Achieving Performance and Programmability for Data-Intensive Applications with Reduction-based APIs

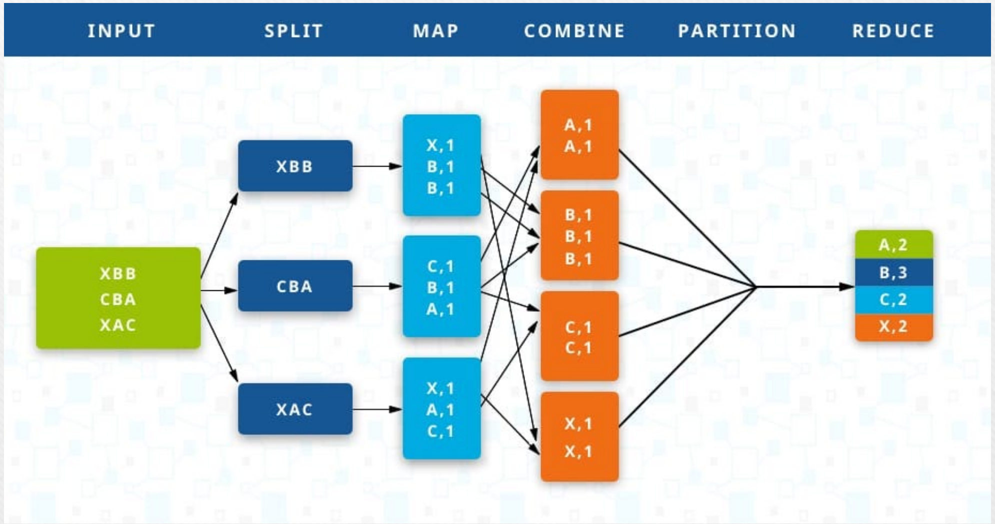
Gagan Agrawal

Augusta University, GA

Background

The success of MapReduce in the past 2 decades

- Supports a wide range of applications
- Optimized for large input size and distributed environment



MapReduce Workflow [1]

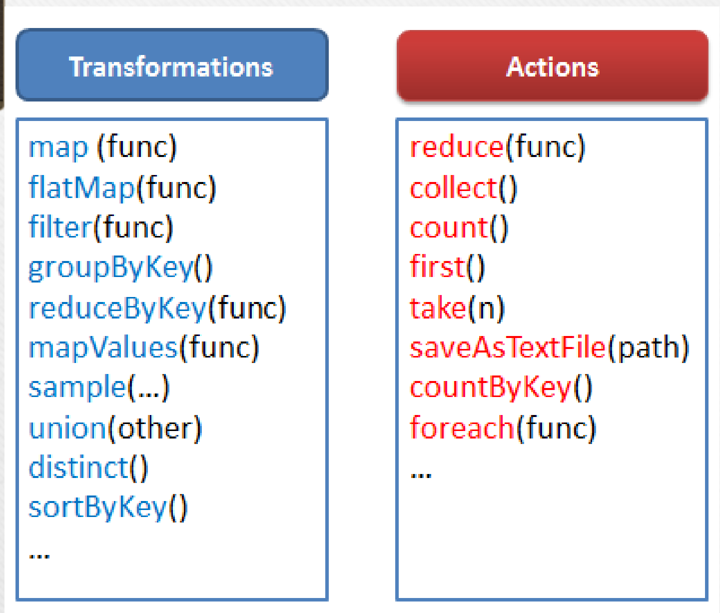
Following the pattern-based API design many works have been done:

- HaLoop, MapReduce-MPI for Graph, Disco, Mariane
- **Smart**
- **Spark**

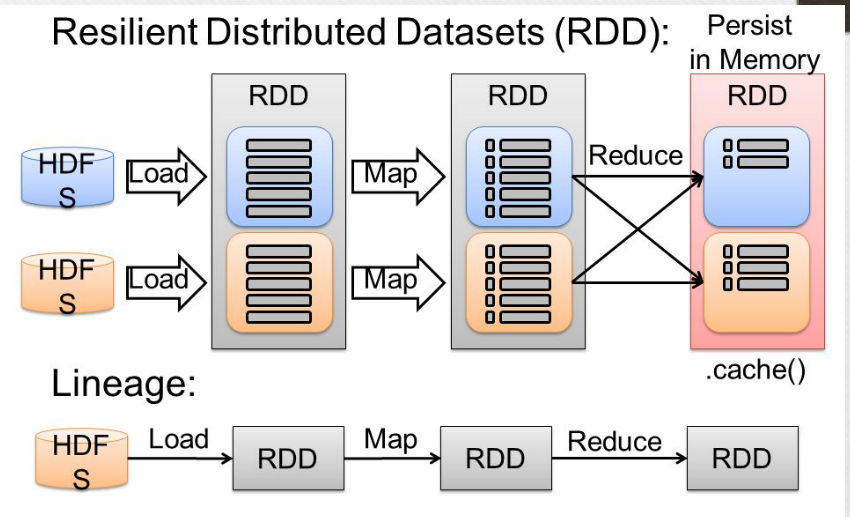
Background

Spark :

- Interfaces: Transformations & Actions
- Data Structure: Resilient Distributed Datasets
- Implementations: Scala
- Advantages: Fast, In-memory, Expressive, Better Resilience Control



Transformations and Actions in Spark [2]



RDD[3]

Motivation

- The goal in this work has been on programmability
- Parallel programming can be done by many
- What about performance?
  - MPI ecosystem focused almost entirely on performance
- Common wisdom is Choose One!
- Can we achieve both performance and programmability?

Outline

- Design of Reduction Based MapReduce Variants
  - **Achieving Performance and Programmability**

- Implementation of Systems:
  - **Smart: An Efficient In-Situ Analysis Framework**
  - **Smart Streaming: A High-Throughput Fault-tolerant Online Processing System**
  - **A Pattern-Based API for Mapping Applications to a Hierarchy of Multi-Core Devices**

I: API Design

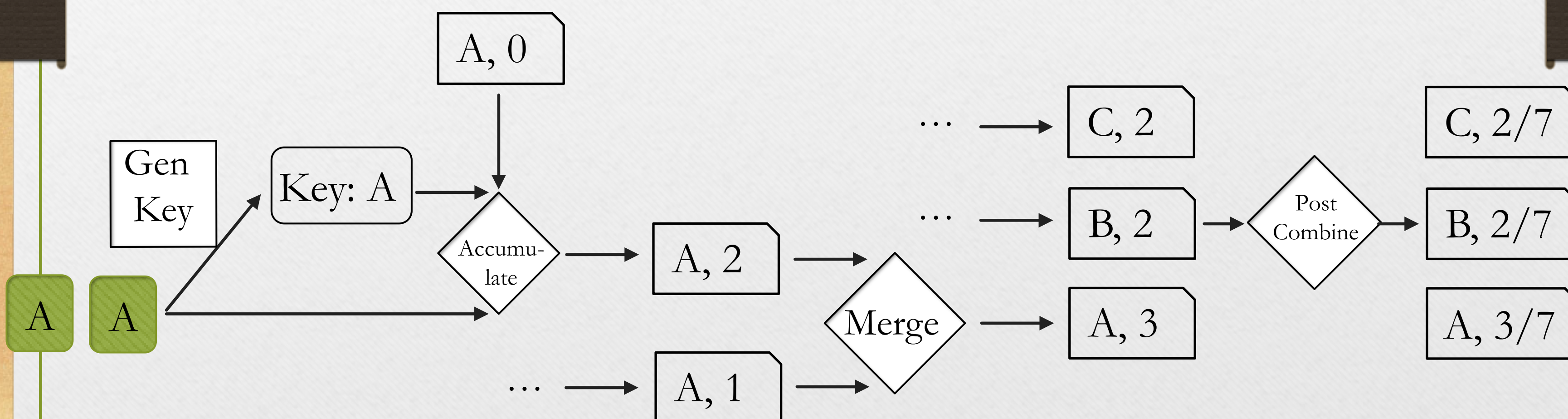
Reduction Object API:

Generate_key: <Input Type> → k

Accumulate: k, <Input Type> × k, list(<v>) → k, list(<v>)

Merge: k, list(<v>) × k, list(<v>) → k, list(<v>)

Post_combine: k, list(<v>) → k, list(<v>)



I: Trade-offs

Programmability:

- **Strict Constraints**

  Merge: strict binary, associative, commutative ↔ Reduce: N-ary

- **Fewer lines of code, less functionality overlap, and directly reuse Map-Reduce code**

```
// Combine
void combine(int key, Head& iter) override {
  double cluster[NUM_DIM]={0,0,...,0};
  double weight=0;

  do{
    ClusterWritable cluster_obj.read(iter.get());

    for (int i = 0; i NUM_DIM; ++i)
      cluster[i]+=cluster_obj.coordinate[i];

    weight+=cluster_obj.weight;
  }while(iter.next());

  ClusterWritable output(cluster,weight);
  this->output(key, output);
}

// Reduce
void reduce(int key, Head& iter) override {
  double cluster[NUM_DIM]={0,0,...,0};
  double weight=0;

  do{
    ClusterWritable cluster_obj.read(iter.get());

    for (int i = 0; i NUM_DIM; ++i)
      cluster[i]+=cluster_obj.coordinate[i];

    weight+=cluster_obj.weight;
  }while(iter.next());

  for (int i = 0; i NUM_DIM; ++i)
    cluster[i]/=weight;

  ClusterWritable output(cluster,weight);
  this->output(key, output);
}
```

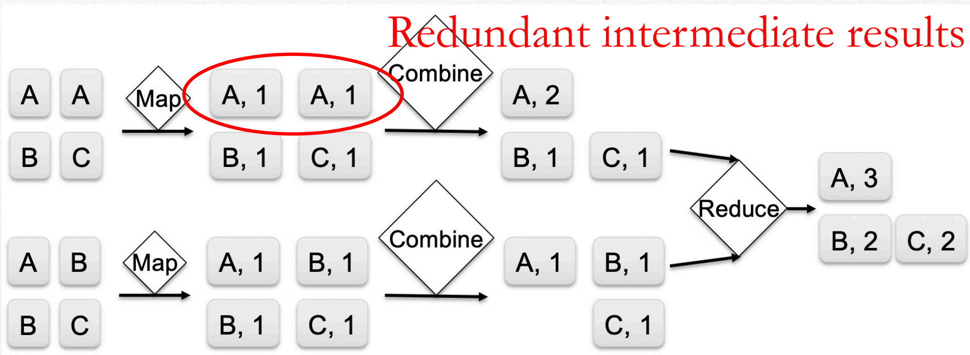
Map-Reduce API

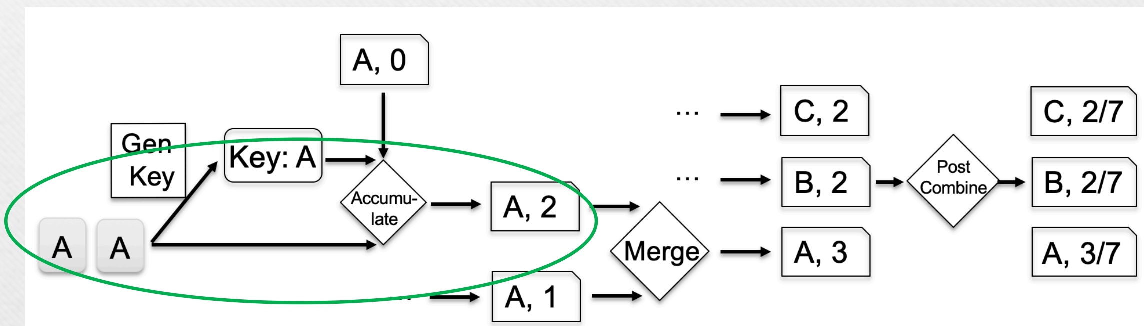
```
// Accumulate chunk on sum and size of red_obj.
void accumulate(Chunk& chunk, double* input,
    unique_ptr<RedObj>& red_obj) override {
  //The input data points are not equally weighted
  double weight=input[chunk.start+NUM_DIM]
  //compute and accumulate weighted coordinates;
  for (int i = 0; i < NUM_DIM; ++i)
    red_obj->sum[i] += chunk[i]*weight;
  //accumulate weight
  red_obj->size+=weight;
}

// Merge red_obj into com_obj on sum and size.
void merge(const RedObj& red_obj, unique_ptr<RedObj
    >& com_obj) override {
  for (int i = 0; i < NUM_DIM; ++i)
    com_obj->sum[i] += red_obj->sum[i];
  com_obj->size += red_obj->size;
}
```

Reduction Object API

I: Trade-Offs

Efficiency: better memory efficiency & locality:





Input values are instantly accumulated to local accumulators

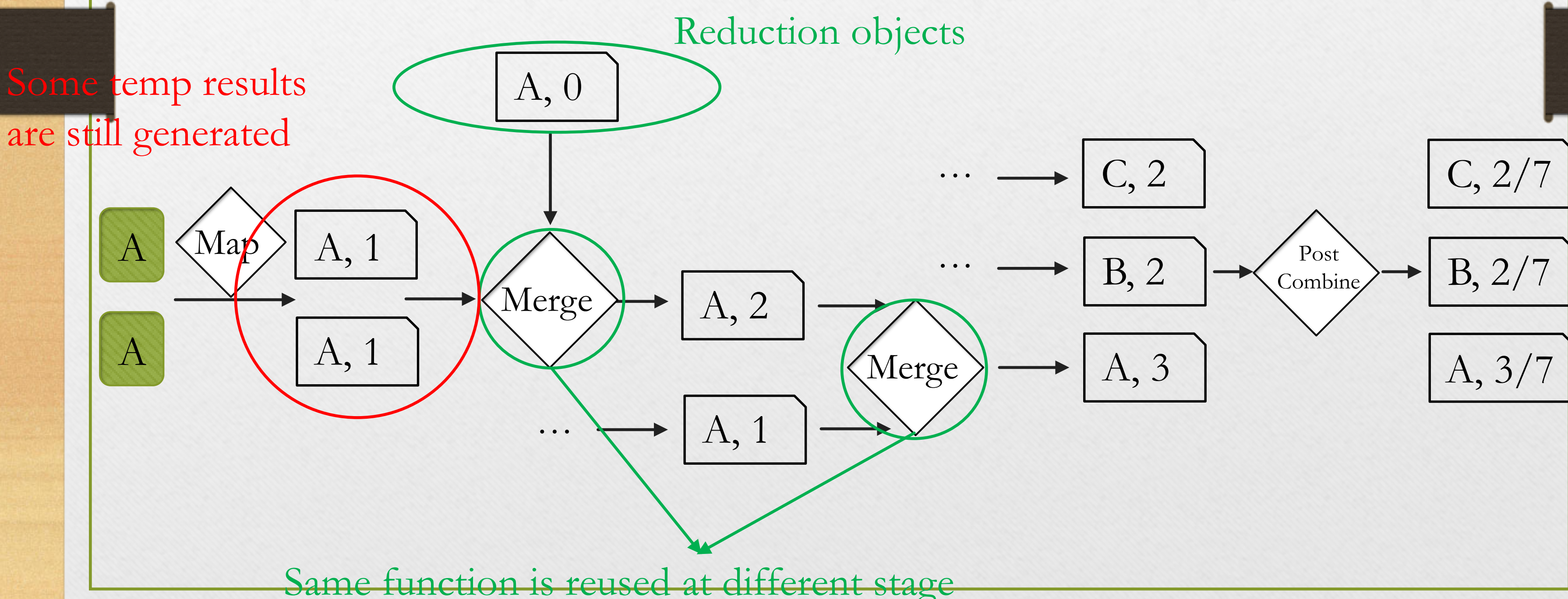
Yet Another API

MR-like API:

Map: <Input Type> → list(<k, v>)

Merge: k, list(<v>) × k, list(<v>) → k, list(<v>)

Post_combine: k, list(<v>) → k, list(<v>)



Effective lines of code:

	MR API	RO API	MR-like API
K-means	36	19	19
SVM	44	28	21
Linear Regression	43	26	19
Logistic Regression	36	17	17

Outline

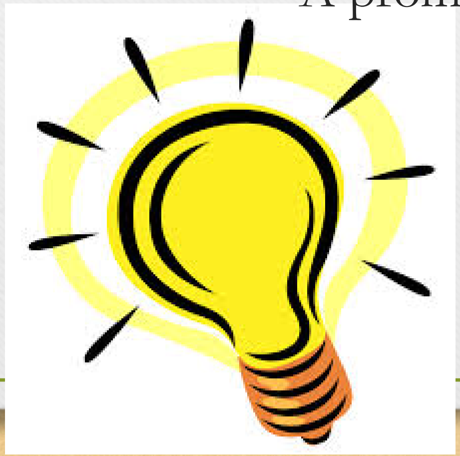
- Design of Reduction Based MapReduce Variants
  - **Achieving Performance and Programmability**
- Implementation of Systems:
  - **Smart: An Efficient In-Situ Analysis Framework**
  - **Smart Streaming: A High-Throughput Fault-tolerant Online Processing System**
  - **A Pattern-Based API for Mapping Applications to a Hierarchy of Multi-Core Devices**

In-Situ Analytics on Simulations

- Driver: Cannot Write all Data to Disk
  - Computation/ I/O Ratio has been changing
- In-Situ Algorithms
  - Implemented with low-level APIs like OpenMP/MPI
  - Manually handle all the parallelization details
- Motivation
  - Can the applications be mapped more easily to the platforms for in-situ analytics?
  - Can the offline and in-situ analytics code be (almost) identical?

Opportunity

- Explore the **Programming Model Level** in In-Situ Environment
  - Between application level and system level
  - Hides all the parallelization complexities by simplified API
  - A prominent example: MapReduce



In Situ + MapReduce?

Challenges

- Hard to Adapt MR to In-Situ Environment
  - MR is not designed for in-situ analytics
- Mismatches
  - Programming View Mismatch
  - Memory Constraint

Programming View Mismatch

- Scientific Simulation
  - Parallel programming view
  - Explicit parallelism: partitioning, message passing, and synchronization
- MapReduce
  - Sequential programming view
  - Partitions are transparent
- Need a Hybrid Programming View
  - Exposes partitions during data loading
  - Hides parallelism after data loading

Memory Constraint Mismatch

- MR is Often Memory/Disk Intensive
  - Map phase creates intermediate data
  - Sorting, shuffling, and grouping do not reduce intermediate data at all
  - Local combiner cannot reduce the peak memory consumption (in map phase)
- Need Alternate MR API
  - Avoids key-value pair emission in the map phase
  - Eliminates intermediate data in the shuffling phase

Bridging the Gap

- Addresses All the Mismatches
  - Loads data from (distributed) memory, even without extra memcpy in time sharing mode
  - Presents a hybrid programming view
  - High memory efficiency with alternate API
  - Implemented in C++11, with OpenMP + MPI

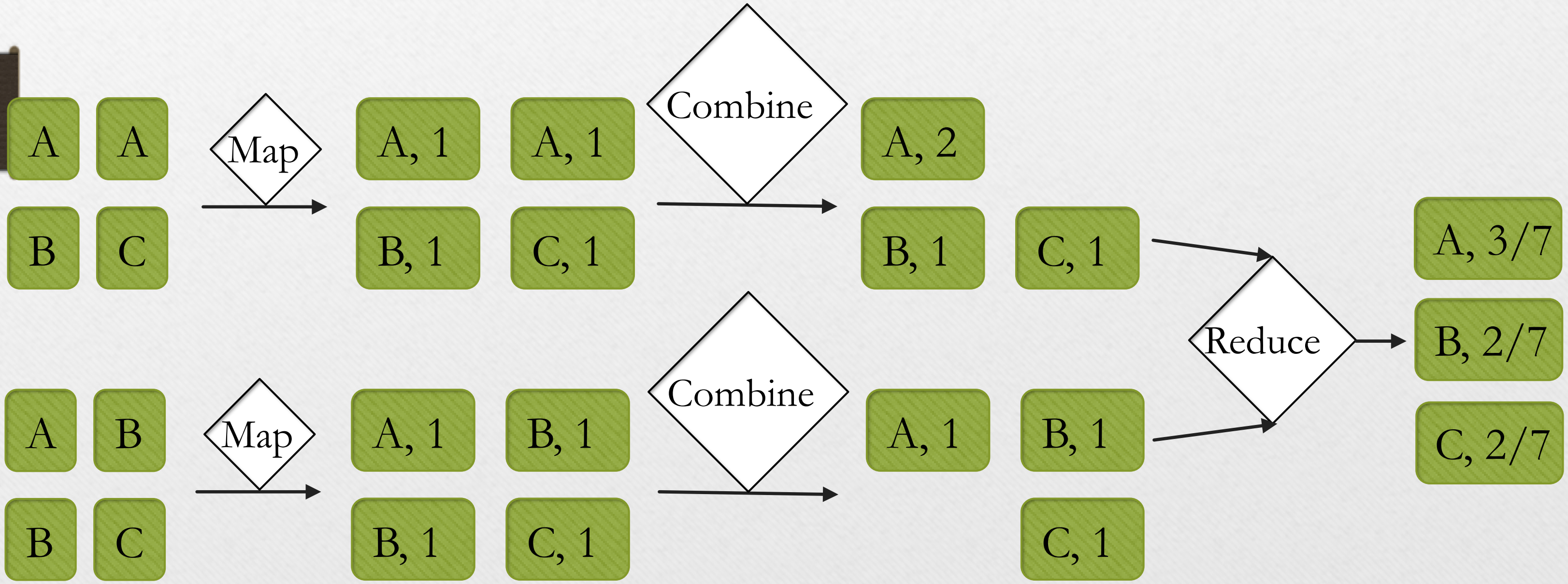
API Design

Original Map-Reduce API:

Map: <Input Type> → list<k, v>

Combine: k, list<v> → list<k, v> (Local)

Reduce: k, list<v> → list<k, v>



API Design

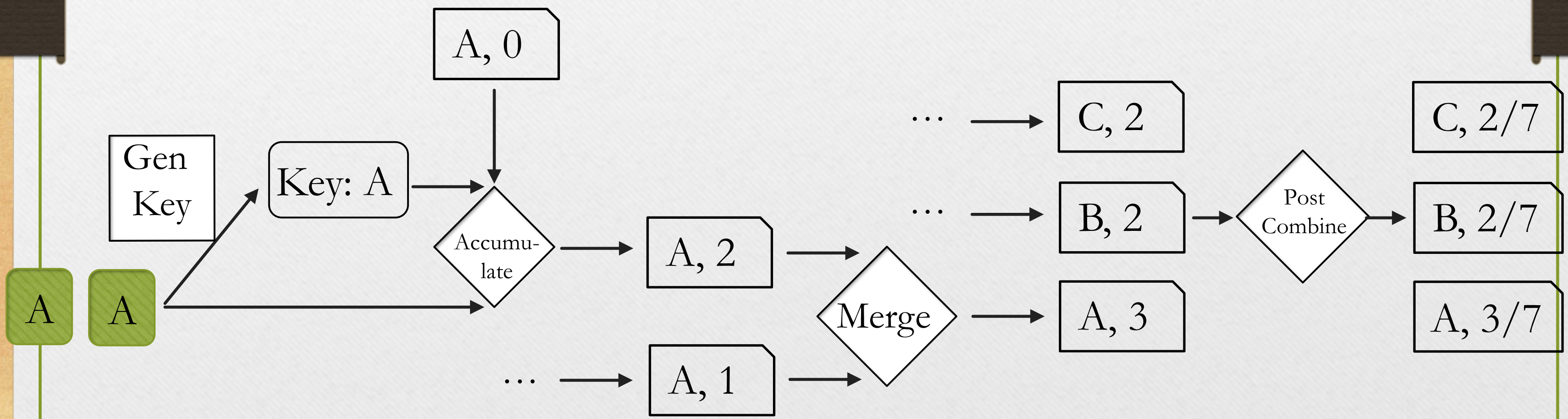
Reduction Object API:

Generate_key: <Input Type> →list(<k>)

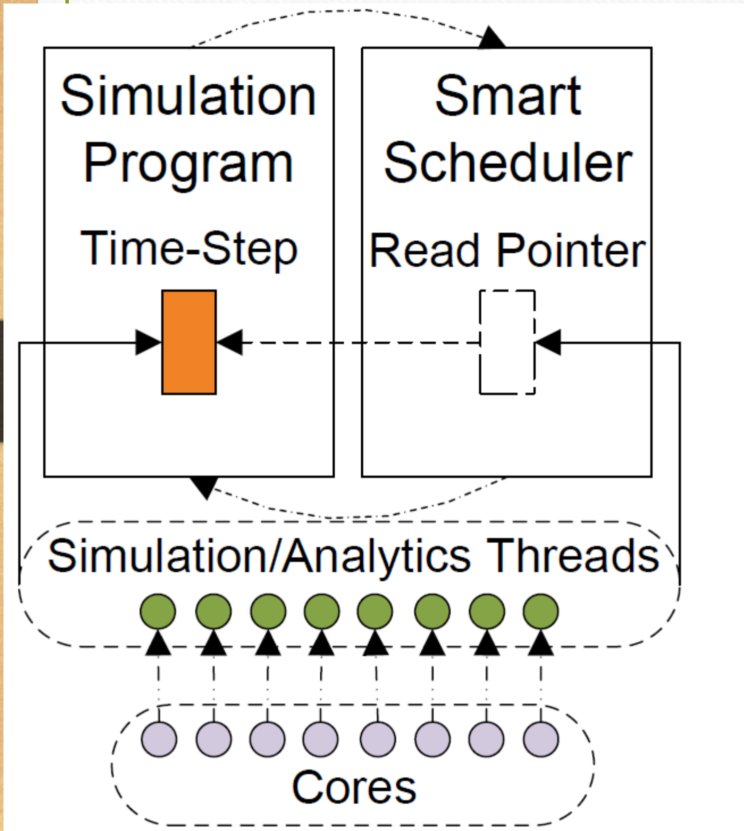
Accumulate: k, <Input Type> × k, list(<v>) → k, list(<v>)

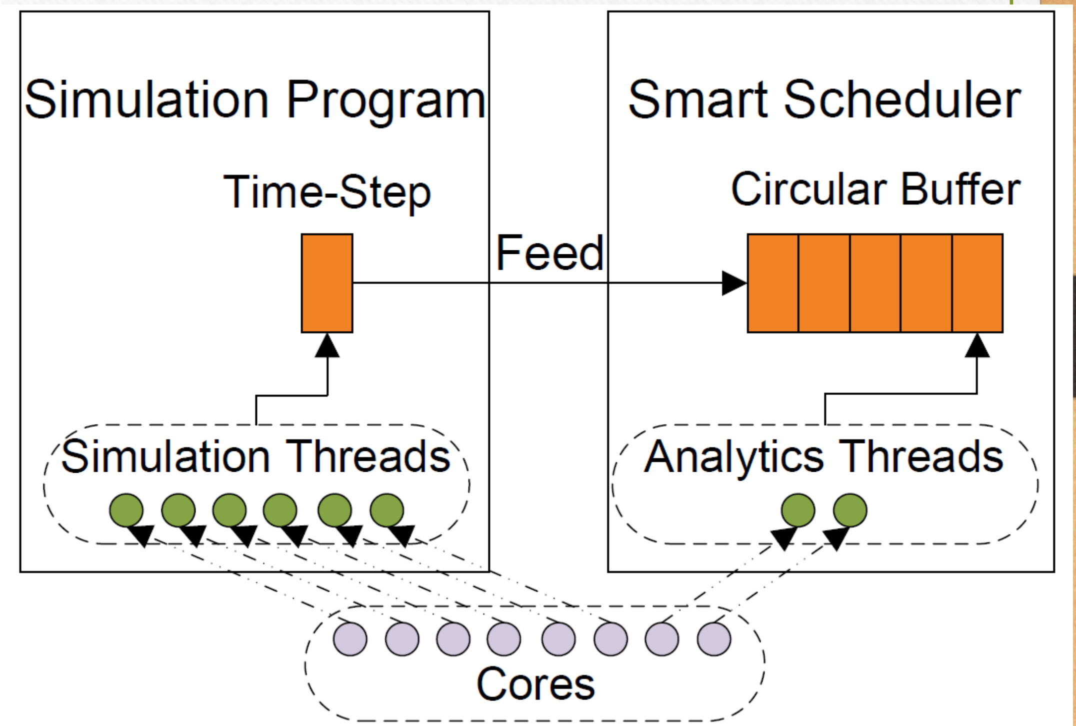
Merge: k, list(<v>) × k, list(<v>) → k, list(<v>)

Post_combine: k, list(<v>) → k, list(<v>)



Two In-Situ Modes

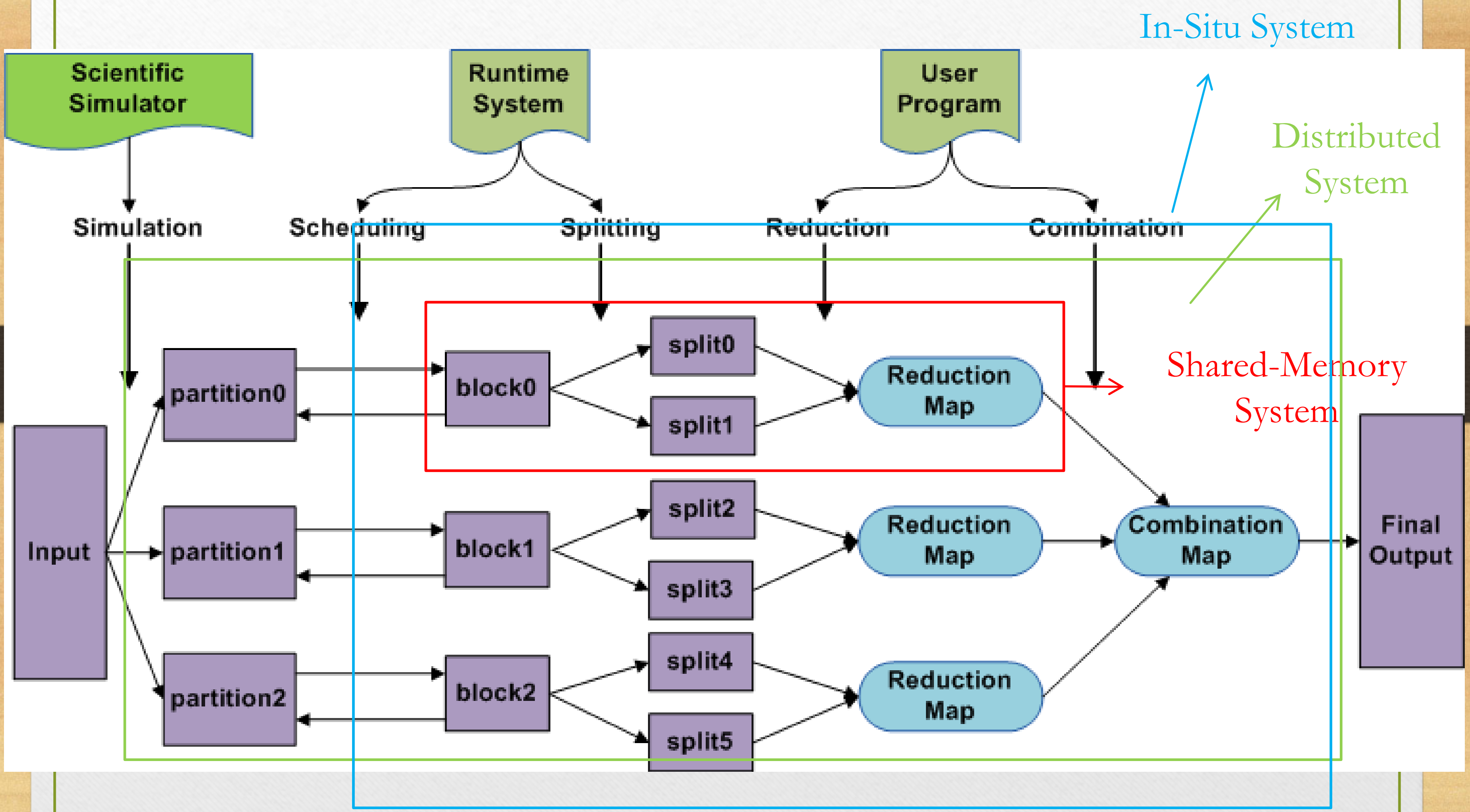




Space Sharing Mode:
Enhances resource utilization when simulation reaches its scalability bottleneck

Time Sharing Mode:
Minimizes memory consumption

System Overview



Ease of Use

- Launching Smart
  - No extra libraries or configuration
  - Minimal changes to the simulation code
  - Analytics code remains the same in different modes
- Application Development
  - Define a reduction object
  - Derive a Smart scheduler class
    - *gen_key(s)*: generates key(s) for a data chunk
    - *accumulate*: accumulates data on a reduction object
    - *merge*: merges two reduction objects

Launching Smart in Space Sharing Mode

Listing 2: Launching Smart in 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);
    }
}
```

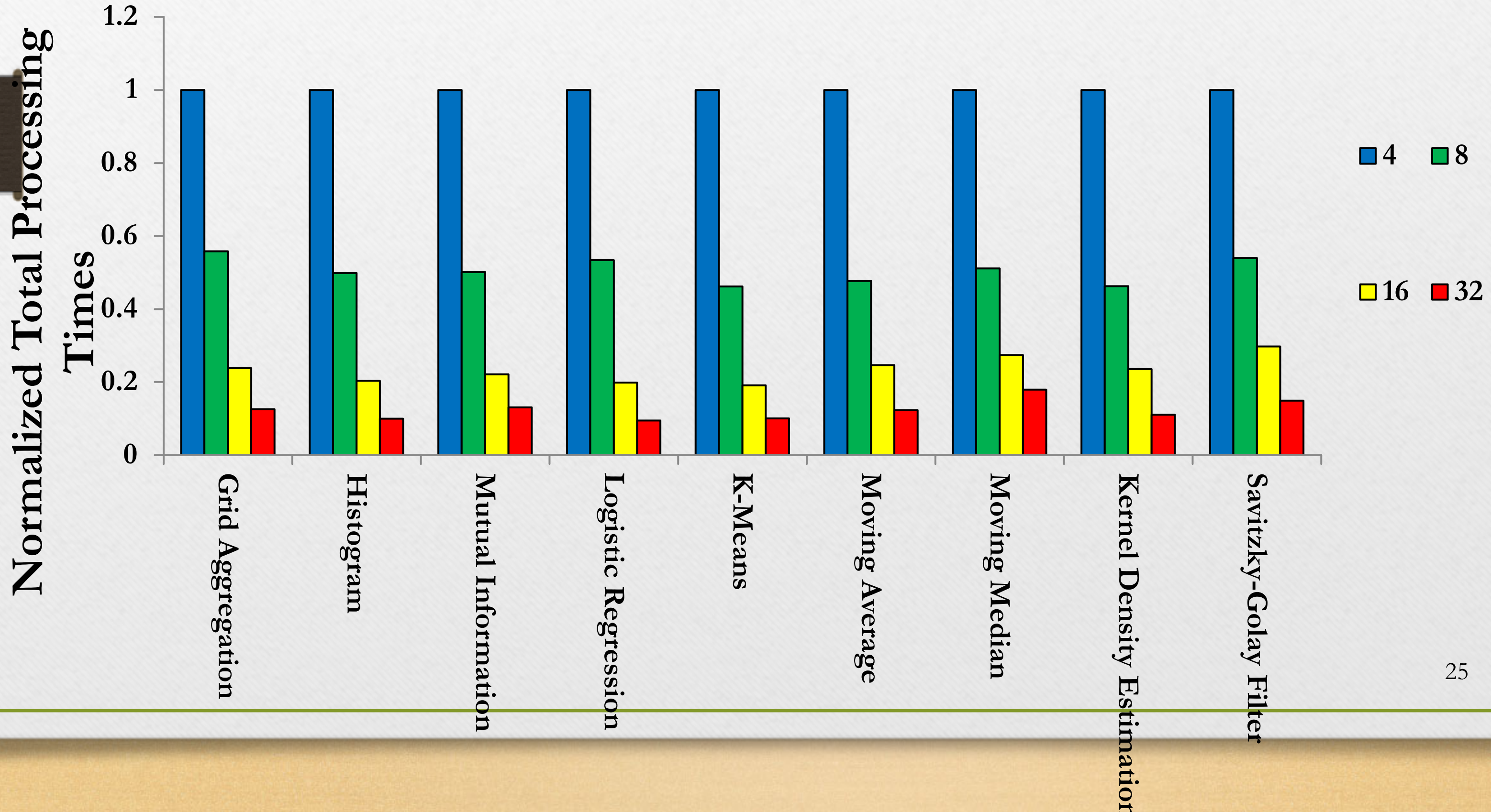
Launching Smart in Time Sharing Mode

Listing 1: Launching Smart in 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);
}
```

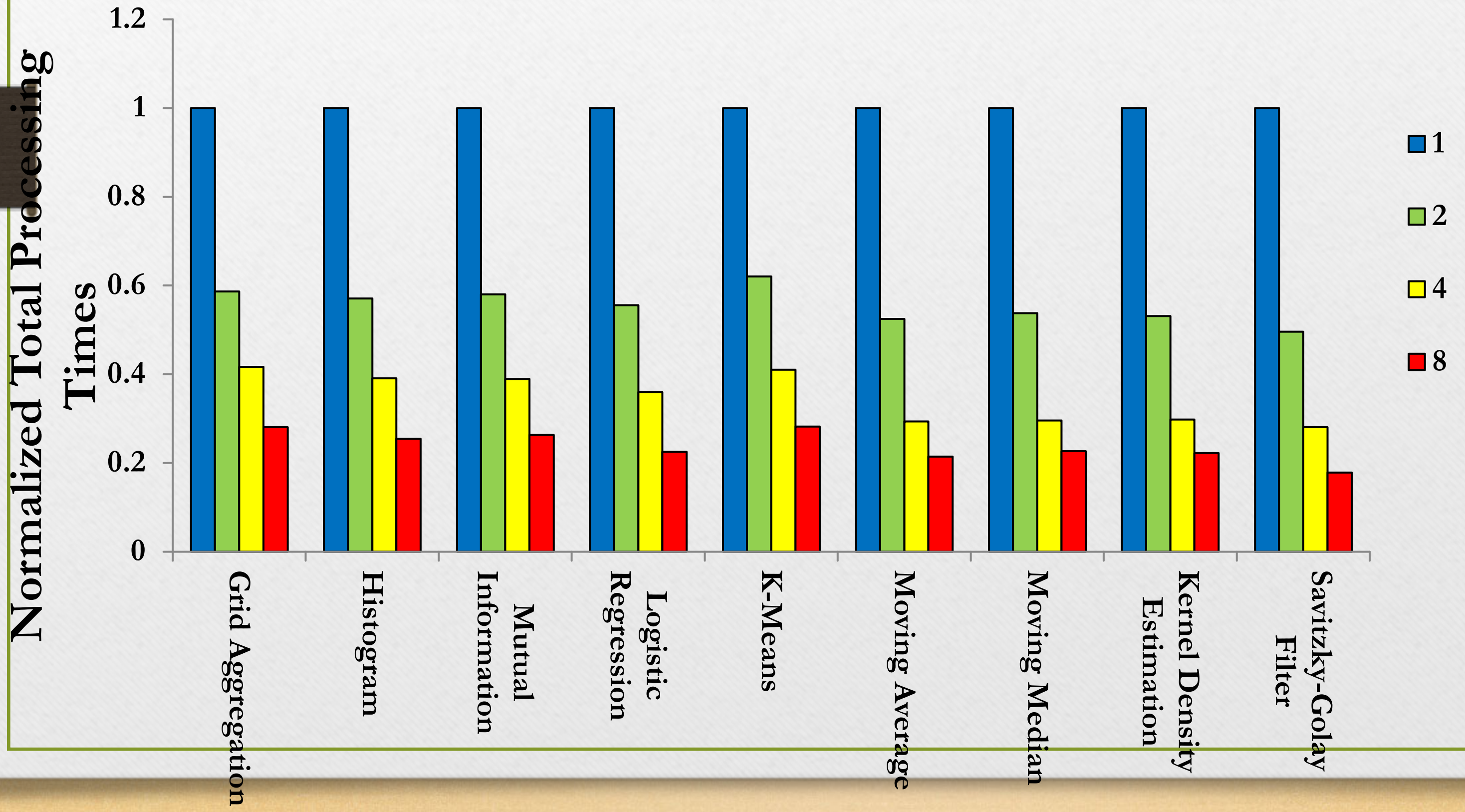
Node Scalability

- Setup
  - 1 TB data output by Heat3D; time sharing; 8 cores per node
  - 4-32 nodes



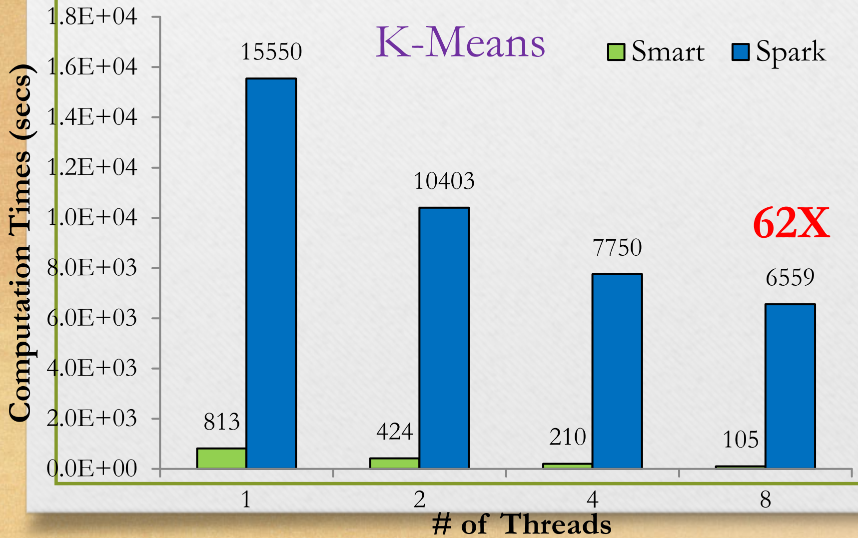
Thread Scalability

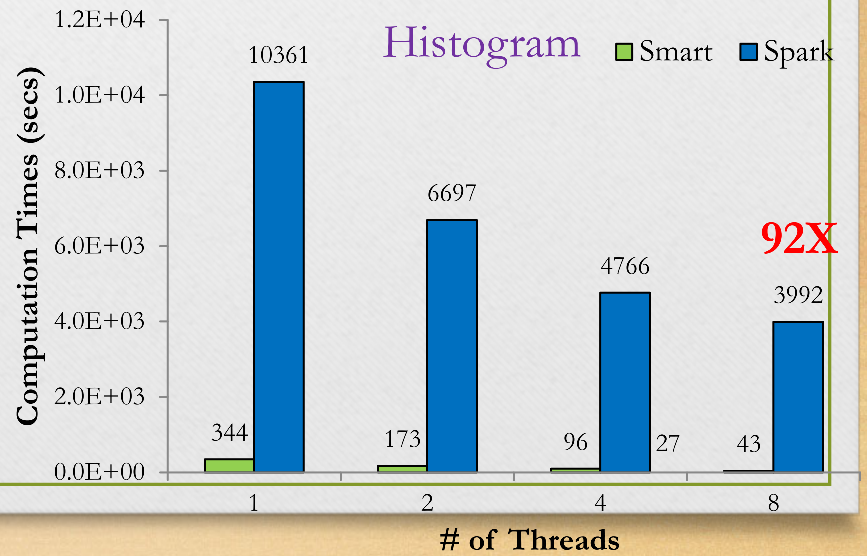
- Setup
  - 1 TB data output by Lulesh; time sharing; 64 nodes
  - 1-8 threads per node



Smart vs. Spark

- To Make a Fair Comparison
  - Bypass programming view mismatch
    - Run on an 8-core node: multi-threaded but not distributed
  - Bypass memory constraint mismatch
    - Use a simulation emulator that consumes little memory
  - Bypass programming language mismatch
    - Rewrite the simulation in Java and only compare computation time
- 40 GB input and 0.5 GB per time-step



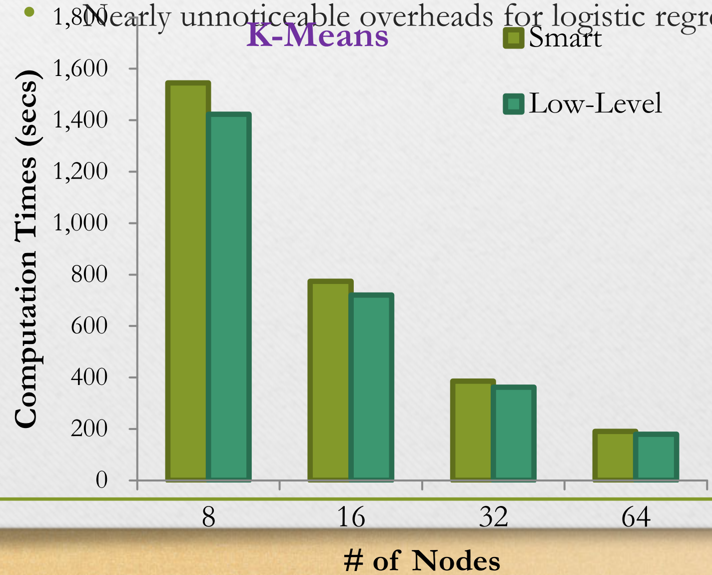


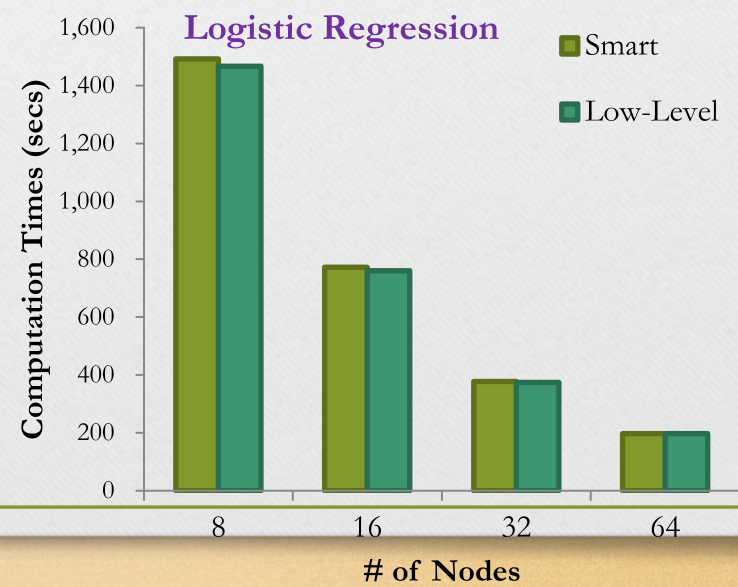
Smart vs. Spark (Cont'd)

- Faster Execution
  - Spark 1) emits intermediate data, 2) makes immutable RDDs, and 3) serializes RDDs and sends them through network even in the local mode
  - Smart 1) avoids intermediate data, 2) performs data reduction in place, and 3) takes advantage of shared-memory environment (of each node)
- Greater (Thread) Scalability
  - Spark launches extra threads for other tasks, e.g., communication and driver's UI
  - Smart launches no extra thread
- Higher Memory Efficiency
  - Spark: over 90% of 12 GB memory
  - Smart: around 16 MB besides 0.5 GB time-step

Smart vs. Low-Level Implementations

- Setup
  - Smart: time sharing mode; Low-Level: OpenMP + MPI
  - Apps: K-means and logistic regression
  - 1 TB input on 8–64 nodes
- Programmability
  - 55% and 69% parallel codes are either eliminated or converted into sequential code
- Performance
  - Up to 9% extra overheads for k-means
  - Nearly unnoticeable overheads for logistic regression





Outline

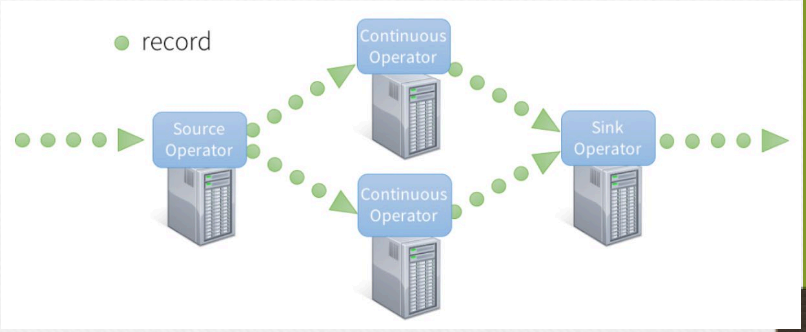
- Design of Reduction Based MapReduce Variants
    - **Achieving Performance and Programmability**
- Implementation of Systems:
    - **Smart: An Efficient In-Situ Analysis Framework**
    - **Smart Streaming: A High-Throughput Fault-tolerant Online Processing System**
    - **A Pattern-Based API for Mapping Applications to a Hierarchy of Multi-Core Devices**

II: Stream Processing

Continuous Stream Processing:

- One record at a time
- Low latency

e.g. Storm, Flink, Samza …

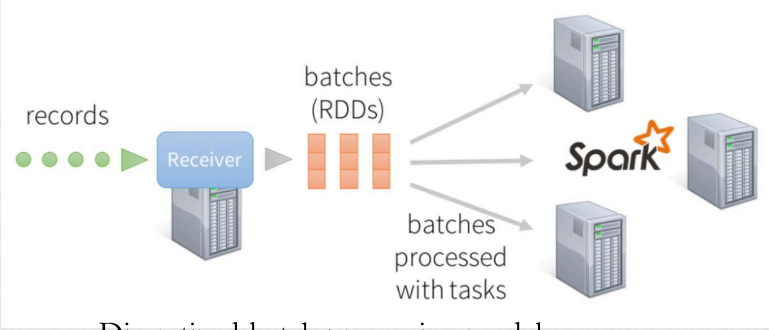


Continuous stream processing model [3]

Discrete Stream Processing:

- A micro-batch of records
- Coherent with batch jobs
- Locality

e.g. Spark Streaming

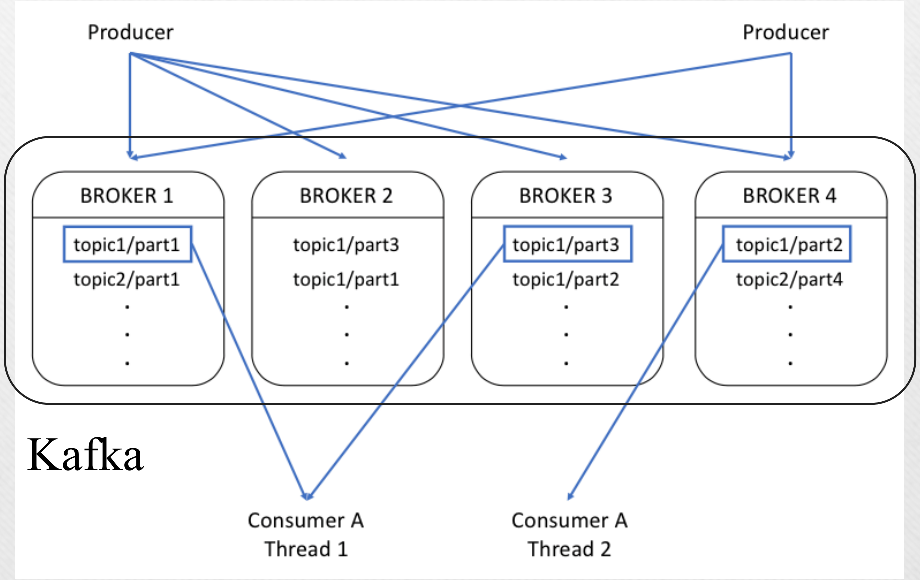


Discretized-batch processing model

II: Kafka

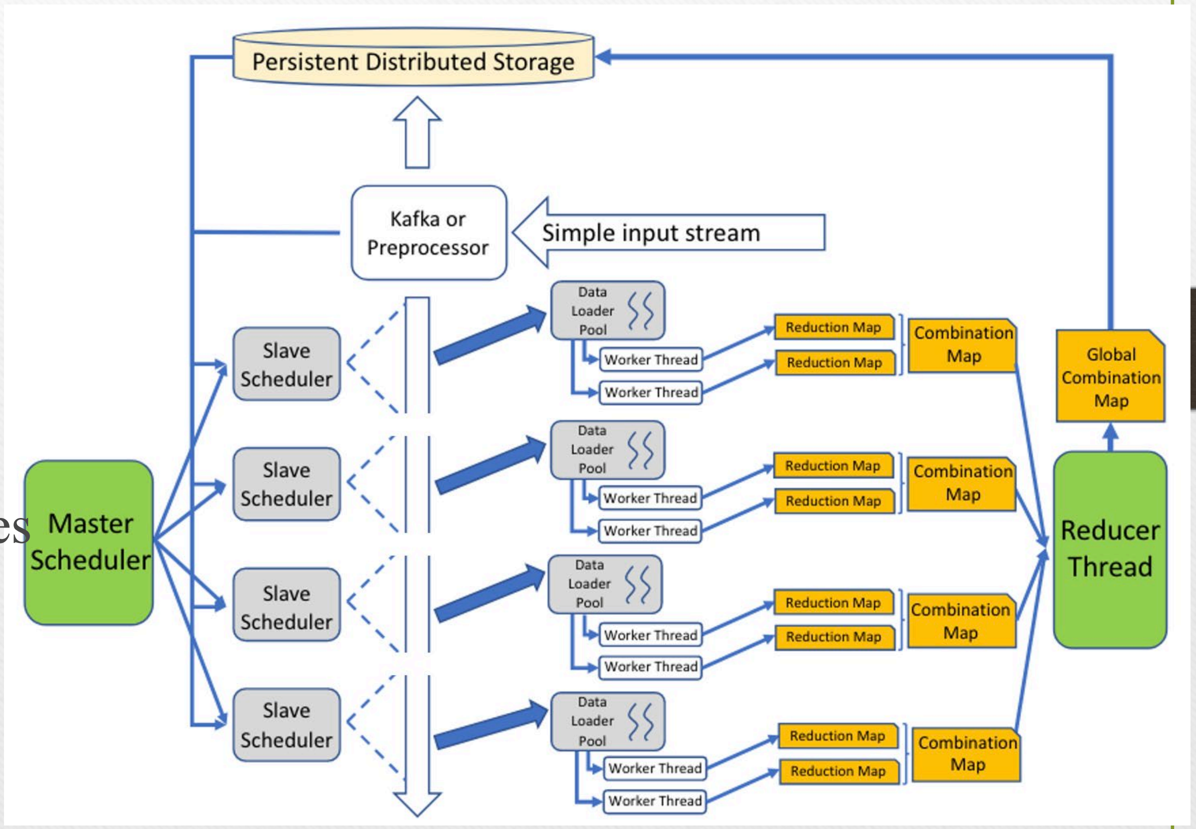
A part of data pipeline, commonly used to ingest events and act as source for downstream processing or ETL

- Topic Based
- Scalability:
  - Partitioned Topics
  - Parallel Consuming
- Fault-tolerance:
  - Duplication
  - Checkpointing
- Interface: cppkafka



Workflow:

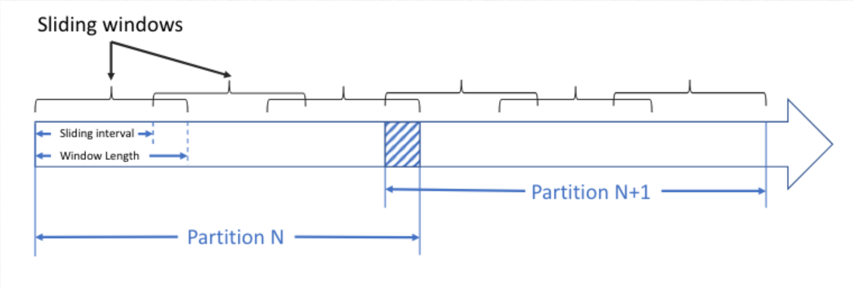
- Master Node:
  - Get checkpoint
  - Schedule workload
- Worker Node:
  - Fetch messages
  - Accumulate messages
- Master Node:
  - Final Reduction
  - Commit Checkpoint



System Design

API:

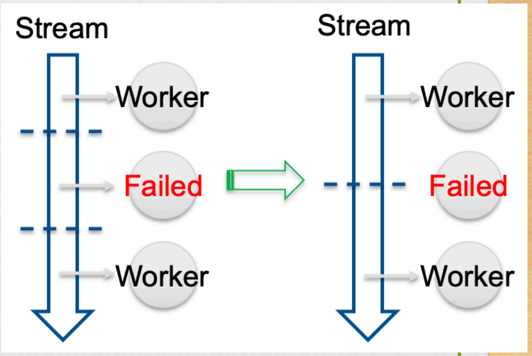
- Reduction-Object based abstraction
- Micro-batch size control
- Sliding window width control



Sliding window processing

Fault-tolerance:

- A heart-beat mechanism implemented using OpenMP/MPI
- On-disk checkpointing of progress and state
- Dynamic load re-balancing upon failure/suspension of worker



Implementations

MPI and OpenMP Collaboration:

- Using Mvapich2: For HPC environment and error handling
- MPI Run-through-stabilization: *--disable-auto-cleanup*
- Process Communication: Non-blocking, point to point send and receive
- Multi-Threading: 1-2 I/O threads (control, data) and a pool of reduction threads
- Thread-safety: MPI *send* and *receive* commands protected by critical zone

II: Experiments

Environment:

- 32 nodes, each with two quad-core Intel E5640
- 12GB RAM
- 40GB/s Infiniband interconnect
- Kafka on 5 nodes

Throughput:

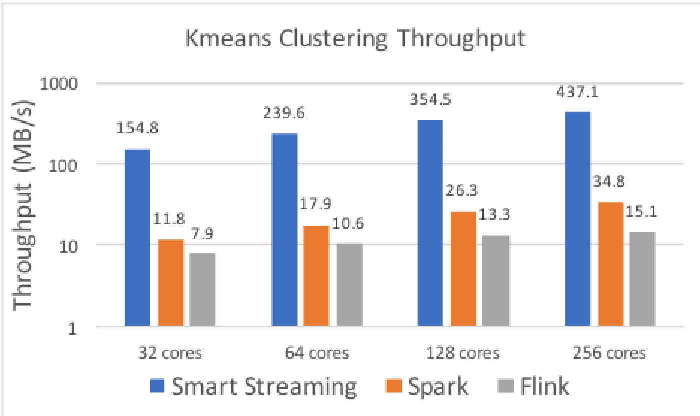
- Ours v.s. Spark v.s. Flink
- Sustainable throughput

Fault-tolerance:

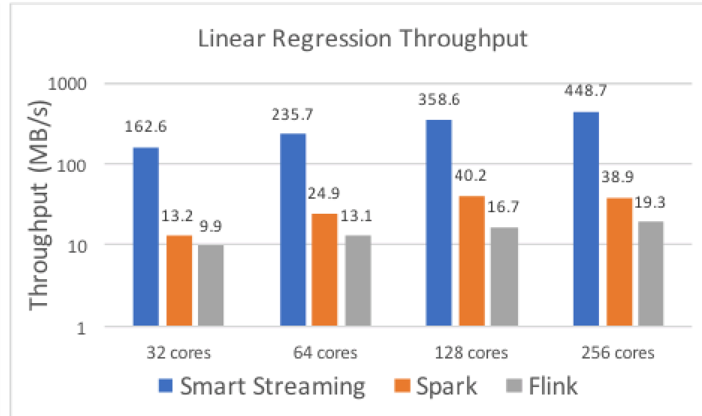
- Ours v.s. Spark
- One worker down

K-means	**Batch Size**: 1s on Spark or equivalent size. **Number of Means**: 8 **Number of Dimensions**: 16 (128 bytes)
Linear Regression	**Batch Size**: 1s on Spark or equivalent size. **Number of Features**: 15 **Lablels**: 1 **Number of Dimensions**: 16 (128 bytes) **Regularization Parameter**: 0.1 **Step Size**: 1.0 **Include Interception**: Yes
Histogram	**Batch Size**: 1s on Spark or equivalent size. **Number of Dimensions**: 1 (8 bytes) **Number of Buckets**: 10
Moving Average	**Number of Dimensions**: 1 (8 bytes) **Window size**: 3s or equivalent size. **Sliding interval**: 1s

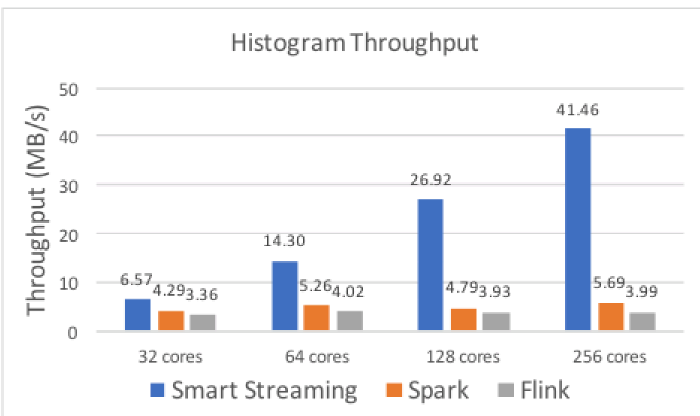
Throughput:



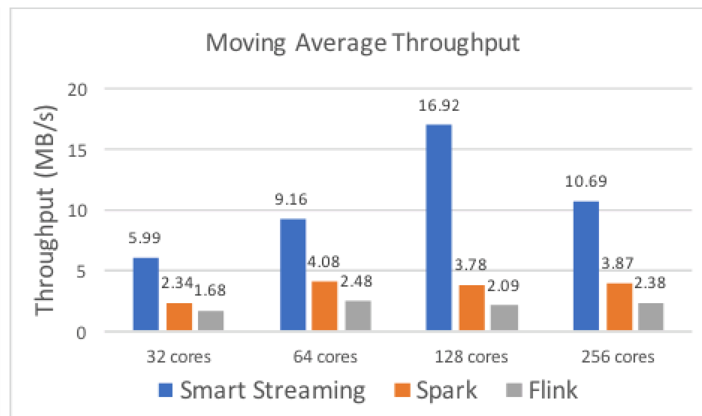
(a) K-Means Throughput



(b) Linear Regression Throughput



(c) Histogram Throughput



(d) Moving Average Throughput

Fault-tolerance: single worker down

(a) K-Means Throughput Loss

(b) Linear Regression Throughput Loss

(c) Histogram Throughput Loss

(d) Moving Average Throughput Loss

Discussions

Throughput:

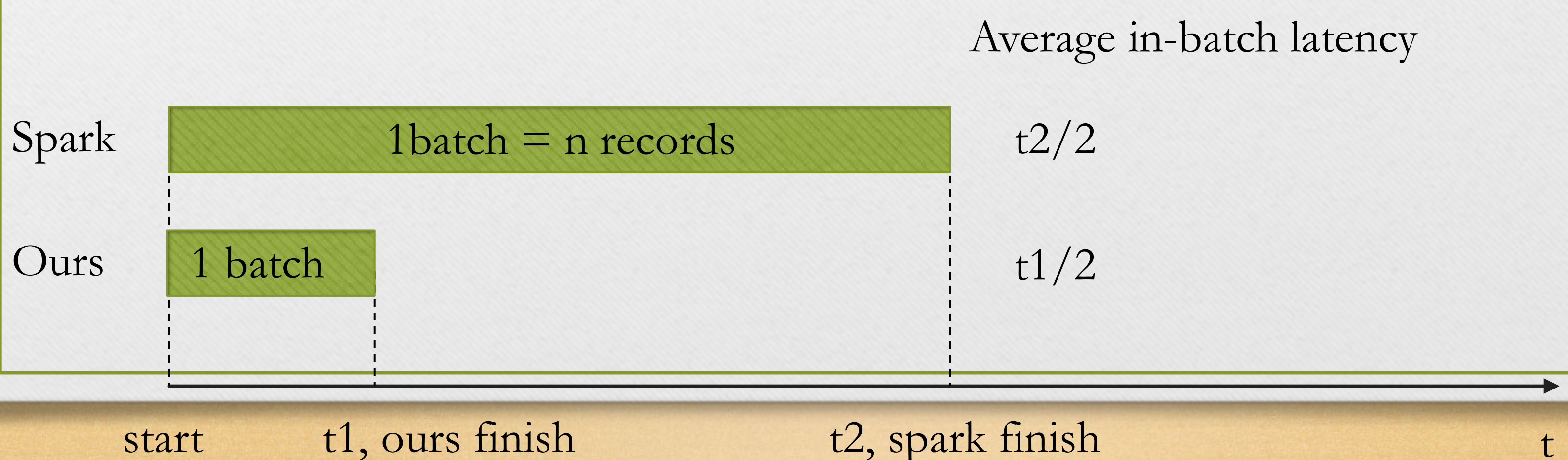
- The throughput of our framework significantly outperforms existing frameworks
- Our framework shows scalability as good as Spark
- In window based applications, throughput is capped by parallelism of Kafka partition

Fault-tolerance:

- The failure detection and load balancing are dynamic and non-blocking
- The throughput loss is roughly propotional to the resource loss
- The failure of master node requires the whole system restart from checkpoint
- When using 64 nodes, the F-T performance can be limited by the parallelism of Kafka partition

Latency:

- With the same microbatch size and smaller batch processing time, our in batch latency < Spark’s



Summary

- Operating over stream data, Kafka support not perfect
- Micro-batch processing with desirable throughput and scalability.
- Efficient failure detection for node failures/suspension.

Outline

- Design of Reduction Based MapReduce Variants
  - **Achieving Performance and Programmability**

- Implementation of Systems:
  - **Smart: An Efficient In-Situ Analysis Framework**
  - **Smart Streaming: A High-Throughput Fault-tolerant Online Processing System**
  - **A Pattern-Based API for Mapping Applications to a Hierarchy of Multi-Core Devices**

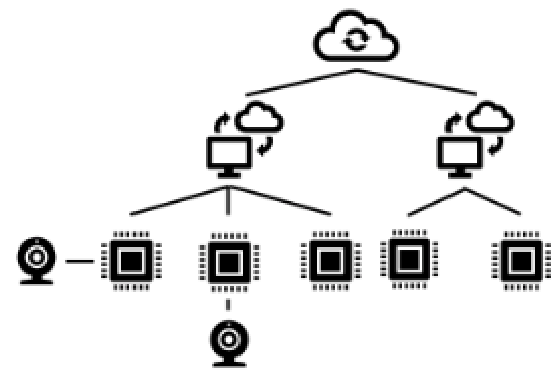
III: IoT and Fog

IoT Devices:

- Limited in clock frequency, memory size, and cache size
- Multi-core CPU structure: RPi 3B+ with a quad core

Hierarchical Structure:

- Devices are of very different processing capabilities
- A vertical hierarchy from sensors to cloud
- Limited bandwidth
- Latency sensitive applications

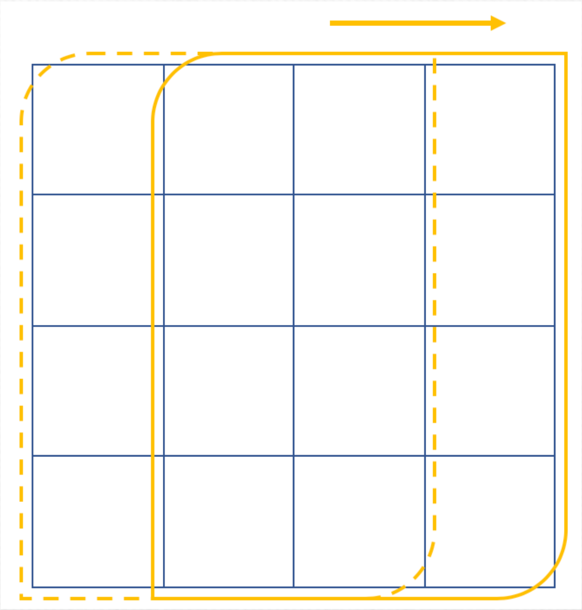


III: Motivations

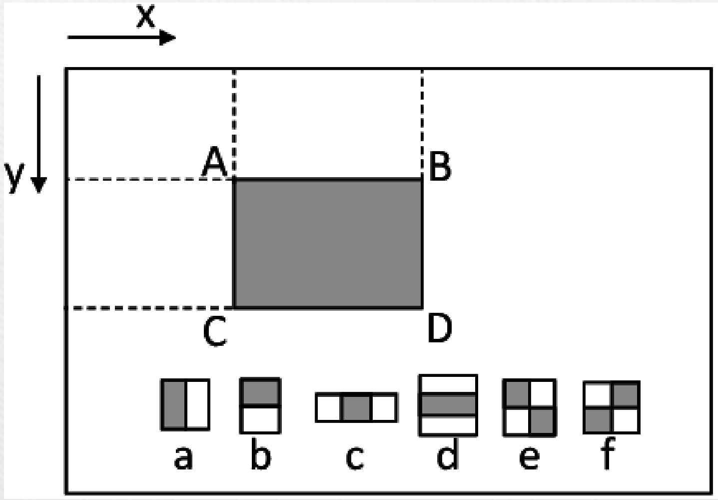
- Extend the Reduction Object APIs to broader types of applications.
- IoT and Fog computing requires high CPU/memory efficiency.
- Heterogenous IoT devices and requires special optimizations.
- We want to offload the computation to the whole network.

III: Application

Basic Concepts:

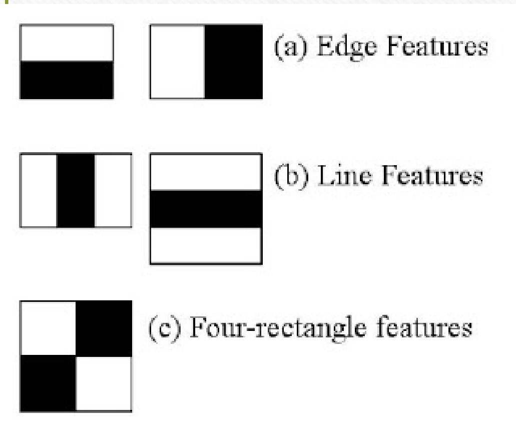


2-D Sliding Window

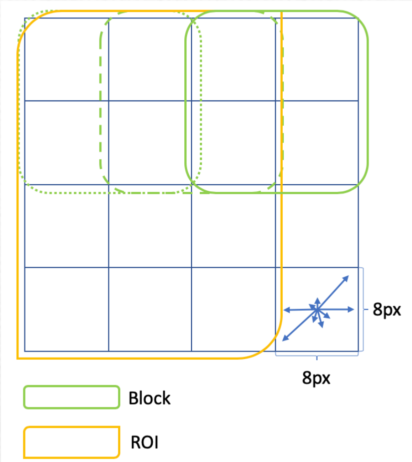


2-D Integral Image

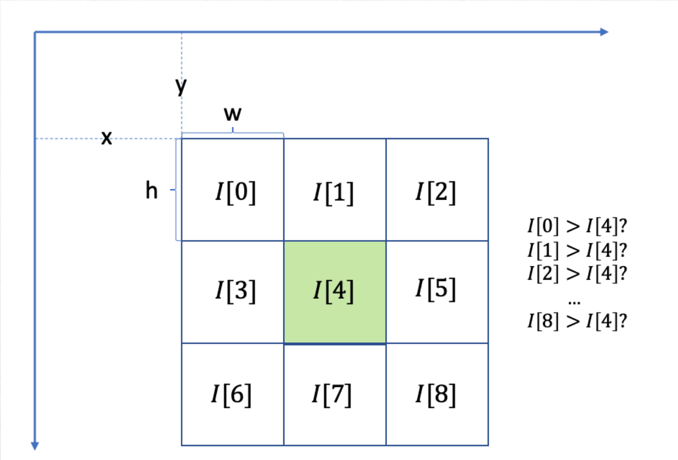
Feature Selection:



Haar

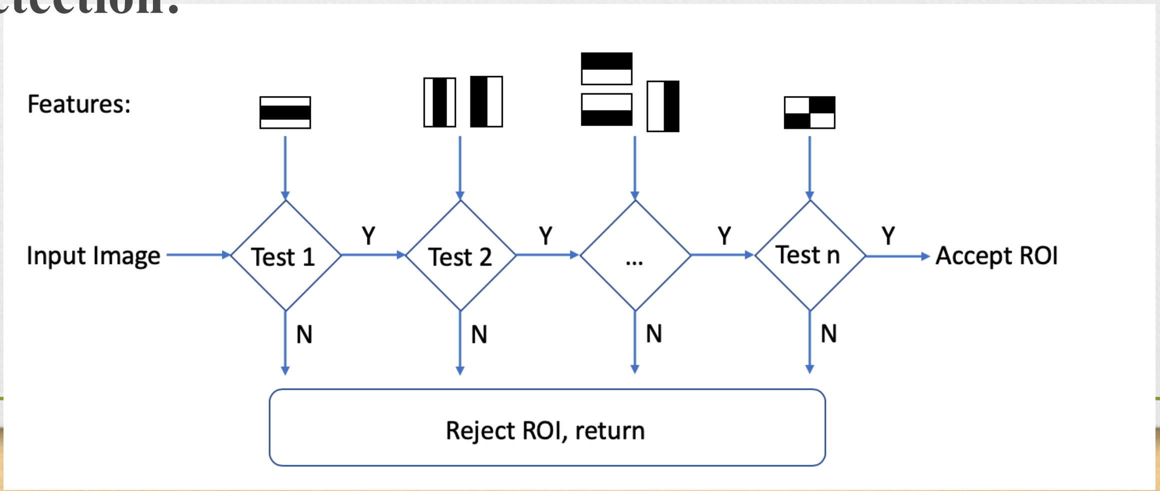


HOG



LBP

Cascaded Detection:



III: APIs

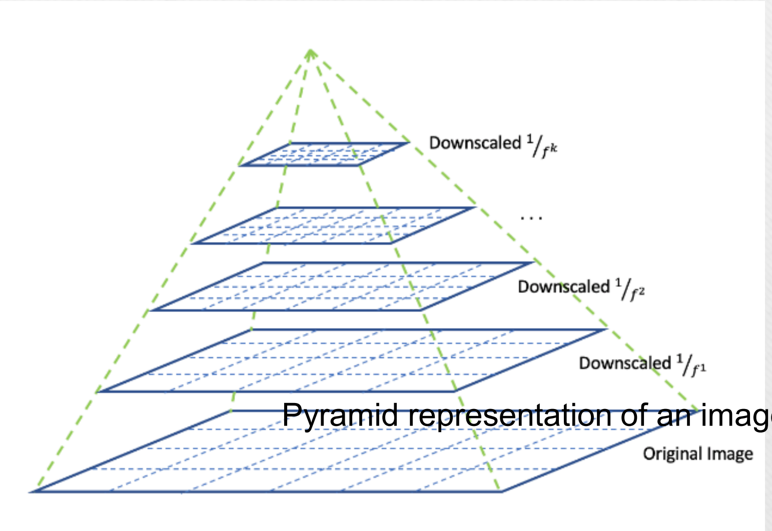
API Design:

- Reduction-Object
- Image Processing: pyramid, sliding window, convolution
- User defined transformation

genKey	Generate one (or multiple) key(s) for each input element
accumulate	Accumulate an input pair to the accumulator
merge	Merge two accumulators with the same Key
window	Perform operation on a sequence of windows
pyramid	Generate a pyramid representation
convolution	Perform a convolution on the given coordinate

Load Balancing For Multi-scale Detection

- The detection is often on multiple scales of one image
- The load for each level of scale can be calculated
- The work can be off-loaded to different scales of network after a load test.

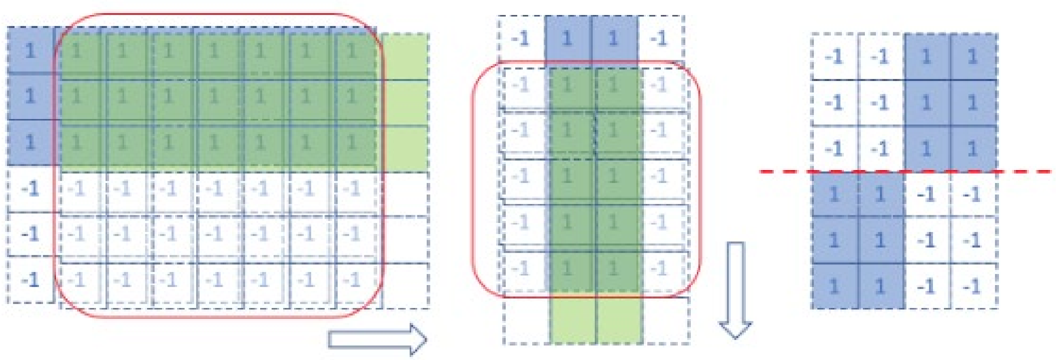


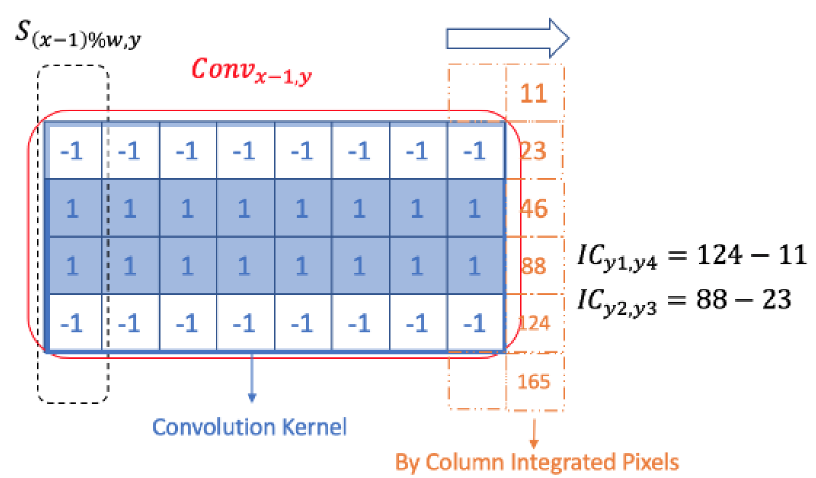
Pyramid representation of an image

III: Optimizations

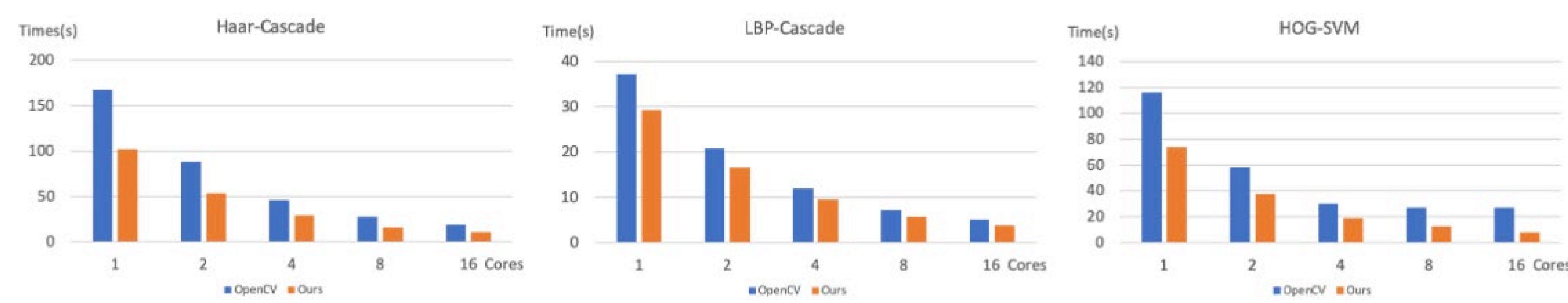
Re-use of Partial Results:

- For Haar-like feature selection
- Result reuse between adjacent windows



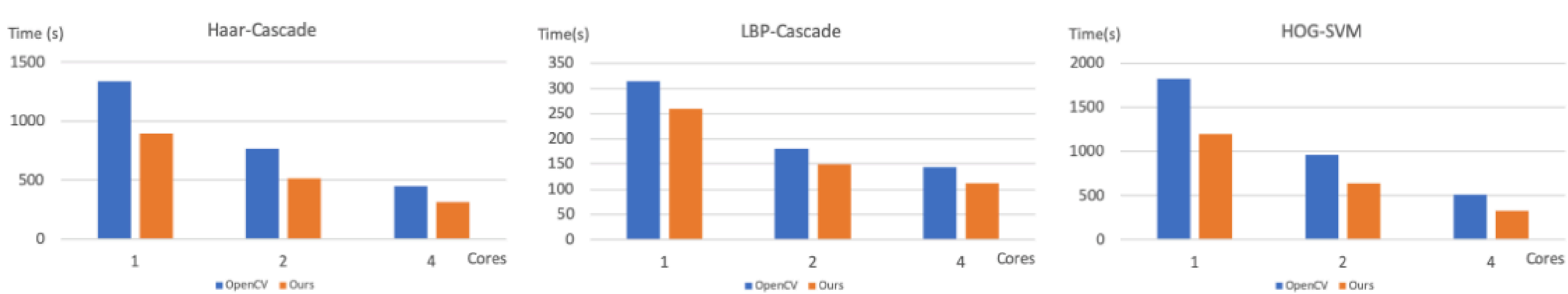


III: Experiment



(a) Haar-Cascade (b) LBP-Cascade (c) HOG-SVM

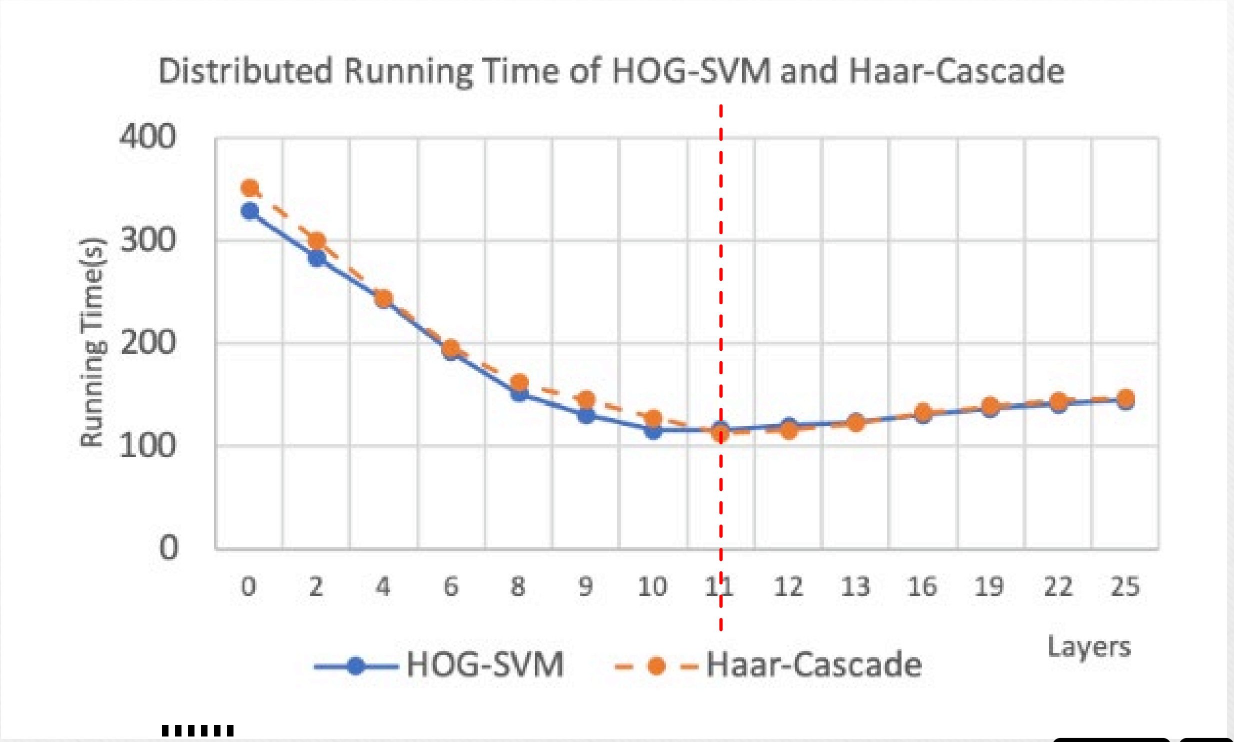
Comparing Scalability of Our Framework with OpenCV on AWS Time(s)/Cores



(a) Haar-Cascade (b) LBP-Cascade (c) HOG-SVM

Comparing Scalability of Our Framework with OpenCV on Raspberry-PiTime(s)/Cores

III: Experiment



6 Rpis

Desktop = 15 x Rpi

Running Time for Distributed Application with Different Layers of Images Processed on Desktop

III: Benchmark

Observations:

- Overall speedups on 16 cores of 9.46x and 7.57x for Haar and LBP-Cascade
- For HOG-SVM ours show better scalability
- On Rpi using one core, ours is faster by 17% - 35% over OpenCV.
- The sweet spot in load balancing experiment matches with our prediction.

III: Summary

- Our results have shown that we can effectively parallelize and scale across cores on both edge and central device.
- Our framework overall out performs OpenCV in all three applications.
- We are also able to reduce latency by dividing the work betweenedge and central devices.

Conclusions

- **Can achieve performance and programmability**
- **Pattern-based APIs enable new optimizations**
- **Many more applications/scenarios can be considered**