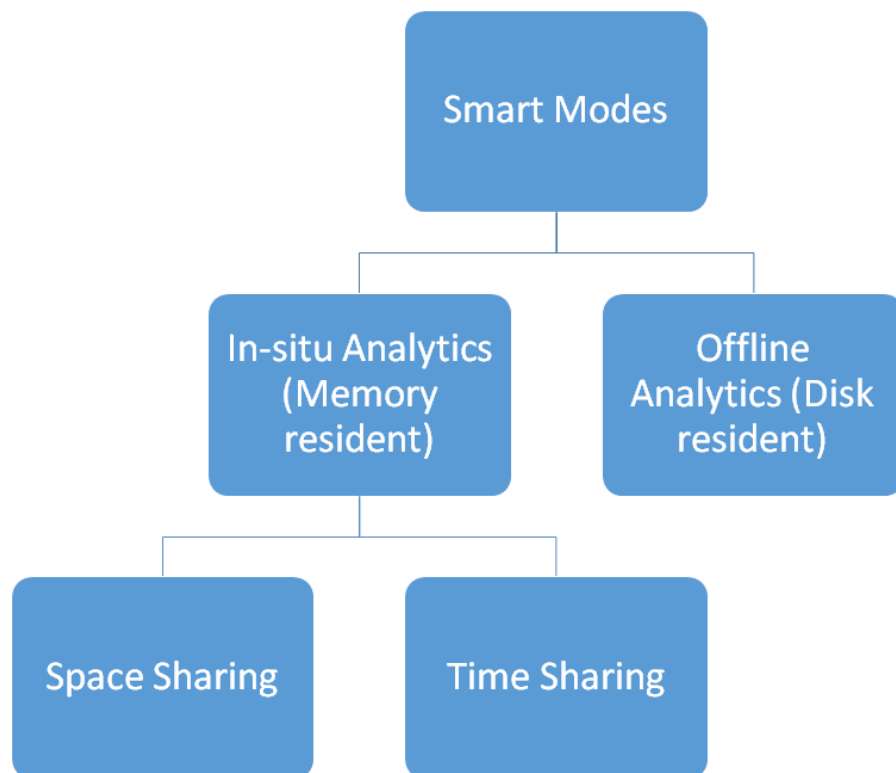


Fig 1. Smart operation modes

Smart can be initialized in various modes as shown in Figure 1. For the purpose of in-situ analytics, Smart provides both *time sharing* and *space sharing* modes for maximizing the

performance in different scenarios. *Time sharing* mode aims to minimize the memory consumption of analytics, by avoiding extra data copy of simulation output. This mode is generally preferred.

*Space 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).

## Time Sharing Mode for In-Situ Analytics

Time 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 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).

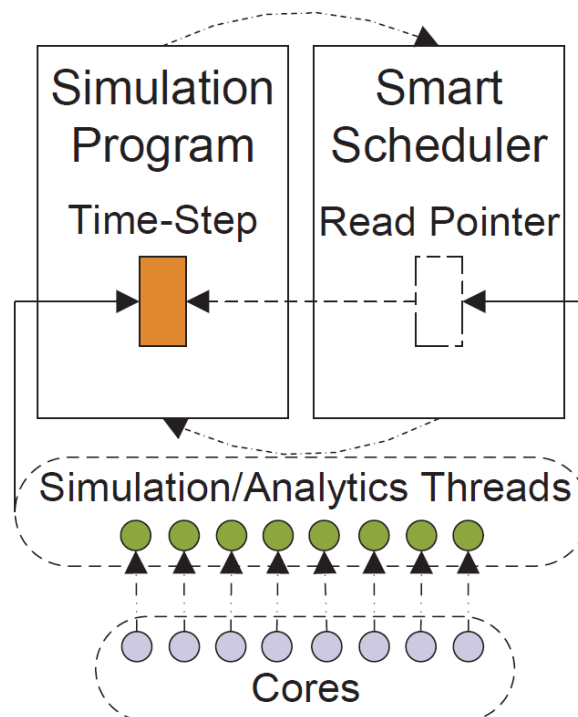


Fig 2: Time sharing mode in Smart

## Space Sharing Mode for In-Situ Analytics

Consider a cluster where every node is an Intel Xeon Phi. Since each coprocessor has a much larger number of cores than the CPU, a simulation program written for a standard multi-core cluster is unlikely to use all cores of the Xeon Phi effectively. In this case, instead of stopping the progress of simulation periodically and performing the analytics, one can easily dedicate a certain number of the available cores for the analytics. More specifically, all the cores are divided into two separate groups – one is specifically used for simulation, and the other is dedicated to analytics.

In this mode, besides the parallelism of multi-threading as well as the parallelism on multiple nodes, another task-level parallelism is placed on top of these two parallelism levels. As shown in Figure 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, the simulation program will be blocked until a cell in circular buffer becomes available. In this example where each node is though represented as an 8-core machine, 6 cores are dedicated to simulation, and 42 cores to analytics. The optimal division of the two groups of cores is determined by both hardware and specific application run, and currently we find such optimal division by sample runs.

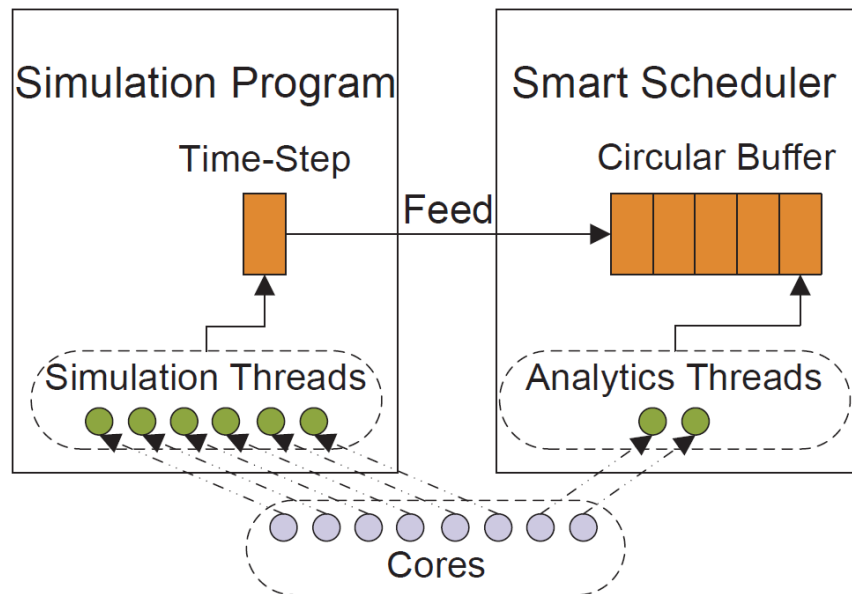


Fig 3: Space sharing mode in Smart

## 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 [3] or HDF5 [4] 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.

## 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.

Fetch the iNFORMER package by visiting the [iNFORMER License and Downloads](#) section. Please follow the instructions on the website and fill in the relevant details to obtain a trial iNFORMER License, which is necessary for using iNFORMER.

The downloaded iNFORMER package will contain two archives - mpich2-yarn and Smart.

## MPICH2-Yarn Installation

The MPICH2-Yarn archive is available as part of the download package. The archive will consist of three files:

- The MPICH2-Yarn JAR file.
- `yarn_mpi_server`, which is a binary executable.
- `mpi-site.xml`, which is processed during MPICH2-Yarn startup.

## Smart Installation

### Obtaining the License

The License file must be obtained from [iNFORMER License and Downloads](#) page as well. Required details must be filled in and an email will be sent with the trial license files which will be valid for a period of 30 days. For a complete license file, please send a request to [informer@RNET-Tech.com](mailto:informer@RNET-Tech.com)

### Licensing Setup

License file locations are specified for iNFORMER by setting two environment variables

```
INFORMER_CERTIFICATE_FILE  
INFORMER_PUBLIC_FILE
```

to point to the Certificate and Public Key files respectively.

## Required Dependencies

To be adapted to scientific computing environment, Smart uses OpenMP for shared-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. The MPI version used is MPICH-3.2 and the Hydra nameserver from the MPICH package must be present for full integration with YARN.

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
```

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

## Compiler with C++11 support

The version of GNU compiler should be at least 4.8.4 for C++11 support. 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.

## Compilation

Smart is distributed as a single library under folder (lib). Example programs are compiled against this library with appropriate flags for HDF5 [3] or NetCDF [4], if needed. The User has to ensure correct setup of HDF5 and other optional dependencies.

CMake is used for generating the build files for Smart. As per recommended CMake practices, it is advisable to have an *out of source tree* build. CMake doesn't currently support finding NetCDF installed modules, so if you would like to use your NetCDF installation, you would need to pass a hint to CMake.

1. Create an out of source build in a folder, say `build/`. Change directory to this folder.
2. From this folder, execute `cmake ${SMART_DWNLODED_BINARY_DIR} -DNETCDF_PATH=<netcdf path>`
3. The CMake script will look for the `include/` and `lib/` files under `<netcdf path>`. If the NetCDF path is not specified, a warning will be thrown, but the build setup will continue.



## Building Examples

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*: Contains some application-specific analytics code.
2. *in\_situ\_time\_sharing\_analytics*: Contains the examples that run in-situ analytics in time sharing mode.
3. *in\_situ\_space\_sharing\_analytics*: Contains the examples that run in-situ analytics in space sharing mode.
4. *offline\_analytics*: Contains the examples that run offline analytics over NetCDF/HDF5 data.

Since all the three modes can use same application-specific analytics code, the common header declarations are provided under the directory *common\_app\_headers*.

Once the CMake setup is done, simply call `make` at the top level build folder to build all example files. To build a specific examples folder, simply call `make` from that examples folder. Only the *offline\_analytics* examples would need HDF5 or NetCDF libraries. The other examples should compile without additional dependencies.

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.

## Building Custom Codes

If you would like to build your own test codes, you can modify the CMake generated Makefile or use a simplified Makefile that is present in each of the *examples/* folder (named `Makefile.bkup`). This file has placeholders for the location of HDF5, NetCDF and MPI libraries which need to be manually filled in.

## Running Smart Binaries

Smart is executed as an MPI process through Yarn. To facilitate this, we need `mpich-rnet-yarn` to be setup and installed as explained before. The user also needs to ensure that the Hydra nameserver (from MPICH-3.2) is up and running.

The `Mpich-rnet-yarn` application takes in the following command line parameters (also generated by doing `yarn jar <mpich-rnet-jar> --help`):

Opt	Expansion and Args	Default	Description
-----	--------------------	---------	-------------

		Value	
-a	--mpi-application <arg>	--	Location of the mpi application to be executed.
-c	--container-vcores <arg>	1	Number of virtual cores requested.
-D	-D <property=value>	--	DFS location, representing the source data of MPI.
-d	--debug	--	Dump out debug information.
-h	--help	--	Print usage information.
-k	--kill <arg>	--	Kill running MPI Application ID.
-M	--master-memory <arg>	64	Amount of memory in MB to be requested to run the Application Master.
-m	--container-memory <arg>	64	Amount of memory in MB to be requested to run the shell command.
-n	--num-containers <arg>	1	Number of containers on which the MPI program needs to be executed.
-o	--mpi-options <arg>	--	Options for MPI program.
-O	-O <property=value>	--	DFS location, representing the MPI result.
-P	--priority <arg>	0	Application Priority
-p	--container-priority <arg>	0	Priority for the shell command containers.
-q	--queue <arg>	default	RM Queue in which the application will be submitted.
-s	--mpi-name-service <arg>	""	MPICH Nameserver hostname
-S	-S <property=value>	--	Do all containers download the same file?
-t	--timeout <arg>	8.64 X 10 <sup>7</sup> , i.e. 24 hours	Wall-time, application timeout in milliseconds.

Table 1: Outline of MPI Yarn options.

To run a particular example, say, the *histogram\_time\_sharing*, we would execute it as:

```
yarn jar <JAR file of mpich-rnet-yarn> -a <histogram_time_sharing
binary> -n 1 -c 1 -s <nameserver host>
```

This particular example does not take in any filenames or parameters.

Consider running the `logistic_regression_offline` example. This may require more memory, so we explicitly specify it now, using the `-M` and `-m` options. This example also needs the filename (disk resident mode), so we specify that with the `-o` option.

```
yarn jar <JAR of mpich-rnet-yarn> -a <logistic_regression_offline
binary path> -M 1024 -m 1024 -n 1 -c 1 -s <nameserver host>
```

## Developing with Smart code

### Smart System Overview

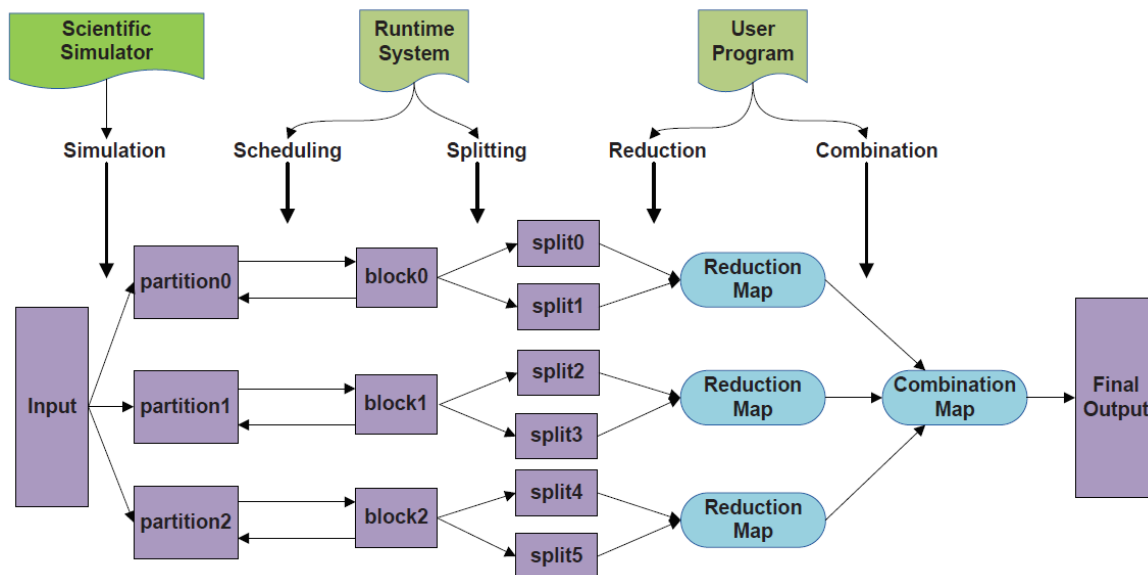


Fig 4: Design overview of Smart.

Figure 4 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. The data partition can be fed directly to the Smart analytics jobs (in-situ mode) or can be output to a disk and analyzed later (offline mode). 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, if each thread is bound to a specific CPU core, the performance is improved significantly (using `OMP_PROC_BIND` as shown earlier).

In processing elements within a split, there are two key operations, *reduction* and *combination*, which are carried out on two core map structures, a reduction map and a 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.

## Best Practices

The default mode for any executable created (like the examples) is for Yarn integration to be enabled, i.e. for the process to be run through Yarn using the `yarn jar` command. It is often beneficial to develop and validate the Smart code as an MPI executable first without running it through Yarn.

For this reason, when testing the code in a *standalone* manner, it is useful to define the macro

```
#define DISABLE_YARN_INTEGRATION
#include "scheduler.h"
```

before the first instance of `scheduler.h` is included.

Once the code is compiled, we will have a standard MPI executable that can be run with `mpiexec/mpiexec` as:

```
mpiexec -n 2 <smart mpi binary> <input file>
```

If the code output is as expected, the macro `DISABLE_YARN_INTEGRATION` can then be removed and the code recompiled. Now the executable can only be run through Yarn using the familiar pattern of

```
yarn jar <mpich yarn jar> -a <yarn enabled binary> -o <input file> ...
```

## Examining Output

Consider running the `histogram_time_sharing` example. In standalone development mode (refer section: “Best Practices”), the example can be run as:

```
mpiexec -n 1 ./histogram_time_sharing
```

The expected output should be

```
Simulation time = 0.00 secs.
```

```
Simulation data is ready...
```

```
Run in-situ processing...
```

```
Scheduler: Initializing with 2 threads and 1 nodes...
```

```
Scheduler: Constructing the reduction map for all the threads...
```

```
Scheduler: Reduction map for 2 threads is ready.
```

```
In-situ processing is done.
```

```
Combination map on node 0:
```

```
(key = 0, value = (count = 100))
```

```
(key = 1, value = (count = 100))
```

```
(key = 2, value = (count = 100))
```

```
(key = 3, value = (count = 100))
```

```
(key = 4, value = (count = 100))
```

```
(key = 5, value = (count = 100))
```

```
(key = 6, value = (count = 100))
```

```
(key = 7, value = (count = 100))
```

```
(key = 8, value = (count = 100))
```

```
(key = 9, value = (count = 100))
```

```
(key = 10, value = (count = 24))
```

```
Final output on the master node:
```

```
100 100 100 100 100 100 100 100 100 100 100 24
```

```
Analytics time = 0.00 secs.
```

```
Total processing time on node 0 = 0.00 secs.
```

When running through Yarn integrated mode, the output is stored as part of the Container logs. These can be inspected using the Yarn logs command, i.e.

```
yarn logs -applicationId <applicationId>
```

Like any Yarn application, logs belonging to the Client, Container and ApplicationMaster's Container would be displayed and the program output is stored as part of the Container logs. This requires the relevant Yarn log settings to be properly configured.

## Smart Initialization API

Smart is implemented in C++ and thus has a C++ based API. The following table illustrates the APIs that are relevant for the initialization of Smart.

Functions Provided by the Scheduler
<pre>SchedArgs(int num_threads, size_t chunk_size, const void* extra_data, int num_iters)</pre> <p>Initializes the Smart scheduler argument by specifying the # of threads, the size of a unit chunk, the extra data, and the # of iterations.</p>
<pre>explicit Scheduler(const SchedArgs&amp; args)</pre> <p>Initializes the Smart runtime system.</p>
<pre>void set_global_combination(bool flag)</pre> <p>Enable or disable global combination, which is enabled by default.</p>
<pre>const map &lt; int, unique_ptr &lt; RedObj &gt;&gt;&amp; get_combination_map() const</pre> <p>Retrieves the combination map.</p>
<pre>void run(const In* in, size_t in_len, Out* out, size_t out_len)</pre> <p>Runs the analytics by generating a single key given a unit chunk in time sharing mode.</p>
<pre>void run2(const In* in, size_t in_len, Out* out, size_t out_len)</pre> <p>Runs the analytics by generating multiple keys given a unit chunk in time sharing mode.</p>
<pre>void feed(const In* in, size_t in_len)</pre> <p>Feeds an input in space sharing mode.</p>
<pre>void run(Out* out, size_t out_len)</pre> <p>Runs the analytics by generating a single key given a unit chunk in space sharing mode.</p>
<pre>void run2(Out* out, size_t out_len)</pre> <p>Runs the analytics by generating multiple keys given a unit chunk in space sharing mode.</p>
Functions Provided by the Partitioner
<pre>Partitioner(const string&amp; filename, const string&amp; varname, size_t chunk_size)</pre> <p>Initializes the Smart partitioner by specifying the filename, the variable name and the size of a unit chunk.</p>

<code>load_partition()</code> Loads a data partition.
<code>get_len() const</code> Retrieves the partition length.
<code>get_data() const</code> Retrieves the partitioned data.

Table 2: Smart Initialization API.

## Initializing Smart in Various Modes

To initialize Smart, a `SchedArgs` object is created with the relevant parameters as shown in Table 2. Classes which represent the actual problem must inherit the `Scheduler` class as explained later. The `SchedArgs` created earlier is then simply passed into the constructor, assuming it is defined in line with C++ inheritance rules.

### Time Sharing Mode

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

Listing 1: Initializing Smart in Time Sharing

In time sharing mode, Smart can minimize the modification of the original simulation code. As demonstrated in Listing 1, to run Smart in this mode, only 3 lines 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 the Section on Smart Runtime and Execution API.

After each data partition is simulated given a set of simulation parameters `p`, scheduler argument `args` is constructed, 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. A derived Smart scheduler instance `smart` is constructed with the scheduler argument `args`.

The Smart scheduler class is internally a template class and is explicitly instantiated for certain Input and Output types only. These types are documented at [?](#). 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. Smart then 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.

## Space Sharing Mode

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

        #pragma omp task // Analytics task.
        for (int i = 0; i < num_steps; ++i)
            smart->run(out, out_len);
    }
}
```



```
}

```

Listing 2: Initializing Smart in Space Sharing

Space sharing mode requires more code reorganization than time sharing mode, since an extra task-level parallelism has to be deployed. Particularly, two Open-MP 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, and the other task runs the analytics. 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.

## Offline Analytics

```
void offlineRun(Out* out, size_t out_len, const string& filename,
const string& varname, const SchedArgs& args)
{
    // Data partitioning and data loading.
    unique_ptr<Partitioner> p(new DerivedPartitioner(filename, varname,
args.chunk_size)); // Both NetCDF and HDF5 formats are natively
supported.
    p->load_partition();

    // Launch Smart after a data partition is loaded.
    unique_ptr<Scheduler<In, Out>> smart(new DerivedScheduler<In,
Out>(args));
    smart->set_red_obj_size(sizeof(RedObj));
    smart->run((const In*)p->get_data(), p->get_len(), out, out_len);
}
```

Listing 3: Initializing Smart in Offline Mode

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.

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`. A data partition is loaded as a 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. A derived Smart scheduler instance is then created. In the `run` command, both partitioned data and partition size are passed 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`.

## Explicit Instantiations of Smart

The distributed binary package has explicit instantiations of Smart for the following classes:

Class In	Class Out
double	double
double	double*
double	unsigned long
float	unsigned long
float	double

Table 3: Explicit Instantiations of Smart

## Smart Runtime and Execution API

Functions Implemented by the Scheduler
<pre>virtual int gen_key(const Chunk&amp; chunk, const In* data, const map &lt; int, unique_ptr&lt; RedObj &gt;&gt;&amp; com_map) const</pre> <p>Generates a single key given the unit chunk (and combination map if necessary)</p>
<pre>virtual void gen_keys(const Chunk&amp; chunk, const In* data, vector &lt; int &gt;&amp; keys, const map &lt;int, unique_ptr&lt; RedObj &gt;&gt;&amp; com_map) const</pre>

<b>Generates multiple keys given the unit chunk (and combination map if necessary)</b>
<pre>virtual void accumulate(const Chunk&amp; chunk, const In* data, unique_ptr &lt; RedObj &gt;&amp; red_obj) = 0</pre> <p>Accumulates the unit chunk on a reduction object</p>
<pre>virtual void merge(const RedObj&amp; red_obj, unique_ptr &lt; RedObj &gt;&amp; com_obj) = 0</pre> <p>Merges the first reduction object into the second reduction object, i.e., a combination object</p>
<pre>virtual void process_extra_data(const void* extra_data, map &lt; int, unique_ptr &lt; RedObj &gt;&gt;&amp; com_map)</pre> <p>Processes the extra input data to help initialize the combination map if necessary</p>
<pre>virtual void post_combine(map &lt; int, unique_ptr &lt; RedObj &gt;&gt; &amp; com_map)</pre> <p>Performs post-combination processing and update the combination map if necessary</p>
<pre>virtual void deserialize(unique_ptr&lt;RedObj&gt;&amp; obj, const char* data) const = 0;</pre> <p>Construct a reduction object from serialized reduction object.</p>
<pre>virtual void convert(const RedObj&amp; red_obj, Out* out) const</pre> <p>Converts a reduction object to an output result if necessary</p>
<b>Functions Implemented by the RedObj</b>
<pre>virtual void reset()</pre> <p>Reset the reduction object</p>
<pre>virtual bool trigger() const</pre> <p>Set a trigger function for early emission</p>

Table 4: Smart Execution API.

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. The functions that have to be defined by the derived class are usually the `gen_key`, `gen_keys` (optional), `accumulate`, `merge`, `deserialize`.

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 2. 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` 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`.

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`.

## Histogram as a Non-Iterative Example Application

### Derive a reduction object:

```
struct Bucket : public RedObj {
    size_t count = 0;
};
```

### Derive a system scheduler:

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

Listing 4: Histogram Non-Iterative.

As the first example, Listing 4 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 – `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 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.

A step-by-step development of a Histogram application is outlined in <PPT link>.

## K-Means as an Iterative Example Application

### Derive a reduction object:

```
template <class T>
struct ClusterObj<T> : public RedObj {
    T centroid[NUM_DIMS];
    T sum[NUM_DIMS];
    size_t size = 0;
    void update(); // Update centroid by sum and size, which are then
reset.
};
```

### Derive a system scheduler:

```
template <class T>
class KMeans : public Scheduler<T, T*> {
    // Compute the ID of the nearest centroid as the key.
    int gen_key(const Chunk& chunk, const T* data, const map<int,
unique_ptr<RedObj>>& combination_map) const override {
        /* Let C be the a set of centroids from the reduction objects
in
        combination map. */
        /* Find the centroid c nearest to the point represented by
chunk from C.*/
        /* Return the key associated with c in combination map. */
    }

    // Accumulate chunk on sum and size of red_obj.
    void accumulate(const Chunk& chunk, const T* data,
unique_ptr<RedObj>& red_obj) override {
        red_obj->sum += chunk; // Vector addition.
    }
};
```

```

        red_obj->size++;
    }
    // Merge red obj into com_obj on sum and size.
    void merge(const RedObj& red_obj, unique_ptr<RedObj>& com_obj)
override {
    com_obj->sum += red_obj->sum; // Vector addition.
    com_obj->size += red_obj->size;
}
    // Process extra_data to set up the initial centroids in
    combination_map.
    void process_extra_data(const void* extra_data, map<int,
unique_ptr<RedObj>>&
combination_map) override {
    /* Transform extra data into a set of cluster objects C. */
    /* Load C into combination_map. */
}
    // Update the clusters for the next iteration.
    void post_combine(map<int, unique_ptr<RedObj>>& combination_map)
override {
    for (auto& pair : combination_map) {
        RedObj* red_obj = pair->second.get();
        red_obj->update();
    }
}

    // Extract the centroid from red_obj as the output.
    void convert(const RedObj& red_obj, T** out) const override {
        memcpy(*out, red_obj->centroid, sizeof(T) * NUM_DIMS);
    }
};

```

Listing 5: K-Means Iterative.

As shown by Listing 5, 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`.

The `process_extra_data` function then initializes the `combination_map` with the `extra_data` that indicates some initial centroids, and the `post_combine` function prepares for the next iteration, by updating all the clusters. Specifically, the centroid coordinates are computed by `sum` and `size`, which are then reset as zeros. Lastly, the `convert` function extracts the centroid coordinate from each reduction object as an output result. To make use of this function, a restriction is that, the integer key should start from 0.

## Moving Average as a Window-Based Example Application

### Derive a reduction object:

```
struct WinObj : public RedObj {
    double sum = 0;
    size_t count = 0;
    bool trigger() const override {
        return count == WIN_SIZE;
    }
};
```

### Derive a system scheduler:

```
template <class In>
class MovingAverage : public Scheduler<In, double> {
    // Take all the element positions covered by the window as the
    keys.
    void gen_keys(const Chunk& chunk, const In* data, vector<int>&
    keys, const
    map<int, unique_ptr<RedObj>>& combination_map) const override {
        // Each chunk has a single element, which is the center of the
        window.
        for (int i = max(chunk.start - WIN_SIZE / 2, 0); i <=
        min(chunk.start + WIN_SIZE / 2, total_len_); ++i) {
            keys.emplace_back(i);
        }
    }

    // Accumulate chunk on red obj.
    void accumulate(const Chunk& chunk, const In* data,
    unique_ptr<RedObj>& red_obj) override {
        if (red_obj == nullptr) red_obj.reset(new WinObj);
        red_obj->sum += data[chunk.start];
        red_obj->count++;
    }

    // Merge red obj into com obj.
    void merge(const RedObj& red_obj, unique_ptr<RedObj>& com_obj)
    override {
        com_obj->sum += red_obj->sum;
        com_obj->count += red_obj->count;
    }
};
```

```
// Transform red obj into average as the output.
void convert(const RedObj& red_obj, double* out) const override {
    *out = red_obj->sum / red_obj->count;
}
};
```

Listing 6: K-Means Iterative.

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 and smoothing, which also execute on a sliding window basis. A simple example of such window-based analytics is *moving average*, where the average of the elements within every window snapshot is computed. A critical challenge in the implementation of such window-based analytics is that of high memory consumption.

As an optimization for memory efficiency, Smart can support early emission of reduction object. To support such an 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 6 shows the implementation of moving average as a window-based application example. In this example, the reduction object counts the number of elements covered by a window, and the emission condition can be whether the count is equal to the window size. 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, to map each element to multiple keys. It should be noted that, this optimization is not only specific to in-situ window-based analytics, but also can be broadly applied to other applications, even for offline analytics. 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. Another example is time series, which can be viewed be as a window-based application in time dimension.

## Known Issues

1. Currently cannot pass MPI Hostfile through mpich2-yarn.
2. Specifying -n 2 on a single node cluster, seems to cause a hang.



## Frequently Asked Questions (FAQs)

For typical problems faced and their resolution, please visit the [iNFORMER FAQ](#) page.

## Support Details

For further queries and iNFORMER Support, please send a mail to

[informer@RNET-Tech.com](mailto:informer@RNET-Tech.com)

## Works Cited

1. Wang, Yi, Gagan Agrawal, Tekin Bicer, and Wei Jiang. "Smart." *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis on - SC '15* (2015). Print.
2. Wang, Yi, "Smart UserGuide".
3. HDF5 - <https://www.hdfgroup.org/HDF5/>
4. NETCDF - <http://www.unidata.ucar.edu/software/netcdf/>



240 W.Elmwood Dr,

Suite 2010

Dayton, OH, 45459-4248

