GoSpark: An In-Memory Distributed Computation Platform in Go

Kuan-Ting Yu
CSAIL
MIT

Jiasi Shen
CSAIL
MIT

Bolei Zhou
CSAIL
MIT

1 Introduction

Computation tasks in Computer Vision research are data-intensive. Such tasks usually involve performing repeated single operations, such as image resizing and feature extraction, on thousands of images. They also require training iterative algorithms, such as K-means clustering or logistic regressions, on thousands of image feature vectors. To support the computation of such tasks, the CSAIL Computer Vision Group has been using 36 high-performance in-house computation servers[1]. However, currently there is no efficient way to distribute the processing of large data sets across these servers. Each user can only connect to one single server using Secure Shell (SSH) and perform computation task (under Matlab, Python, or C++) with data and intermediate results stored in the NFS space maintained by CSAIL TIG. The use of centralized NFS space as data storage has caused I/O to be the bottleneck for large scale data processing tasks. Thus, a distributed computation platform with distributed file system would highly improve the performance of computations in computer vision research.

With Hadoop Distributed File System (HDFS) installed and configured on selected vision servers, we build a distributed computation platform, **GoSpark**[2], to process large data sets across these servers. GoSpark is a fast in-memory cluster computing framework Spark [3] under the implementation of Go programming language. Our version of Resilient Distributed Dataset (RDD) in GoSpark supports a variety of transformations and actions. GoSpark is fault-tolerant to the worker failures and network disconnections.

In the following sections, we will first explain the design and implementation of GoSpark, and then demonstrate its functionality and fault tolerance with two image processing examples.

2 Design and Implementation

2.1 RDD Data Abstraction

RDD is the data abstraction in GoSpark. It defines the interfaces of transformations and actions, as listed in [3]. We implemented various kinds of `operationType`, including `Map`, `FlatMap`, `Reduce`, `ReduceByKey`, `Filter`, `Collect`.

Code 1: RDD data structure

```
type RDD struct {
  splits []*Split
  dependType    string    // Narrow or Wide
  splitType     string    // Hash partition or range partition
  operationType string    // Map, reduceBykey ...
  fnName        string    // function name for map, reduce ...
  prevRDD1 *RDD
  prevRDD2 *RDD
  ...
}
```

The RDD data structure maintains `prevRDD1` and `prevRDD2` to track the RDD(s) that this current RDD is derived from. `prevRDD2` is used for join or union operation. `fnName` keeps the name[3] of input function literal. `splits` is used to keep the splitIDs of the RDD, as each splitID is a unique 64-bit integer.

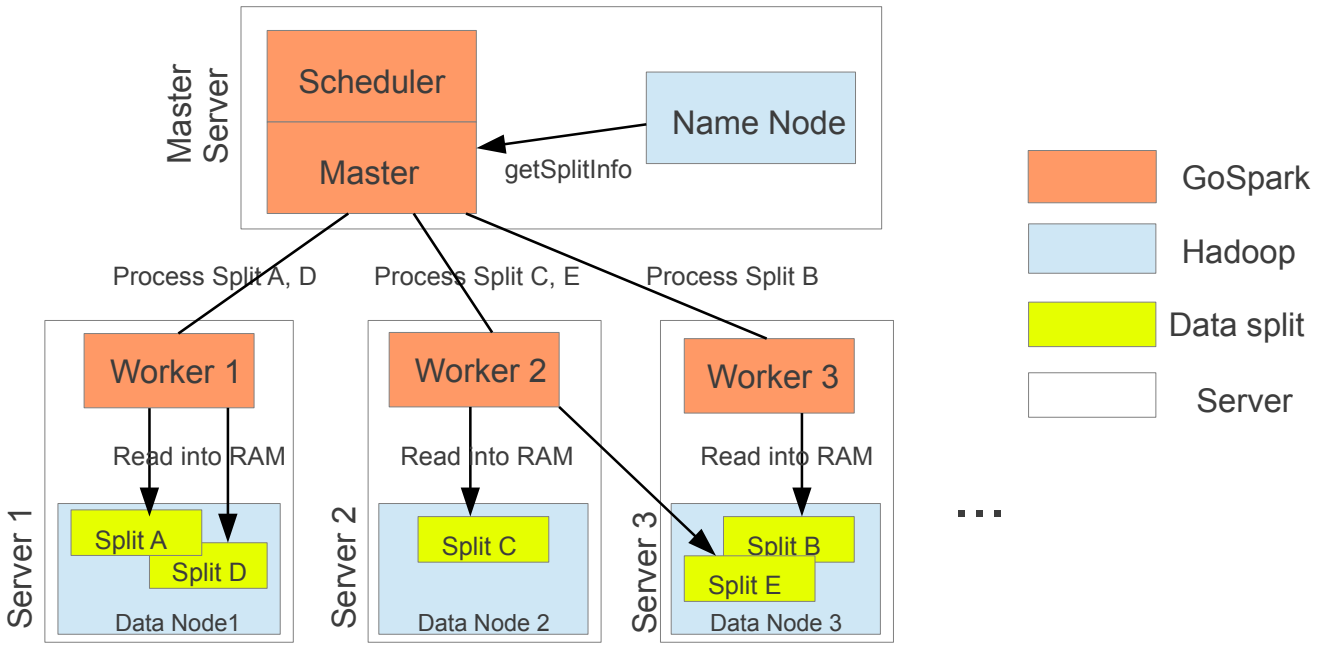


Figure 1: The system design of GoSpark (Scheduler, Master, and Workers) and HDFS (Name node, data nodes, and data splits). Due to the high-performance of the vision servers, each vision server is both a data node in Hadoop HDFS and a worker in GoSpark at the same time.

2.2 Roles

Figure 1 shows the overall design of GoSpark. There are three main roles:

Scheduler builds the lineage of RDDs and tracks the lineage across a wide range of transformations.

Master maintains the status of workers and communicates with them through TCP connections.

Workers read input data from either HDFS or other workers, perform specified computation, and store results in memory. Their jobs are decided by the scheduler module and distributed by the master.

2.2.1 Scheduler

The scheduler is activated when an RDD action is triggered. When scheduling, it examines the full lineage of RDDs related to the target, and identifies the execution order.

Code 2: Scheduler API

```
type Scheduler struct {
  master *Master
}

func makeDagFromRdd(rdd *RDD) *Dag{...}
func (d *Scheduler) runThisSplit(rdd *RDD, SpInd int) error {...
    switch rdd.operationType {
        case Map: ...
            ok, _ := d.master.AssignJob([]string{addressWorkerInMaster}, true, &args, &reply)
        case ReduceByKey:...
}
func (d *Scheduler) computeRDD(rdd* RDD, operationType string, fn string) []interface{} {
 switch (operationType) {
  case "Collect":...
  case "Count":...
  case "Reduce":...
}
```

[1]Server viewboard is at `http://foxtrot.csail.mit.edu/cgi/machine_stats.cgi`.
[2]Our code is available at `https://github.com/metalbubble/824project`.
[3]Please refer to Section 2.2.3 for more detail.

The scheduling process has the following steps: 1) Extracting stages, which are the RDDs before a wide dependency, and the target RDD; 2) Building a directed acyclic graph among the stages; 3) Topologically sorting the DAG to obtain an execution order of stages, and execute the stages one-by-one; 4) Executing a stage and computing each split in the stage in parallel. This includes tracing back the previous RDDs recursively and executing the corresponding split. 5) When the previous splits that another split depend on are done, `runThisSplit()` is called to send job specification, including opType, inputSplits, addresses, and outputSplits through `master.AssignJob()` to workers.

Data Locality The scheduler will consider the locality of data splits when distributing jobs to workers. Specifically, when the scheduler commands the master to assign a job to workers, the scheduler will provide a list of preferred workers based on data locality. The master will choose a worker based on this list. For example, at the beginning of a lineage, the scheduler assigns the workers that are close to the HDFS splits with `ReadHDFSSplit` jobs. These workers are responsible for reading data from HDFS. Other workers that need these data will send out `GetSplit` requests to them to fetch the splits.

2.2.2 Master

The master module builds the bridge between the scheduler and the workers. Specifically, when the scheduler needs to assign a job to a worker, it calls `master.AssignJob()`, which selects a worker and assigns it the job through the `DoJob` RPC. In addition, the master also provides a `Register` RPC for workers to register and to update their status.

Code 3: Master API

```
type Master struct {
  MasterAddress string
  MasterPort string
  workers map[string]WorkerInfo
}
func MakeMaster(ip string, port string) *Master {...}
func (mr *Master) StartRegistrationServer() {...}
func (mr *Master) Register(args *RegisterArgs, res *RegisterReply) error {...}
func (mr *Master) WorkersAvailable() map[string]WorkerInfo {...}
func (mr *Master) AssignJob(workersPreferred []string, force bool, args *DoJobArgs, reply *DoJobReply) (bool, string) {...
    ok := call(w, "Worker.DoJob", args, reply)...
}
func (mr *Master) webHandler(ws *websocket.Conn) {...}
```

Load Balancing When assigning a job, the master selects an available worker based on the following information: a) the list of data locality preference that the scheduler provides, b) the number of CPUs on each worker machine, and c) the number of GoSpark jobs currently running on each worker. To demonstrate the data locality and load balancing features of GoSpark, we implemented a monitor page[4] that displays the real-time CPU and memory usage of all the workers. Its implementation establishes `websocket` connections between Javascript and Go. Figure 2 shows two snapshots of the status of the workers.

2.2.3 Workers

Workers directly conduct data processing tasks as specified in `DoJob` RPC arguments. Different jobs are processed in parallel.

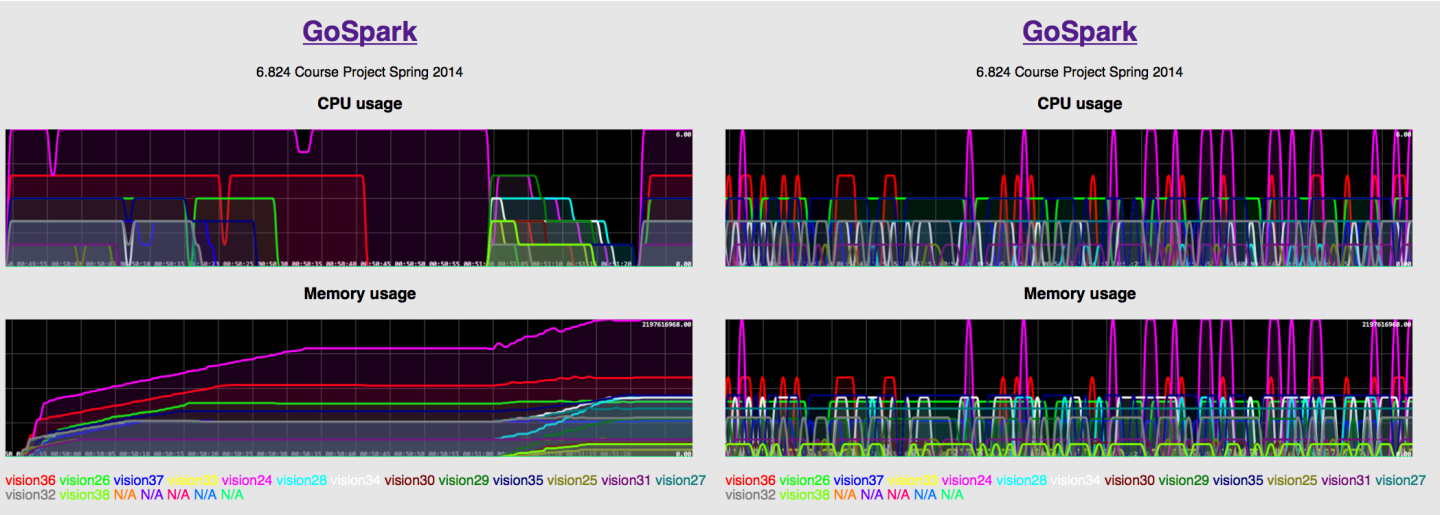
Code 4: DoJob RPC arguments

```
type DoJobArgs struct {
  Operation JobType
  InputIDs []Split
  OutputIDs []Split
  Function string
  ...
}
```

Code 5: Worker API

```
type Worker struct {
  unreliable bool // for testing
  mem map[string]([]interface{})
  ...
}
func (wk *Worker) DoJob(args *DoJobArgs, res *DoJobReply) error {...}
func (wk *Worker) Shutdown(args *ShutdownArgs, res *ShutdownReply) error {...}
func MakeWorker(MasterAddress string, MasterPort string, me string, port string, unrel bool) *Worker {...}
func (wk *Worker) kill() {...} // for testing
```

[4]The worker status monitor is accessible at `http://web.mit.edu/jiasi/www/ws.html`.



(a) At the beginning of a lineage

(b) At the middle of a lineage

Figure 2: Two snapshots of worker load distribution

To tolerate network failures, the workers not only register to the master during initialization, but also frequently do so whenever they are alive.

Memory management To achieve fast in-memory computation, each worker stores all computation results in its local memory `mem` until being told to delete them by `DelSplit` jobs. For workers to cooperate, each intermediate result is uniquely identified by a `SplitID`. The `SplitID`s are allocated by the scheduler in `DoJob` arguments. For example, when worker A needs input data from worker B, the scheduler tells it the address of worker B and the `SplitID` of the data. With these information, worker A can send a `GetSplit` request to worker B. Then, worker B looks up the data in its local memory and replies to worker A with its contents.

Function Literals Since Go cannot serialize function literals, it is impossible to pass their definitions from the master to workers through RPCs. Thus, we instead compile user functions together with our GoSpark code on worker machines. The master instructs workers of which function to run by passing function names as `strings` in `DoJob` RPC arguments. Workers then use the run-time reflection in Go to call the desired functions by name.

Code 6: Call a function by name with Go reflection

```
1  fn := reflect.ValueOf(&UserFunc{}).MethodByName(args.Function) // look up by name
2  a1 := reflect.ValueOf(KeyValue{Value:line}) // wrap arguments in data structures
3  a2 := reflect.ValueOf(KeyValue{Value:args.Data})
4  r: = fn.Call([]reflect.Value{a1, a2}) // call function by name
5  s: = r[0].Interface() // result
```

There are still some technical restrictions, however. The `reflect` package requires a data structure to be defined to look up methods by name. Thus, we define `UserFunc` and restrict users to implement function literals under this type. Another restriction is that `reflect` does not support calling functions whose arguments have type `interface{}`. Thus, we also have to wrap user function arguments in a data structure (we chose `KeyValue`).

Unreliability In order to test the fault tolerance of GoSpark, we need to sometimes force the network connections to fail. To this end, we use `conn.Close()` to discard RPC requests, and use `syscall.Shutdown()` to force discard of replies after processing requests.

2.3 Fault Tolerance

GoSpark has three fault tolerance mechanisms, from three different levels.

RDD fault tolerance At the execution stage of RDDs, the worker will first check the dependency of RDDs based on the assigned job. If there are some splits missing, that worker will report to the scheduler. Then scheduler will check the dependency of that missing split and re-launch the task of processing the split on some worker. Until all the dependency of RDDs is complete, the failed job will be assigned again.

Worker fault tolerance Sometimes workers will be disconnected to the master due to network disconnection or the crash of the worker server. If an RPC call from the master to a worker fails, the master will retry a few times until success. If a worker is still unreachable, the master considers the worker as crashed and removes it from the available workers list. Then it selects another worker from the list and assigns it the specified job. If the master cannot reach a worker while the worker is in fact alive, the worker will keep calling the `Register` RPC on master. When its register requests eventually reach the master, the master will mark the worker as available again.

HDFS fault tolerance The HDFS is configured to have n replicas among the workers. If there is a machine failed after 10-minute time-out, the HDFS master will automatically recover the lost replicas on other servers. Note that we assume the scheduler and master will not fail.

3 Experiments

3.1 HDFS setup and access

To build and test our distributed computation platform, we installed and configured HDFS (Hadoop 2.4) on 15 selected vision servers. Since vision servers use the shared NFS space as the primary data storage, to avoid the bottleneck of data I/O the default HDFS path is set to the local drive of each vision server in `/scratch/tmp`. The replicate number is set as 4. Vision24 server is selected as the name node of Hadoop [5] and the master/scheduler of GoSpark, while all the selected vision servers are served as data nodes and workers of GoSpark.

HDFS partitions a large file into 128 MB data splits among data nodes. One issue here is the locality of data access between workers and HDFS, *i.e.*, each worker should access the data splits stored in its own drive in priority. HDFS API is provided for workers to access the splits of the large file separately. However the official HDFS APIs are mainly in JAVA, we implement a HDFS split reader in JAVA. We wrote a wrapper in go to call this Java reader. For example, for a large file stored at `file := "hdfs://..."`, the scheduler will first get the information of data splits of HDFS by calling `GetSplitInfo(file)`, then the master in GoSpark will know where the splits are stored and assign the HDFS link of these data splits to each worker based on locality. Finally the worker will read the data split in HDFS using `ReadHDFSSplit(file, splitID)` and conduct data processing operations independently. Figure 1 illustrates the data access process in GoSpark.

3.2 Image clustering using K-Means

We demonstrate image clustering using K-means on the GoSpark platform. We use the SUN Database [2], a scene categorization benchmark, as the data for K-Means clustering. There are totally 108,754 images in the database, which come from 397 scene categories, such as kitchen, coast, and mountain. The feature of each image in the database is extracted as a 4096 dimensional vector by the pre-trained neural network [1]. All the feature vectors are then stored as a single 3.32 GB CSV file at HDFS.

K-Means is a commonly used clustering method. Given an initial set of k centers by random, the algorithm iteratively proceeds two steps: 1) Assignment step: to assign each data vector to its closest center. 2) Update step: to recalculate the k centers by taking the mean of vectors assigned to each label.

The key lines of the K-Means clustering in GoSpark are listed as following

```
pointsText := c.TextFile("hdfs://vision24.csail.mit.edu:54310/user/featureSUN397_combine.csv")
points := pointsText.Map("MapLineToFloatVectorCSVWithCat").Cache()
var mappedPoints *RDD
for i := 0; i < IterNum; i++ {
  mappedPoints = points.MapWithData("MapToClosestCenter", centers) //AssignmentStep
  sumCenters := mappedPoints.ReduceByKey("AddCenterWCounter")
    newCenters := sumCenters.Map("AvgCenter")             //Update Step
    newCentersCollected := newCenters.Collect()
    for j:=0; j<len(newCentersCollected); j++ {
        centers[j] = *(newCentersCollected[j].(KeyValue).Value.(*Vector))
    }
}
```

Since Go language cannot pass function literals between master and worker via RPC, we use run-time reflection in Go to call the function in test, such as `"MapToClosestCenter"` listed below. The other defined function literals could be found in `kmeans_test.go`.

```
// function literal of "MapToClosestCenter"
func (f *UserFunc) MapToClosestCenter(line interface{}, userData interface{}) interface{} {
  p := line.(KeyValue).Value.(Vector)
  centers := userData.(KeyValue).Value.([]Vector)

  minDist := p.EulaDistance(centers[0])
  minIndex := 0
```

[5] The dashboard of our Hadoop system is at `http://vision24.csail.mit.edu:50070`.

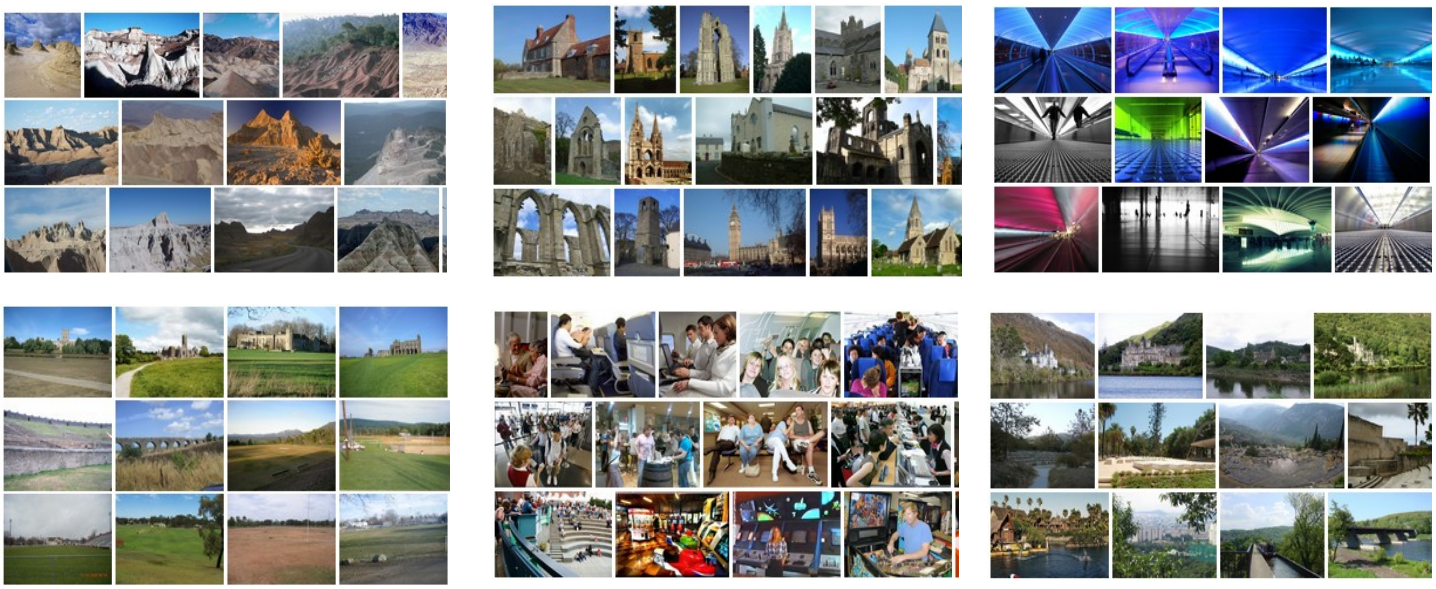


Figure 3: Representative images in 6 clusters from K-Means clustering.

```
  for i := 1; i < len(centers); i++ {
    dist := p.EulaDistance(centers[i])
    if dist < minDist {
      minDist = dist
      minIndex = i
    }
  }
  return KeyValue{
                Key: minIndex,
                Value: CenterCounter{p, 1},
        }
}
```

The representative images from 6 clusters of K-Means (here we set k=50) [6] are shown in Figure 3. We can see that GoSpark K-Means clusters images with the similar appearance together.

To evaluate the performance of GoSpark, we plot the iteration time of K-Means clustering with different number of workers along with unreliable worker scenario in Figure 4. We can see that as the number of workers increases, the iteration time drops. As the data shuffle also increases with the number of workers, the relation is not linear. Meanwhile, because of retrying RPC, the iteration time in unreliable worker case is higher than the normal reliable worker case.

3.3 Scene classification using logistic regression

Since images in the SUN database are from 397 scene categories, we can train a classifier to recognize the scene categories based on the image features. Here we train the logistic regression classifiers on our GoSpark platform. We use one-vs-all training scenario, *i.e.*, for each category we take the images in this category as positive samples and take images in all the other categories as negative samples. Thus, we could train 397 logistic regression classifiers. Given one image in testing set, we classify the images to the category with the largest output of logistic regression classifier.

The key lines of training logistic regression in GoSpark are listed as

```
pointsText := c.TextFile("hdfs://vision24.csail.mit.edu:54310/user/featureSUN397_binary.csv");
points := pointsText.Map("MapLineToFloatVectorCatCSV").Cache();
for i:=0; i<IterNum; i++ {
    mappedPoints := points.MapWithData("MapToVectorGradient", w);
    gradInterface := mappedPoints.Reduce("ReduceToOneGradient")
    w = w.Minus((gradInterface.(Vector)))
}
```

[6] Cluster result is at `http://wednesday.csail.mit.edu/cityimage/SUN_source_code_v2/data/clusterResult.html`

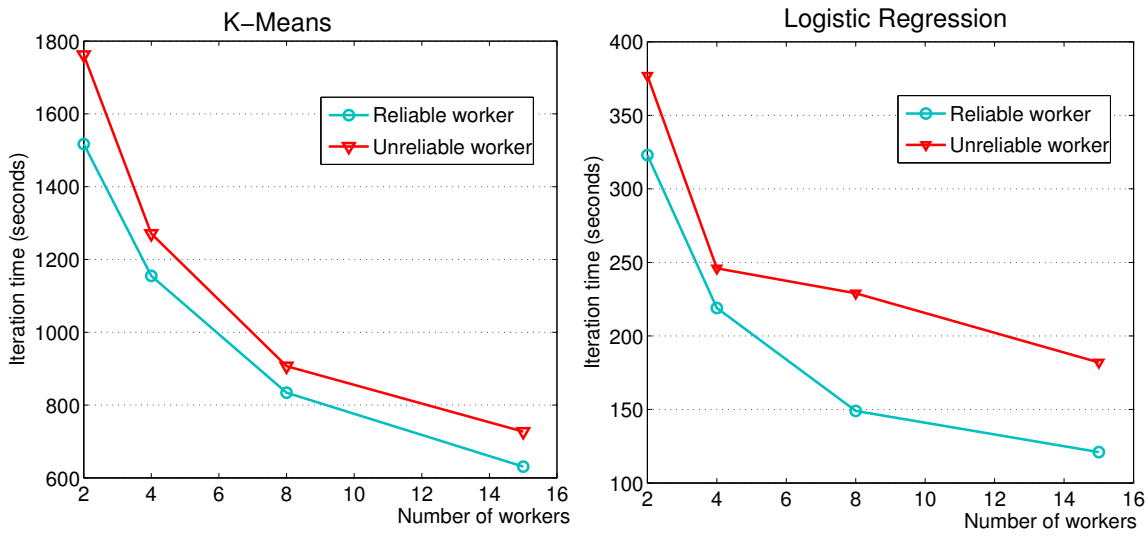


Figure 4: The running time (seconds) with various numbers of workers for K-Means and logistic regression

We define function literals "`MapToVectorGradient`" as follows. Note that due to the restriction that we cannot pass in interface directly to a function obtained from reflection, we need to encapsulate the arguments into a type, *e.g.* we put the xy (x: feature, y: label) into the "`Value`" of "`KeyValue`" type.

```
// function literal of "MapToVectorGradient"
func (f *UserFunc) MapToVectorGradient(xy interface{}, wInterface interface{}) interface{} {
  y := xy.(KeyValue).Value.(KeyValue).Key.(float64)
  x := xy.(KeyValue).Value.(KeyValue).Value.(Vector)
  w := wInterface.(KeyValue).Value.(*Vector)
  grad := x.Multiply(1/(1+math.Exp(-y*(w.Dot(x))))-1).Multiply(y)
  return grad
}
```

The whole logistic regression test can be found in `LR_test.go`.

3.4 Runtime anaysis

The main purpose of adopting Spark to Computer Vision tasks is its capability to scale with huge data size and increasing number of workers. Figure 4 shows the total runtime of K-Means and logistic regression (LR) on various number of workers, and on both scenerios of reliable and unreliable workers. As worker number increases, the runtime generally decreases. However, there seems to be some non-trivial overhead that prevents our system to be close to linear speedup. The source may be non-uniform processing speed of workers, *i.e.* some workers were processing jobs by other users, or network bottleneck while exchanging data in the shuffle stage of K-Means. This issue is worth investigating in the future for better scalability.

To further examine the runtime, we plot the average time for computing each RDD operations for the two applications in Figure 5, and the runtime proportion of certain operations in the whole tests in Figure 6. The tests were run on 15 machines. For both applications, we found that the jobs of reading splits from HDFS and parsing strings into float vectors are quite expensive in Figure 5. With the advantage of the design of Spark, we can cache these preprocessing in memory, so that they are only one-time expense. Otherwise, in traditional MapReduce, these expenses are added to every iteration, which greatly increases the total runtime. In LR case, the real data process jobs in iterations like `MapToVectorGradient`, and `ReduceToOneGradient` are relatively insignificant, because they are done on each worker independently and in memory. For K-Means, the `MapToClosestCenter` mapping, and `AddCenterWithCounter` reduction are two expensive jobs in every iteration. The former compares 1 vector against k vectors of centers, and the latter obtains data from other workers for reduction like the shuffle stage in MapReduce. On the right-hand side of the chart are `HashPart`, the preparation of shuffle stage on individual machines, and `AvgCenter` an one-to-one vector mapping computation. These are relatively cheap operations.

Figure 6 shows how the total runtime are spend. In LR case, the preprocessing occupies a major percentage of the total runtime. The overhead would be amortized if we run LR with more number of iterations, which is currently 10 only. On the other hand for K-Means, the preprocessing time is insignificant. The main protion of time was spent on `MapToClosestCenter`, and `AddCenterWithCounter` in every iteration. This is a more ideal situation in terms of efficiency where time are spent on computation than preprocessing.

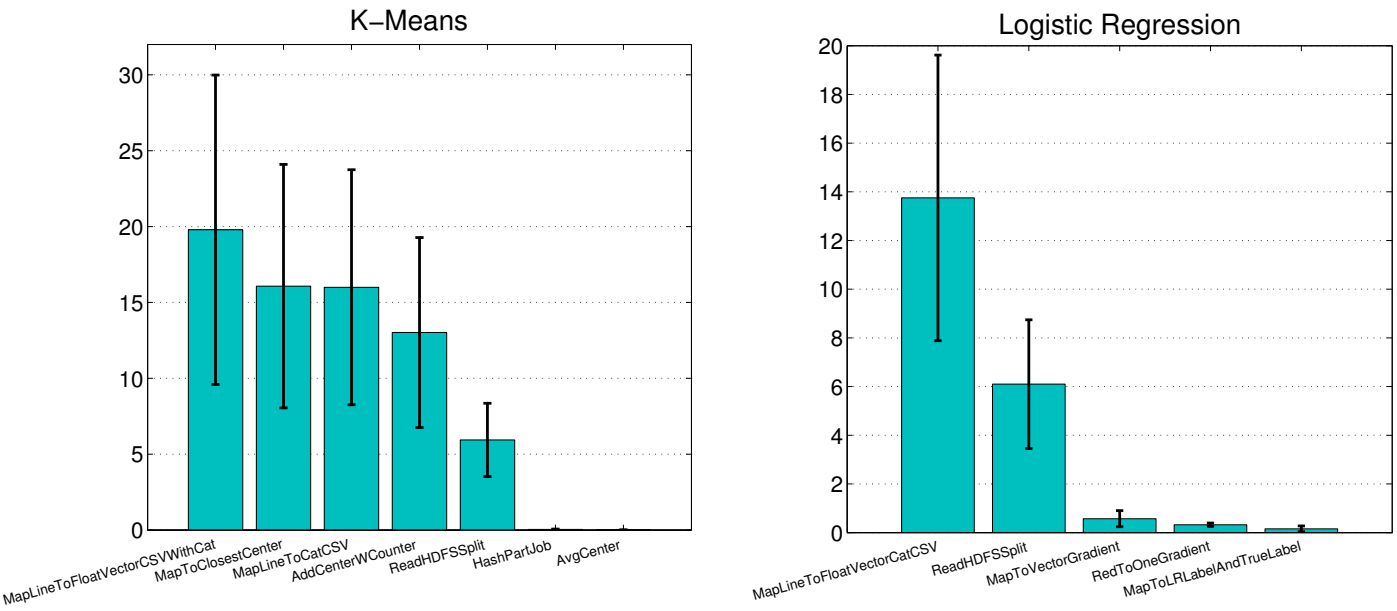


Figure 5: Average runtime (seconds) of computing each type of rdd operation for one split

4 Conclusion

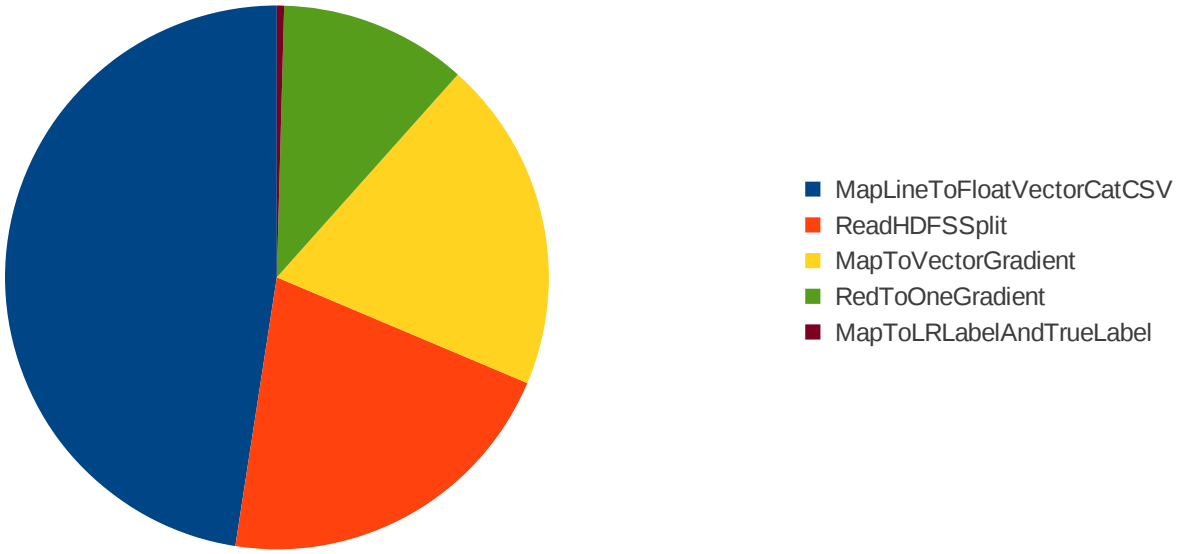
In this course project, we build a practical distributed in-memory computation platform GoSpark based on the RDD framework [3] using Go language. We implemented some of the major transformations and actions for RDD, and a scheduler to order the execution of each split based on dependency rules. We integrated HDFS to store large data and for workers to access data splits locally. Fault tolerance to unreliable workers are tested. We evaluate the final system on 15 vision servers through the examples of image clustering and image classification, and show that with the design of caching in Spark workers need to load the data from disk once, which effectively cut down this expensive overhead in traditional MapReduce.

References

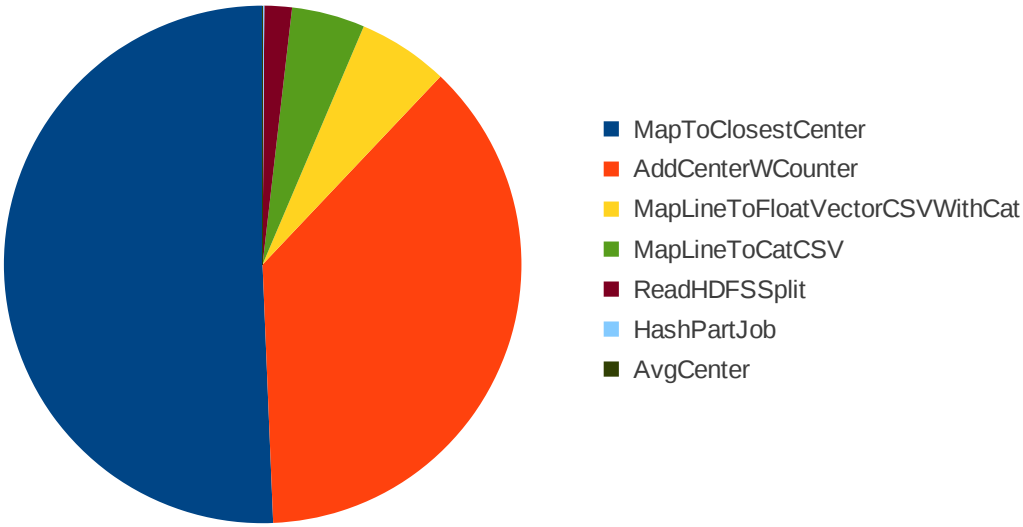
[1] J. Donahue, Y. Jia, O. Vinyals, J. Hoffman, N. Zhang, E. Tzeng, and T. Darrell. Decaf: A deep convolutional activation feature for generic visual recognition. *arXiv preprint arXiv:1310.1531*, 2013.

[2] J. Xiao, J. Hays, K. A. Ehinger, A. Oliva, and A. Torralba. Sun database: Large-scale scene recognition from abbey to zoo. In *Computer vision and pattern recognition (CVPR), 2010 IEEE conference on*, 2010.

[3] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation*.



(a) Logistic Regression



(b) Kmeans

Figure 6: Proportion of time spent on each type of rdd operation