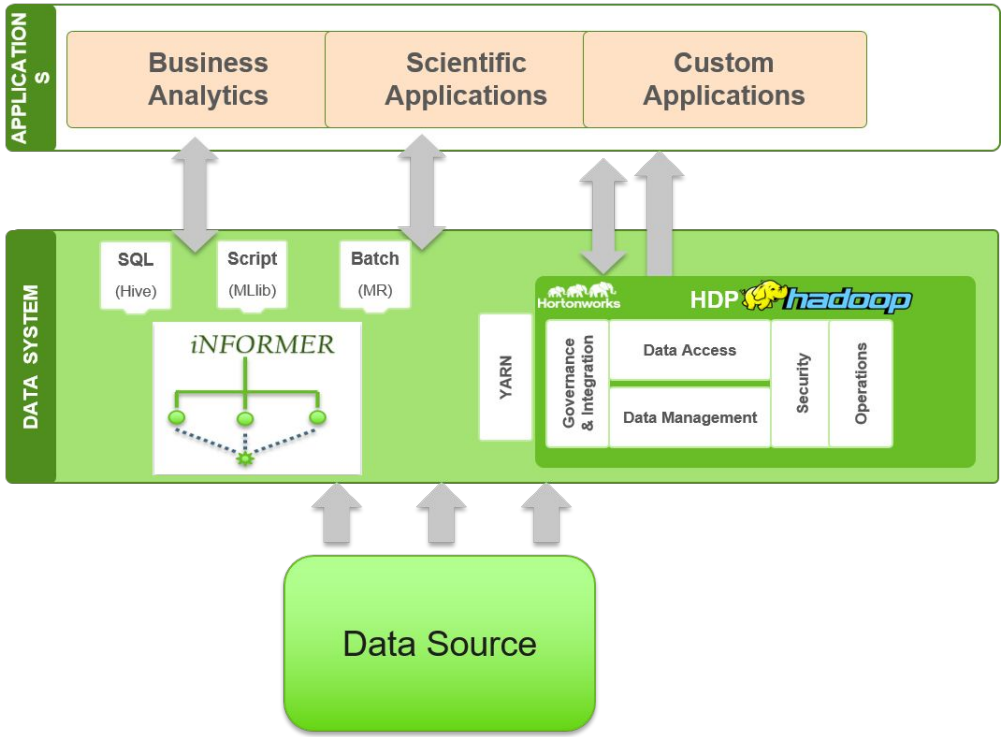
iNFORMER Walkthrough

RNET Team

Outline

- iNFORMER Introduction
- Obtaining iNFORMER.
- Installation and Licensing.
- Writing a Sample Application.
- Smart API
- Support Information

Introduction



- iNFORMER is a MapReduce like API for in-situ/streaming data processing framework.
- Implemented in C/C++ and has low-overhead.
- An MPI process that runs through YARN.
- Easy to use API with only a small number of functions to implement.

Obtaining iNFORMER and Licensing

Licensing

- Go to iNFORMER License and Downloads page and fill in the relevant details to obtain a trial License that would be valid for 30 days.
- The user will obtain a License certificate and a Public Key file after a few days.
- For full details on the Installation and Licensing for Smart, please refer to iNFORMERManual.pdf that is present in the iNFORMER package.

Writing a Sample Application

Outline

- The Smart system exposes two base classes (`Scheduler` and `RedObj`) and requires the user to override particular functions in the `Scheduler` class.
- This API is outlined at the end of the document for Reference.
- All MapReduce like problems can be broken down to this API and Smart presents a simplified interface using this API.
- The `examples/` folder outlines typical algorithms implemented using this API and in different execution modes.

Running an Example

- Smart based jobs can be run either through YARN or in a standalone mode.
- To run through YARN, the syntax to use is

```
yarn jar -a <binary> -o <args> -n <number of nodes> -c
                <cores> -s <nameserver host>
```

- To run Smart binary in a standalone mode, YARN integration must be disabled and the resulting application should be recompiled.
- For more information, please refer to the iNFORMER Manual.

Histogram Example - Step by Step Explanation

- Here we explain a histogram implementation using the `histogram_time_sharing` example.
- In this example, the input data is generated in the range (`rank`) to (`rank + (n-1)`) with `n` being the desired number of elements and `rank` being the process rank. Say `n = 5` and `rank = 0` (only one process, say), then the generated elements are (0, 1, 2, 3, 4).
- A histogram is simply a frequency distribution which counts the number of occurrences of elements belonging to a “bucket”, i.e. an interval size.

- If the bucket width was 1 (say), the histogram would then look like

Element	0	1	2	3	4
Bucket ID	0	1	2	3	4
Count	1	1	1	1	1

→ Key

→ Value

- With the bucket width as 2, the corresponding histogram would be

Element	0 1	2 3	4
Bucket ID	0	1	2
Count	2	2	1

→ Key

→ Value

- In the Smart system, the Key and Values shown are stored in the combination map.

Setting up the Example

- To reproduce the previous results, the example settings can be modified as follows:
  - In `histogram_time_sharing.cpp`, set the macro `NUM_ELEMS` to 5.
  - In `histogram.h`, set the `BUCKET_WIDTH` to 1.
  - If you would like to run the example as a standalone MPI binary, define the macro `DISABLE_YARN_INTEGRATION` before the first instance of `scheduler.h`.
- Compile the example and run it with -n 1 (either in standalone mode or through YARN). Examine the output and this should be the same as:

```
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 = 1))
      (key = 1, value = (count = 1))
      (key = 2, value = (count = 1))
      (key = 3, value = (count = 1))
      (key = 4, value = (count = 1))
Final output on the master node:
1 1 1 1 1
Analytics time = 0.00 secs.
Total processing time on node 0 = 0.00 secs.
```

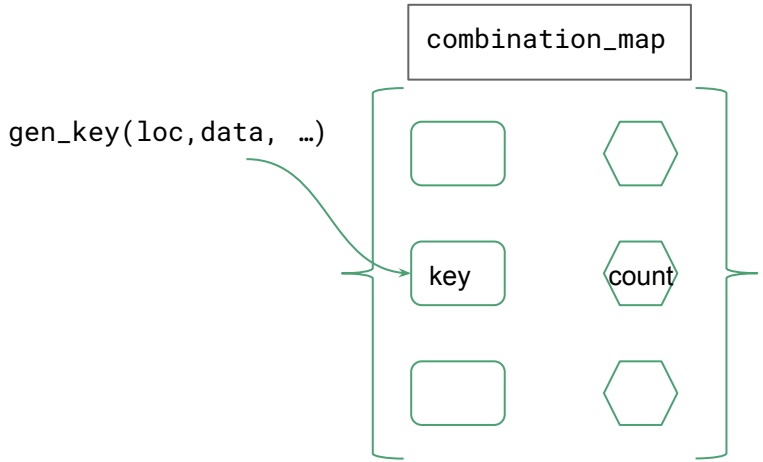
- Now set the `BUCKET_WIDTH` to `2`.
- Recompile and run with `-n 1` and examine the output, which should be as shown on the right (only the relevant part is shown here).
- How is this realized with the Smart API? We will explore that next.

```
…
…
…
In-situ processing is
done.

Combination map on node 0:
     (key = 0, value = (count = 2))
     (key = 1, value = (count = 2))
     (key = 2, value = (count =
1))
Final output on the master node:
2 2 1
Analytics time = 0.00 secs.
```

Implementation

- As we have seen before, the Bucket Id forms the `Key` in the combination map and the count forms the `Value`.
- The key is calculated from the `gen_key` function that needs to be defined when inheriting the Scheduler class.



- For our particular case, the `gen_key` is defined as:

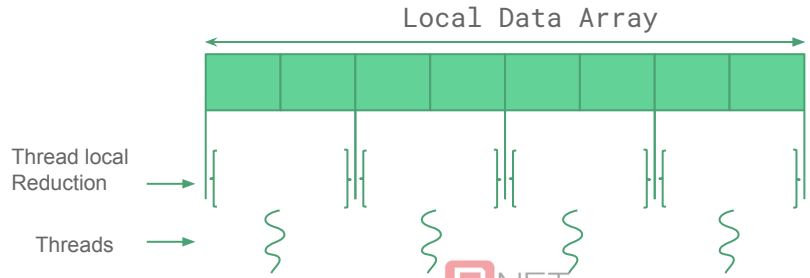
```
int gen_key(loc, data, ...) {
    return (int)(data[loc] - MIN_VAL) / BUCKET_WIDTH;
  }
```

- Which corresponds to the relevant bucket id, i.e. the key in the combination map.

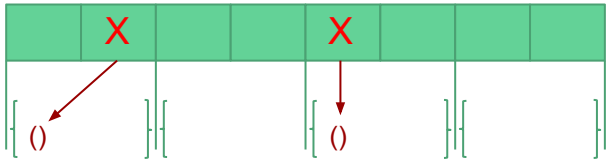
- The value in the key-value pair is updated by the accumulate function. This uses the pointer to the value object to simply increment the count (frequency).

```
  void accumulate(chunk, data, red_obj*) {
    if (red_obj == nullptr) {
      red_obj.reset(new Hist);
    }
    Hist* h = static_cast<Hist*>(red_obj.get());
    for (size_t i = 0; i < chunk.length; ++i) {
      dprintf("Adding the element chunk[%lu] = %.0f.\n", chunk.start + i,
data[chunk.start + i]);
      h->count++;
    }
  }
```

- Within a single rank, each thread can work on a portion of the array, i.e.



- If there are duplicate keys, they would need to be merged to ensure an accurate reduction, i.e.



X - duplicate value

- This is achieved by the function `local_combine`, which in turn calls the `merge` function, that needs to be implemented by the user as well.
- In the histogram case, it simply means that the bucket counts need to be summed up.

```
void merge(const red_obj, com_obj) {
  const Hist* hr = static_cast<const Hist*>(&red_obj);
  Hist* hc = static_cast<Hist*>(com_obj.get());

  hc->count += hr->count;
}
```

Multi-process case

- If we run the same example with more than one process, by specifying -n 2 (say) and keeping the Bucket width as 1, then the generated input data would be

Rank 0

0	1	2	3	4

Rank 1

1	2	3	4	5

- Within the rank, the corresponding combination map entries would then be

Rank 0

Element	0	1	2	3	4
Bucket ID	0	1	2	3	4
Count	1	1	1	1	1

Rank 1

Element	1	2	3	4	5
Bucket ID	1	2	3	4	5
Count	1	1	1	1	1

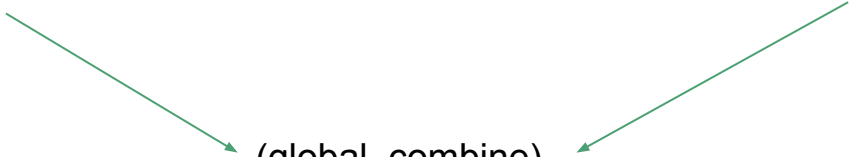
- Merging of the two entries is then carried out by `global_combine`.
- MPI routines are called here since this is an interprocess communication. This needs the combination maps to be serialized at the sender and deserialized at the receiving end.
- Serialization is handled internally and the combination map is sent as a sequence of bytes. Deserialization must then be implemented by the user:

```
void deserialize(obj, const char* data) {
  obj.reset(new Hist);
  memcpy(obj.get(), data, sizeof(Hist));
}
```

- The result of the merging should be

Key	0	1	2	3	4
Value	1	1	1	1	1

Key	1	2	3	4	5
Value	1	1	1	1	1



Key	0	1	2	3	4	5
Value	1	2	2	2	2	1

- Running the example with -n 2, we obtain the output (relevant parts shown) as:

```
In-situ processing is done.

Combination map on node 0:
     (key = 0, value = (count = 1))
     (key = 1, value = (count = 2))
     (key = 2, value = (count = 2))
     (key = 3, value = (count = 2))
     (key = 4, value = (count = 2))
     (key = 5, value = (count = 1))
Final output on the master node:
1 2 2 2 2 1
```

Histogram Class - Definition

```
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(Chunk, data, combination_map) override {
    // Each chunk has a single element.
    return (data[chunk.start] - MIN)) / BUCKET_WIDTH;
  }
  // Accumulate chunk on red_obj.
  void accumulate(chunk, data, red_obj) override {
    if (red_obj == nullptr) red_obj.reset(new Bucket);
    red_obj->count++;
  }
  // Merge red_obj into com_obj.
  void merge(red_obj, com_obj) override {
    com_obj->count += red_obj->count;
  }
};
```

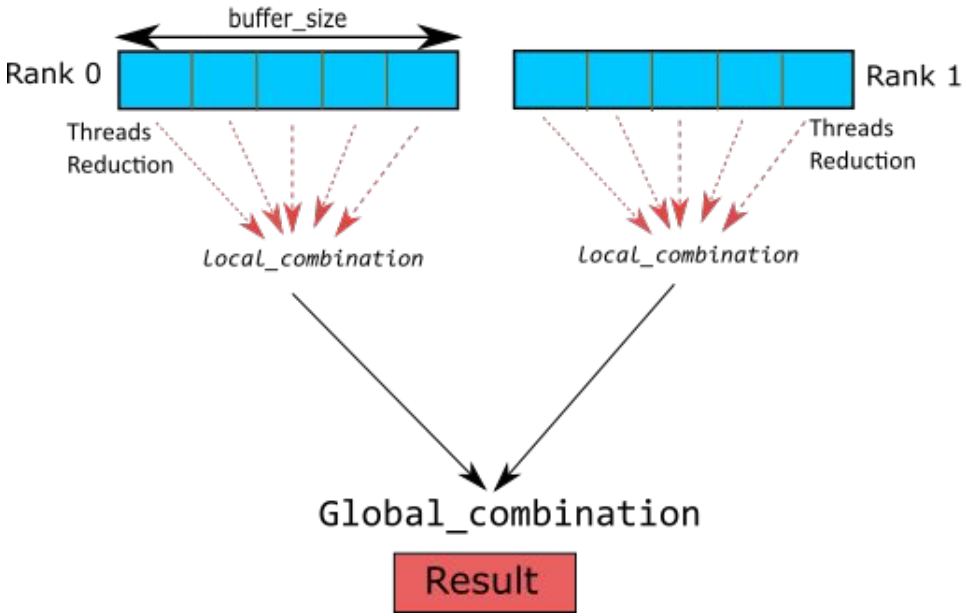
- Function signatures are approximated here. For full details, please refer to Table 4 in the iNFORMER User Manual.

- For this particular example, we need to implement only the three functions:
  - gen_key
  - accumulate
  - merge

Execution - Time Sharing Mode

```
  SchedArgs args(NUM_THREADS, STEP); // predefined macros
  unique_ptr<Scheduler<float, size_t>> h( new
Histogram<float>(args));
  h->set_red_obj_size(sizeof(Hist));
  h->run(in, total_len, nullptr, 0);   // Note that here the
output array is nullptr.
  if (rank == 0)
    printf("In-situ processing is done.\n");
```

- Initialize the Scheduler arguments.
- Initialize the Smart runtime with the arguments.
- Set the size of the reduction object.
- Trigger the run function for time sharing mode.



- Each rank takes care of a partition of data.
- Within a rank, reduction is carried out with threads and local combination.
- If desired, global combination is carried out.

Smart API

Initialization API

`SchedArgs(int num_threads, …)`	Initialize the Scheduler with relevant args.
`Scheduler(const SchedArgs& args)`	Initialize Smart runtime.
`set_global_combination(bool flag)`	Enable/disable global combination. (Default = enabled)
`get_combination_map()`	Retrieve the local combination map.
`run(Type* in, size_t len, …)`	Runs the analytics by generating a single key given a unit chunk in time sharing mode.
`run2(Type* in, size_t len, …)`	Runs analytics by generating multiple keys.
`feed(args, …)`	Feeds input in space sharing mode.
`run(Type* out, …)`	Run by generating single key in space sharing mode
`run2(Type* out, …)`	Run by generating multiple keys in space sharing mode

Execution API - Scheduler

`int gen_key(chunk, data, …)`	Generate a single key given unit chunk.
`gen_keys(chunk, data, keys, …)`	Generate multiple keys given unit chunk.
`accumulate(chunk, data, red_obj)`	Accumulate unit chunk on reduction object.
`merge(red_obj, com_obj)`	Merge first reduction object into second.
`process_extra_data(extra_data, …)`	Processes extra data to initialize combination map, if needed.
`deserialize(obj, data)`	Construct a reduction object from serialized reduction object.
`post_combine(…)`	Perform post-combination processing.
`convert(red_obj, out)`	Converts reduction to output object.

RedObj Functions

`void reset()`	Reset reduction object.
`bool trigger()`	Set trigger function for early emission.

Support and Contact Information

- For further questions and iNFORMER Support, please send a mail to

  informer@RNET-Tech.com