Design and Evaluation of a Novel DataFlow based BigData Solution

Yao Wu
ET International Inc.
Department of Electrical and Computer Engineering,
University of Delaware,
Newark, DE
ywu@etinternational.com

Long Zheng
Department of Computer Science and Engineering,
Shanghai Jiao Tong University,
Shanghai, China
longzheng@sjtu.edu.cn

Brian Heilig*
ET International Inc.
Newark, DE
bheilig@etinternational.com

Guang R. Gao
Department of Electrical and Computer Engineering,
University of Delaware,
Newark, DE
ggao.capsl@gmail.com

ABSTRACT

As the attention given to big data grows, cluster computing systems for distributed processing of large data sets become the mainstream and critical requirement in high performance distributed system research. One of the most successful system is Hadoop which uses MapReduce as programming/execution model and takes disks as intermedia to process huge volume of data. However, currently, it is a consensus that Hadoop is not the final solution to BigData due to MapReduce programming model, disk based computing, synchronous execution model and the constraint that only supports batch processing, and so on. A new solution, especially, a fundamental evolution is needed to bring BigData solution into a new era.

In this paper, we introduce a new cluster computing system called HAMR which supports both batch and streaming processing. To achieve better performance, HAMR integrates HPC approaches, i.e. DataFlow fundamental into a big data solution. With more speicifications, HAMR is fully designed based on In-Memory computing to reduce the unnecessary disk access overhead; task scheduling and memory management are in fine-grain manner to explore more parallelism; asynchronous execution improves efficiency of computation resource usage, and furtherly makes workload balance across the whole cluster better. The experimental results show that HAMR can outperform Hadoop by up to 10x in the same cluster environment.

*This author is the corresponding author.

PMAM '15, February 7-8, 2015, San Francisco Bay Area, USA

http://dx.doi.org/10.1145/2712386.2712397.

Categories and Subject Descriptors

D.4.7 [**Operating Systems**]: Organization and Design—*Distributed systems*

General Terms

Algorithms, Design, Performance

Keywords

Dataflow, In-memory processing, Fine-Grain, Bigdata

1. INTRODUCTION

The MapReduce model has become widely popular in cluster computing systems for processing a large volume of data sets. As the most famous implementation of MapReduce, Hadoop [16] achieves massively parallel computing on distributed large data sets across vast clusters of commodity computers, while providing simple workflow expression, fault tolerance, and scalability. Nowadays, Hadoop has evolved into an ecosystem including Hive, HBase, Pig, *etc.* and has become a standard framework for the big data industry. The Hadoop MapReduce is a prominent programming model for batch processing, while it cannot fully meet the demand of streaming and real-time requests. Due to the on-disk feature, Hadoop performs poorly on iterative algorithms which are very common in machine learning and data mining.

In terms of these critical limitations, a couple of solutions have been proposed to handle streaming processing *e.g.* Storm and Spark [17]. However, none of the solutions provide a general framework supporting both batch and streaming processing. Moreover, most of the existing cluster computing systems still rely on the coarse-grain synchronization *i.e.* barrier to coordinate between phases. Therefore, the parallelism is limited and the computation resource tends to be wasted.

In this paper, we present a new cluster computing framework called HAMR which is a dataflow-based real-time in-memory cluster computing engine. Comparing with existing

big data solutions, HAMR is designed as a big data processing run-time and makes a breakthrough in task scheduling, memory management, execution model, and programming model. Based on dataflow technology, HAMR proposes a radically different model from MapReduce for big data processing and naturally supports streaming and real-time computing. Therefore, HAMR fully supports Lambda big data architecture [1] by using the same programming and processing model in only one computing engine. In HAMR, data drives and determines computation, which makes the system more intelligent, efficient, and adaptive when processing data. HAMR can be seamlessly integrated into Hadoop ecosystem. By configuration, HAMR can use YARN as the resource negotiator to allocate and monitor compute containers for flowlet tasks on machines in the cluster. Furthermore, HAMR is implemented in Java to provide user-friendly interfaces.

The main component in HAMR is the flowlet which represents each MapReduce phase. The flowlet API is also similar to Hadoop MapReduce one for easy use. However, the internal mechanism is considerably different. The flowlet model is developed from codelet execution model which is rooted in dataflow technology. In each flowlet, there are a number of flowlet tasks which execute a portion of work. Flowlet tasks are scheduled asynchronously in a fine-grain manner. Once the data for the flowlet task is available, it gets ready and will be executed when the computation resources are free. The flowlet execution model leads to better computation resource usage and workload balance. Moreover, the data movement flows in memory among flowlets. It saves unnecessary disk IO overhead. If the data is too large to fit into memory, it will be partially spilled to disk. We evaluate HAMR through eight big data benchmarks with different properties. The experimental results show that HAMR can outperform Hadoop by up to 10x for streaming and real-time applications.

The rest of the paper is organized as follows. Section 2 introduces the framework of HAMR and its dataflow-based run-time design. Section 3 analyze the features of HAMR and the benefits by apply them in the benchmarks. Section 4 describes the implementations of eight big data benchmarks in the flowlet model. Section 5 reports the experimental setup and results. Section 6 discusses the related work. Section 7 summarizes the paper.

2. FRAMEWORK DESIGN

In this section, we introduce the components in the framework of HAMR and focus on HAMR dataflow-based run-time design.

As a Big Data solution, HAMR is designed to process vast amounts of data in parallel on large clusters of commodity hardware in a reliable, fault-tolerant manner. Fig. 1 shows the overview of the system design.

The user interface is provided based on the execution model of the system. The MapReduce phase style called flowlet in HAMR is preserved in the model. According to the parallelism and data dependencies, the programs are divided and capsulated into a flowlet. Different from the existing MapReduce model, multiple flowlets in a single HAMR job are organized as a Directed Acyclic Graph (DAG) to represent a complex workflow. There are four types of flowlets that serve most purposes in the HAMR DAG workflow:

- Loader

  In the starting phase in the workflow, the loader flowlet tasks work to pull directly from multiple data sources simultaneously. The data sources include but are not limited to HDFS, HBase, local disks, distributed file system, relational database, NoSQL database, message broker, and other structured data sources.

- Map

  Functionally, the map flowlet is similar to the map phase in MapReduce. It consumes the key-value pairs from upstream flowlets and produces new key-values pairs to downstream flowlets. As shown in Fig. 1, a map flowlet can connect to any types of flowlets instead of only reduce in MapReduce.

- Reduce

  The reduce flowlet is also similar to the reduce in MapReduce. Unlike map, it must collects all key-value pairs which are grouped by key after shuffling and then process group by group.

- Partial Reduce

  Different from reduce, partial reduce processes the available data immediately instead of waiting for the whole data collection. It is extremely suitable for a computation having both commutative and associative properties. In such case, the computation can start early to better overlap network latency and the memory usage is compressed.

In-memory computation is supported by deploying buffers between flowlets. By keeping intermediate data in memory, disk spillover is reduced. In most cases the memory buffer can be freed timely and reused after data processing, because computation is usually not very intensive in big data applications and data flows out quickly. However, the data may exceed the limit of the buffer for the reduce flowlet. Similar to the reducer in MapReduce, the reduce flowlet also internally forms a barrier waiting until the upstreaming flowlets complete. Therefore, if the data flowing into the reduce flowlet is too large to fit into memory, it will be spilled to local disks.

In addition, the system provides a very flexible input/output way in that all flowlets can communicate with data sources dynamically. Therefore, the data can be pushed out as soon as possible. It eliminates the unnecessary data movement and memory occupation in the long workflow. On the other hand, flowlets can pull the data from the source when needed during the workflow.

HAMR exhibits similar external functions like MapReduce, but the run-time mechanism is completely different. The execution is driven by dataflow technology which is the fundamental breakthrough in the engine. The dataflow-based run-time environment is established by scheduling the current tasks of all flowlets in a data driven manner. The scheduler schedules tasks to be executed asynchronously based on required events which can be the availability of shared resources including data and computation elements *e.g.* CPU cores and memory. Such a fine-grain execution leads to explore maximum parallelism in a program. In the workflow of the system, the downstream phases can start processing

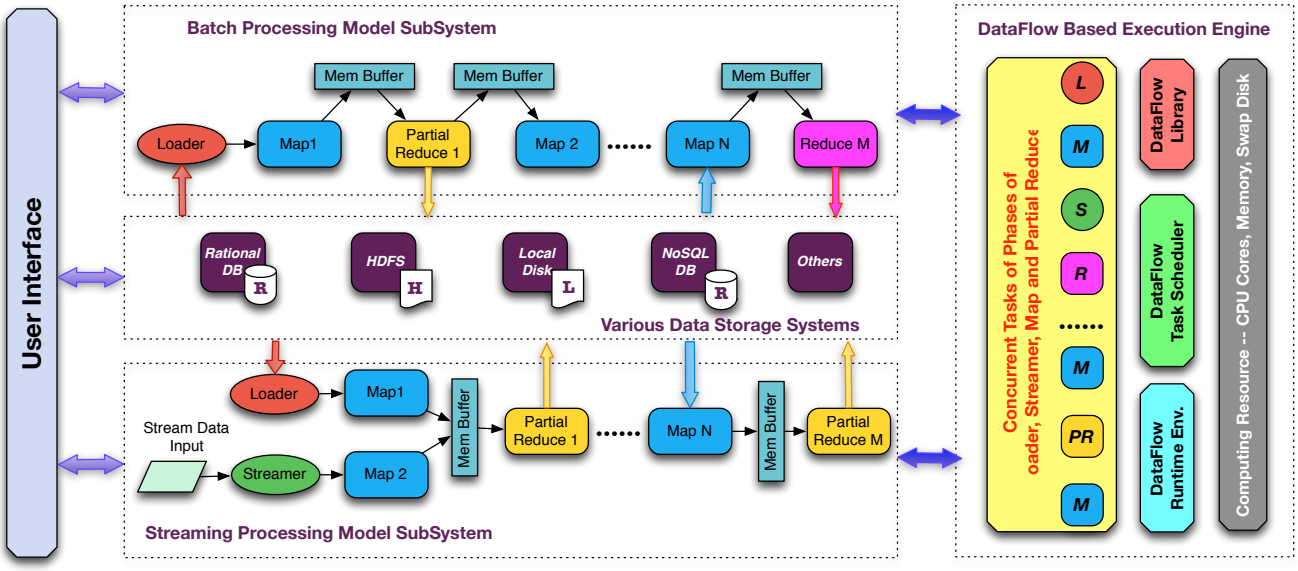


Figure 1: HAMR System Design. Based on dataflow execution engine, HAMR supports both batch and streaming processing on different data sources.

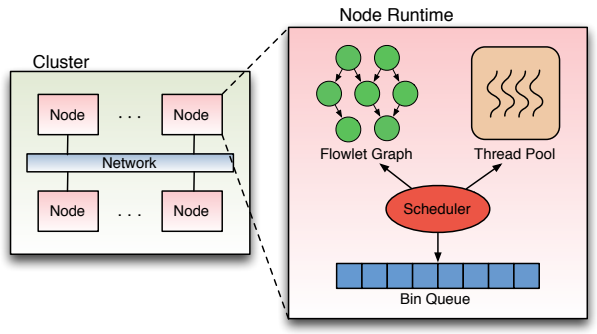


Figure 2: HAMR dataflow-based run-time system design. The run-time on each node works collaboratively to schedule the flowlet tasks based on the availability of data and computation resources.

the data from upstream phases before upstream phases complete. As a result, HAMR achieves high resource utilization and balanced workload. Fig. 2 shows the dataflow-based run-time design in the cluster environment.

In a HAMR cluster, each node maintains a distributed dataflow-based run-time called flowlet run-time. Different from other graph engines *e.g.* Dryad [10], the run-time on each node includes the whole flowlet graph instead of subgraph. It brings in more balanced workload on each node. The flowlet run-time schedules each flowlet task, which is the finest granularity of parallelism in the run-time, to the computation unit by the scheduling rules. Inside a flowlet task, the instructions are executed sequentially. A flowlet task run asynchronously without blocking until it completes. Flowlet execution is event-driven which means only when the required data and resources are available, the flowlets receive all synchronized signal and can start execution.

Each flowlet can have one of the following three status:

- Dormant: Not yet receive all required data.
- Ready: Receive all the data required to execute a flowlet task.
- Complete: No more data from upstream flowlets.

By the run-time coordination, a flowlet changes among the three status according to availability of data and computation resources. Initially, only loader flowlet is ready and others are dormant.

As shown in Fig. 2, HAMR run-time handles a queue which stores received key-value bins in each node. Each bin represents the minimum data required to enable a flowlet. The scheduler fetches a bin from the queue and finds its corresponding flowlet. If the flowlet is map or partial reduce, the status of the flowlet becomes ready. In the case of reduce flowlet, it needs all data instead of only one bin data produced by upstream flowlets *i.e.* must wait until all its upstream flowlets complete. The completion message is propagated from the beginning of the workflow *i.e.* loader to downstream flowlets by each node collaboratively. When the completion message arrives at the reduce flowlet, its dependency is satisfied and it is ready to execute. The run-time fires the ready flowlet once there is a free thread in the thread pool. The output of a flowlet is buffered, partitioned, packed into bins. The bins are sent out by shuffling. Note that the partial reduce flowlet does not output until the completion of its upstream flowlets. In the cluster, each node works on a portion of the whole key space. If the key space can be evenly divided by partition function, the workload tends to be balanced.

According to the run-time design, the data propagates among the flowlets in a streaming manner. Therefore, an effective flow control mechanism is needed to coordinate the data steam velocity in the flowlet graph. In HAMR, when

the output bin buffer of a flowlet is full which means its downstream flowlets are not able to process the data in time, the flowlet stops the current execution immediately and will be scheduled in a later time. Furthermore, the number of concurrent loader tasks can be decreased to control the amount of input data.

3. KEY FEATURES

In this section, we analyze three features of HAMR and the benchmarks which take advantage of these features.

3.1 In-memory Computation

The reason why Hadoop is very popular and widely used in BigData is that Hadoop can handle hundreds or thousands of Gigabytes of data on the distributed cluster, which is barely handled by the traditional methodology. However in order to support such quantity of data, Hadoop adopted the on-disk solution, which contains three aspects: 1) all input data are firstly distributed stored in disk across the cluster. 2) Each block will be processed separately by map and the temperate out data need disks to store/transferring to reduce phase. 3) If the processing logic contains multiple jobs, data must be stored in disk to connect these jobs. In a word, Hadoop realize processing large volume of data at cost of using disks as intermediate storage/communication, which leads to dramatic performance degradation.

The disk is designed for data persistence not for the temporary store for computing. The role that disk play in Hadoop actually is played by memory in traditional computer computing model. The on-disk solution is a compromise one when Hadoop was proposed. The BigData solution still need in-memory technology to decrease or avoid the disk overhead, further results in a significant performance improvement.

In HAMR, data dynamically flow via memory in a fine-grain manner from disk as input and finally to disk as output. Traditional method to realize in-memory computation is to implement mapping between memory and disk *e.g.* Spark. All data is needed to load into memory before computation. It wastes memory usage and is not scalable for large data set. Besides, memory is precious resource in big data computing not only for data storage but also data structure created during computation. Instead of cores, YARN schedules the tasks based on available memory on nodes. On the other hand, HAMR only loads a part of data set that is required by current computation units into memory. Once the computation for the data is completed, the result flows away and the memory footprint is also vacuumed to load a part of remaining data. All flowlets associate memory with computation in a fine-gain manner. It utilizes memory in a extremely efficient way.

Therefore, HAMR provide a in-memory solution to overcome the existing on-disk one to eliminate the overhead caused by IO access to disk. Besides, the in-memory method is driven by data flow fundamental, which can utilize memory much more efficiently. Applications with multiple iterations can get benefit from it, for example PageRank and KCliques, which will be explained in Section 4.

3.2 Asynchronous Multi-Phase Support

In traditional control flow method, synchronization points in form of barriers are imposed between computation elements that have data dependency to achieve parallelism.

Algorithm 1 Flowlet-style K-Means Algorithm

1: Initialize parameters including input/output file path and initial centroids.
2: TextLoader (loader): Automatically split and read text files, producing ⟨ line offset, line of text ⟩ as the output key-value pair to ClusterGen.
3: ClusterGen (mapper): For each movie, do clustering based on its similarity with the centroids; output each cluster in separate files; pass the similarity info to NewCentroidGen.
4: NewCentroidGen (reducer): Get the new centroids based on similarity info; pass the line offset of the new centroid to corresponding node where the file including the new centroid lives.
5: NewCentroidInfoGet (mapper): Read the new centroid info from files according to the line offset; broadcast the new centroid info to all nodes.
6: CentroidUpdate (mapper): Update new centroid info locally in each node.

This coarse-grain parallelism brings in workload balance problem and computation resource under-utilization. Hadoop uses a simple MapReduce programming model that splits data processing into map and reduce phases in one job. Moreover, a barrier exists between map and reduce phase in a job. Reduce tasks can pull the data from map tasks based on shuffle after the first map task is done. It effectively hides the latency of network. However, the problem is that the computation work of reduce tasks will not start until all the data are ready for all the reduce tasks. As a result, the precious computation resources are wasted by waiting for the data that they don't care about.

HAMR solves the problem by using data to drive the computation. As a result, the unnecessary barriers can be removed and each computation element works asynchronously and adaptively to response to arriving data. Using in-memory support we mentioned in Section 3.1, asynchronous computation can be realized by starting as soon as the data that can trigger it is available in memory. This asynchronous processing style tends to achieve better resource utilization which will lead to better performance.

In Hadoop, each job only has two phases: map and reduce and the order is also fixed: first map, then reduce. Thus, many complex problems can not be implemented with a single MapReduce job, but can be implemented in Hadoop by chaining multiple MapReduce jobs together. It brings in not only the overhead of creating and starting new jobs in Hadoop but also extra disk IO. Besides, between jobs, there is also a barrier to ensure data availability. Sometimes, this barrier is not necessary, since following job may start as soon as a part of data arrives.

However, HAMR employs much more flexible way to organize the computation phases: a chain of flowlets can be setup in a job; one flowlet can to connect to any type of flowlet; there are can be multiple flowlets flowing to one flowlet and vice verse. With multi-phase support, a chain of jobs can be packed into a DAG style job. It can be used to implement some complex application that is not suitable for Hadoop like graph problem. The barriers between jobs are automatically removed since each phase in the big job works asynchronously. In-memory flow style within the big job helps to eliminate the unnecessary disk IO. Besides, multi-phase

implementation breaks MapReduce style and provides more opportunities to achieve some optimizations *e.g.* data reuse. If one data set requires two different of operations, HAMR only needs to load data once and connect the loader to two flowlets with different functions.

Therefore, HAMR provides an asynchronous multi-phase solution to overcome the barrier in a job or between jobs and eliminate the overhead caused by creating jobs and extra IO access to disk. Besides, the asynchronous multi-phase method is driven by data flow fundamental, which can utilize computation resource much more efficiently. Applications with multiple jobs can get benefit from it, for example PageRank and KCliques, which will be explained in Section 4.

3.3 Data Locality Awareness

Hadoop always tries to assign computation to the node which is closest to the data in HDFS in order to save the overhead of network transmission. However, this kind of locality exploration is limited and HAMR has more flexible input and output which can be aware of locality. Each computation node can only process and output if needed the data on its local disk and output can happen not only in reduce as Hadoop does but also in map. The benefit is that it is not necessary to deliver large data between two flowlets. Instead, in the case that only a part of data is needed in the downstream flowlet, small data *e.g.* index or identifier is passed. When large data is needed again, by adding new flowlet, go back to the node which the data resides in. It not only save the overhead of network transmission but also memory usage. K-Means and Classification are typical benchmarks to show the locality advantage. Classification is very similar to K-Means except that it does not generate new centroids, so we do not provide flowlet-style Classification algorithm here.

4. BENCHMARK ANALYSIS

In this section, we give a description of the benchmarks, their input data set and their implementation in the HAMR flowlet model. We implemented eight benchmarks in HAMR: K-Means, Classification, PageRank, KCliques, WordCount, HistogramMovies, HistogramRatings and NaiveBayes Training. The corresponding Hadoop implementation can be found in Purdue MapReduce Benchmarks Suite (PUMA) [3] and Intel HiBench [2] except K-Cliques which is implemented by us.

- **K-Means**

  K-Means is a useful data mining algorithm to cluster input data into k clusters. K-Means improves the clustering by iterating successively. Since each iteration in K-Means algorithm is exactly the same, only a single iteration is implemented and compared with PUMA K-Means which is a single iteration K-Means benchmark in Hadoop. The movie data is provided by PUMA.

  The flowlet style implementation of K-Means is shown in Alg. 1. Hadoop version uses a single job to implement one iteration of K-Means and the intermediate data are sorted and written to disks in map phase and shuffled through network and merged in reduce phase. This process causes big disk IO and network overhead that slows down the computation. On the other hand, the flowlet style implementation only passes the data location information between flowlets after processing the real data. When the real data is needed again, according to the data location information, route back by shuffle function to the node which the data resides in and work on the data. In K-Means problem, at the expense of involving more flowlets, locality can be explored and the disk IO and network overhead is reduced dramatically. Besides, it also saves memory by storing the data location information instead of huge real data.

Algorithm 2 Flowlet-style PageRank Algorithm

```
1: Initialize parameters including input/output file path, converge condition and max number of iterations.
2: while not converge and less than max number of iterations do
3:     if 1st iteration then
4:         EdgeFileLoader (loader): Read edge input file, producing ⟨ srcPage, dstPage ⟩ as the output key-value pair to HashJoinRed.
5:         HashJoinRed (reducer): For each srcPage, collect its dstPage list and save it into memory; divide srcPage's pagerank by its outdegree, and send the average value to its dstPages.
6:     else
7:         EdgeLoader (loader): For each srcPage, load its dstPage list from memory, calculate and send its average pagerank to its dstPages.
8:     end if
9:     MergeRed (reducer): For each dstPage, sum the average pageranks from its srcPages and produce the updated pagerank; compare the updated pagerank with old one and save it into memory.
10:    ContMap (mapper): Get the results of the comparison and check it converges or not.
11: end while
12: Output the final pagerank to the output file.
```

- **Classification**

  Classification classifies the movies into one of k predetermined clusters. As K-Means, it computes the cosine vector similarity of a given movie with the centroids, and assigns the movie to the cluster whose centroid it is closest to. The difference is that K-Means needs to produce new centroids for the next iteration and centroids in Classification is fixed. The input data is the same as K-Means.

  The flowlet style implementation of Classification is very similar to K-Means, so we do not show it. Classification also takes advantage of data locality awareness feature by reading/writing the data directly from/to local disk.

- **PageRank**

  PageRank assumes that more important pages incline to have more links from other pages. It counts the number and quality of links to a page to make an estimation of the importance of the page. The input data are automatically generated Web data whose hyper-

Algorithm 3 Flowlet-style K-Cliques Algorithm

1: Initialize parameters including input/output file path and K.
2: KCliquesLoader (loader): Stream input relationships *e.g.* a knows b as key-value pairs to KCliquesGraphBuilder.
3: KCliquesGraphBuilder (reducer): Store relationship in internal data structures; when all data is ready in memory, call TwoCliquesGenerator.
4: TwoCliquesGenerator (loader): Generate potential 2-Cliques from data structures; Stream 2-Cliques to 2CliquesVerify.
5: **for** I in 2 **to** K-1 **do**
6: ICliquesVerify (mapper): Take input I-Cliques and validate; If valid, generate (I+1)-Cliques and stream (I+1)-Cliques to (I+1)CliquesVerify.
7: **end for**
8: KCliquesVerify (mapper): Validate K-Cliques and output K-Cliques if valid.

links follow the Zipfian distribution. HiBench provides the input data generator for PageRank benchmark.

The flowlet style implementation of PageRank is shown in Alg. 2. Hadoop version uses two jobs to implement one iteration. HAMR version combines these two jobs into one multi-phase job. By in-memory computation, it eliminates the barrier and disk IO between jobs of Hadoop. In the phase style implementation, when a job *i.e.* an iteration completes, the intermediate data is organized in a distributed manner into the memory of all nodes. Instead of reading from disk, the following job loads the data from memory. As a result, it improves the performance tremendously.

- **K-Cliques**

  K-Cliques is a map-reduce application that works using an iterative map-reduce strategy to find fully connected sets of K vertices. A K-Clique query searches the graph to identify all cliques of size K. The input data is generated by R-MAT graph generator package.

  The flowlet style implementation of K-Cliques is shown in Alg. 3. As PageRank, K-Cliques also replaces jobs in Hadoop with flowlets. Each flowlet works asynchronously and computes in-memory. K-Cliques gets the benefits from these features of HAMR in the same way.

- **WordCount**

  WordCount counts the total occurrences of each unique word in input files. The input data is generated by making multiple copies of a book.

  It is very simple benchmark which can be implemented in two phases. The implementation in HAMR is similar to that in Hadoop. However, instead of using reduce as Hadoop, HAMR can apply partial reduce to increase the count as soon as the occurrence of the word. It eliminates the unnecessary waiting for aggregation.

  On the other hand, by using Combiner in Hadoop, the intermediate data is so small that the disk IO and network overhead can be almost overlapped by computation. As a result, performance gap between HAMR and Hadoop diminishes.

Table 1: Cluster Information.

# of compute nodes	16
CPU Count	2
CPU Type	Intel Xeon Processor E5-2620
CPU MHz	2GHz
Memory	32GB
Network Type	4x FDR InfiniBand
Local Disk Type	SATA-III
# of Local Disk	5

Algorithm 4 Flowlet-style NaiveBayes training Algorithm

1: Initialize parameters including input/output file path.
2: TextLoader (loader): Automatically split and read text files, producing ⟨ line offset, line of text ⟩ as the output key-value pair to IndexInstancesMapper.
3: IndexInstancesMapper (mapper): Parsing the line of text and produce ⟨ label, vector ⟩ key-value pair to VectorSumReducer.
4: VectorSumReducer (partial reducer): For each label, sum up its vectors and output the sum vector; sum up all feature weights in the sum vector and output the sum weight per label; produce ⟨ feature, weight ⟩ key-value pair to WeightSumReducer.
5: WeightSumReducer (partial reducer): For each feature, sum up its weights and output the sum weigh per feature.

- **HistogramMovies**

  HistogramMovies produces a histogram of movie data based on the average rating of movies. The movies data can be downloaded from PUMA website as shown above. This benchmark is very similar to WordCount and can not fully utilize the features of HAMR. The performance improvement is limited.

- **HistogramRatings**

  HistogramRating is very similar to HistogramMovies and produces a histogram of the user ratings. The input data is the same as HistogramMovies. It is very similar to HistogramMovies, so the performance improvement can not be guaranteed.

Table 2: Performance comparison between IDH 3.0 and HAMR. The unit of execution time is second.

Benchmark	Data Size	IDH 3.0	HAMR	Speedup
K-Means	300GB	5215.079	505.685	10.31
Classification	300GB	2773.660	212.815	13.03
PageRank	20GB	2162.102	158.853	13.61
KCliques	168MB	1161.246	100.945	11.50
WordCount	16GB	89.904	75.078	1.20
HistogramMovies	30GB	59.522	34.542	1.72
HistogramRatings	30GB	66.694	252.198	0.26
NaiveBayes	10GB	263.078	108.29	2.43

- **NaiveBayes Training**

  According to the Mahout implementation in Intel HiBench, NaiveBayes Training benchmark trains the Naive Bayesian trainer. The input data are generated documents whose words follow the Zipfian distribution. HiBench provides the input data generator for this benchmark.

  The flowlet style implementation of NaiveBayes Training is shown in Alg. 4. It is implemented by a single job with three flowlets that replace two jobs in Hadoop version. The intermediate data is also small. As explained in WordCount, such benchmark can not take much advantage of the features of HAMR. The performance improvement is also limited.

5. EXPERIMENT

In this section, we evaluate the performance of eight benchmarks on HAMR and compare it with Intel Distribution for Apache Hadoop Software (IDH) 3.0. Section 5.1 introduces our experimental environment. Section 5.2 reports our experimental results.

5.1 System Specification

We did the experiments on a 16 nodes cluster and Table 1 shows the cluster information in detail.

IDH 3.0 is installed on the cluster. IDH 3.0 improves Hadoop performance by optimizations for Intel Xeon processors and other Intel products. Except that, IDH 3.0 does not make any performance optimizations on Apache Hadoop. The main difference between IDH 3.0 and previous versions is that it integrates YARN aka MRv2 which manages compute resources to applications more efficiently than MRv1. Besides, IDH 3.0 supports Lustre, but we still use HDFS for Hadoop input and output. One node is configured as NameNode and ResourceManager. Other 15 nodes are configured as DataNode and NodeManager. Therefore, there are 15 nodes executing the MapReduce program in parallel.

HAMR is also deployed on the cluster. Input and output data is distributed between the local disks of each node and the engine reads and writes from the local disks. In order to make a fair comparison with IDH, 15 nodes are configured to run the HAMR instance.

5.2 Experimental Result

In this section, we report and analyze the experimental results based on the features of HAMR.

Fig. 3 shows the speedup HAMR achieved comparing with IDH 3.0. Table 2 gives the performance numbers of all benchmarks. Note that the input graph of K-Cliques is small (2^{18} vertices and 7612608 edges) that may limit parallel IO advantage of Hadoop. However, because all of the clique information must fit into memory in reduce phase, Hadoop quickly runs out of memory for larger graphs. HAMR solves this problem by building the graph into memory distributedly (We create one JVM per node instead of one JVM per task as Hadoop, so in the same node all tasks can share memory.) and making a fine-grain processing on each clique independently. This kind of distributed memory will be build into HAMR as a component called key-value store. The optimization will be also make on the key-value store *e.g.* using efficient underlying data structure.

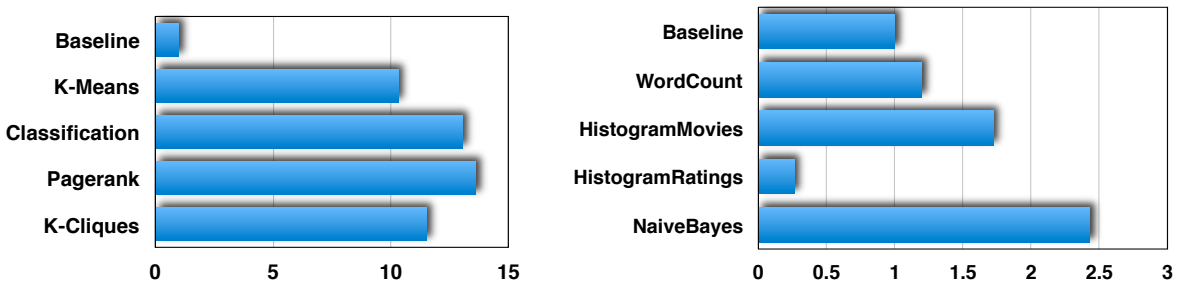
In Fig. 3(a), the performance of the four benchmarks boosts at least 6x by our engine. As expected in Section 3, these four benchmarks take advantage of engine features that beat Hadoop. We can see that the applications which are complex *i.e.* iterative and multi-phase are more likely to make use of the features of HAMR. For example, HAMR should be good at machine learning and graph algorithm.

Fig. 3(b) shows the speedup of another four benchmarks which are simple and IO intensive use case. Hadoop are very good at such applications. We can see that the speedup decreases. Note that for HistogramRating Hadoop is even 3x faster than HAMR. The reason is that the performance of HAMR will degrade when key space produced by input data distributes extremely unevenly and meanwhile flow control happens. If key space is not evenly distributed, by shuffle it is possible that few computation node will process most of the workload. When the workload beyond the computation capacity of that node, flow control will occur to slow or stop loading data. Though busy nodes still keep busy to make the data flow fluently, the nodes which process the rare keys are likely to wait for work. The worst case is that the nodes handling rare keys do not work until other nodes finish dealing with popular keys. It brings in severe under utilization of the computation resources. On the other hand, Hadoop is not suffered by data distribution problem, because of its distributed IO on map input splits and the barrier between map and reduce phase to ensure whole data availability. In HAMR, it will be helpful to design more powerful loader that can cut the whole input data into splits and randomly select splits for further processing in the following flowlets. Moreover, the contention on shared variables also harms the performance. Especially for HistogramRating, the number of the shared variables is very small *i.e.* five rates. In the partial reduce flowlet, all threads (32 threads) atomically update only one variable on each node (By shuffling, five rates are distributed to five nodes in the cluster.). Therefore, severe memory contention happens. This problem will be solved in HAMR run-time design by enforcing serialization on the variable access using a single thread each time.

We also implemented combiner in HistogramMovies and HistogramRatings. Table 3 shows the performance with combiner. We can see that combiner does not help a lot in the speedup comparing with Hadoop. The main contribution of combiner in Hadoop is to reduce the intermediate data. As a result, it saves disk IO overhead and network transmission overhead. However, in HAMR, data is kept in memory and flowing away as soon as possible. There is no disk operation involved and network overhead is overlapped by the computation. Thus, the features of HAMR limits the function of combiner. On the other hand, combiner helps flow control. That is the reason why HistogramRatings improves more that HistogramMovies by using combiner.

Table 3: Performance of HAMR using Combiner. The unit of execution time is second

Benchmark	Data Size	HAMR	Speedup
HistogramMovies	30 GB	33.234	1.79
HistogramRatings	30 GB	215.911	0.31



(a) The speedup of dataflow inspired computation engine on four benchmarks which takes advantage of its features.

(b) The speedup of dataflow inspired computation engine on four benchmarks which are IO intensive *i.e.* that can make use of batch processing of Hadoop.

Figure 3: Performance comparison between IDH 3.0 and HAMR.

6. RELATED WORK

As driven by industrial requirement, cluster computing system for distributed processing of large data sets is drawing more and more attention. One of the most famous works on this topic is Hadoop[16] which uses a very simple programming model called MapReduce. Disk based Hadoop is designed to process the data in a batch manner. The overhead of disk IO and network can largely be hidden by overlapping shuffle with computation. Hadoop performs well in offline processing cases. Hadoop is continually evolving generation by generation. Based on more sophisticated scheduling strategy YARN can make use of computation resource more efficiently. Now Hadoop forms an ecosystem which provides a lot of powerful tools *e.g.* HBase, Hive, Mahout and *etc.*. It seems that no one can replace but only integrate into Hadoop ecosystem.

Although Hadoop is the most popular and widely used bigdata solution, there are several alternatives that provide some advantages. Spark[17] is a scalable cluster computing environment which aims to fast processing by supporting in-memory computation. It is designed for the workload which is iterative and reuses the data multiple times. The data is cached in distributed memory over all nodes and parallel operations are performed on it. Twister[8] enhances the typical MapReduce computation by efficient supporting for iterative applications. It selectively caches the intermediate data into the distributed memory for the next iteration.

Different from Hadoop, Spark, and Twister which are designed for offline data processing, Storm[4] aims to real-time processing for steams of data based on a graph of computation. In Storm, a topology is created to represent the problem into multiple stages where data streams are got processed. The worker nodes are statically assigned a subset of topology. S4[14] is another cluster computing platform that supports distributed stream processing.

Since MapReduce is only able to design a simple computational pattern, GraphLab[12] and Dryad[10] use graph based on data dependency to explore parallelism in complex distributed algorithm design. By efficiently and intuitively expressing solutions, they are very suitable for graph problem and machine learning algorithm.

Our dataflow-based bigdata solution called HAMR is different from all the cluster computing systems above. It is based on the codelet execution model[18] that roots in the classical dataflow model originally proposed by Dennis[6]. The codelet execution model is a hybrid model which incorporates the advantages of macro-dataflow[13] and the Von Neumann model. The codelet execution is driven asynchronously by events which can be data dependency and resource requirement. All codelets are organized as a DAG based these relations. Inside a codelet, the program is executed in a control flow manner. The codelet execution model provides an opportunity to maximize the parallelism while minimize the overhead of resource utilization. As multi and many-core systems are becoming popular, codelet based execution model is drawn more and more attention. The works based on the codelet model include ParalleX execution model[9], SWift Adaptive Runtime Machine (SWARM)[11], TIDeFlow[15], FreshBreeze[7] and Habanero[5].

We extend the codelet execution model from multi and many-core system to cluster system. By taking the advantage of the codelet model, our cluster computing system achieves better task synchronization and resource utilization than other systems. As a result, it provides better performance. Besides, we extend traditional MapReduce programming style to more flexible multi-phase DAG style that is able to implement an complex problem in a more efficient way. Our bigdata solution also uses other the technology *e.g.* in-memory computing. As a result, it is suitable for both batch and real-time processing.

7. CONCLUSION

In this paper, we propose a new cluster computing system *i.e.* HAMR. It incorporates HPC technology into big data scenario to achieve better performance. Dataflow-based runtime efficiently schedules the data processing in a fine-grain manner. In-memory computing is supported in HAMR to reduce unnecessary disk IO. The experimental results show that HAMR can achieve up to 10x speedup comparing with Hadoop. Moreover, HAMR supports multi-phase complex workflow and can seamless integrate into Hadoop ecosystem.

During writing this paper, HAMR is making a great progress. More useful features *e.g.* key-value store and master-slave mode are developed with performance optimization. Flowlet scheduling and flow control mechanism are also improved. More user-friendly APIs are designed. In further, HAMR will provide higher level interactive interfaces like SQL. Equipped with these features, HAMR will become a powerful big data

engine for both batch and streaming processing.

8. ACKNOWLEDGMENTS

This work is supported by ET International Inc.

9. REFERENCES

[1] http://lambda-architecture.net/.

[2] https://github.com/intel-hadoop/hibench.

[3] https://sites.google.com/site/farazahmad/pumabenchmarks.

[4] http://storm.apache.org.

[5] R. Barik, Z. Budimlic, V. Cavè, S. Chatterjee, Y. Guo, D. Peixotto, R. Raman, J. Shirako, S. Taşırlar, Y. Yan, et al. The habanero multicore software research project. In *Proceedings of the 24th ACM SIGPLAN conference companion on Object oriented programming systems languages and applications*, pages 735–736. ACM, 2009.

[6] J. B. Dennis. First Version of a Data Flow Procedure Language. In *Programming Symposium Lecture Notes in Computer Science*, volume 19, pages 362–376. Springer, 1974.

[7] J. B. Dennis. Fresh Breeze: a Multiprocessor Chip Architecture Guided by Modular Programming Principles. *ACM SIGARCH Computer Architecture News*, 31(1):7–15, 2003.

[8] J. Ekanayake, B. Zhang, T. Gunarathne, S. Bae, J. Qiu, and G. Fox. Twister: A Runtime for Iterative Mapreduce. In *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing (HPDC-10)*, pages 810–818, Chicago, Illinois, USA, June 20-25, 2010.

[9] G. R. Gao, T. Sterling, R. Stevens, M. Hereld, and W. Zhu. Parallex: A study of a new parallel computation model. In *Proceedings of IEEE International Parallel and Distributed Processing Symposium (IPDPS-07)*, pages 1–6, Long Beach, CA, USA, March 26-30 2007.

[10] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly. Dryad: Distributed Data-Parallel Programs from Sequential Building Blocks. In *ACM SIGOPS Operating Systems Review*, volume 41, pages 59–72, 2007.

[11] C. Lauderdale and R. Khan. Towards a Codelet-based Runtime for Exascale Computing: Position Paper. In *Proceedings of the 2nd International Workshop on Adaptive Self-Tuning Computing Systems for the Exaflop Era (EXADAPT-12)*, pages 21–26, London, United Kingdom, March 3, 2012.

[12] Y. Low, J. Gonzalez, A. Kyrola, D. Bickson, C. Guestrin, and J. M. Hellerstein. Distributed Graphlab: A Framework for Machine Learning and Data Mining in the Cloud. *Proceedings of the VLDB Endowment*, 5(8):716–727, 2012.

[13] W. A. Najjar, E. A. Lee, and G. R. Gao. Advances in the Dataflow Computational Model. *Parallel Computing*, 25(13):1907–1929, 1999.

[14] L. Neumeyer, B. Robbins, A. Nair, and A. Kesari. S4: Distributed Stream Computing Platform. In *The 10th IEEE International Conference on Data Mining Workshops (ICDMW-10)*, pages 170–177, Sydney, Australia, December 13, 2010.

[15] D. Orozco, E. Garcia, R. Pavel, R. Khan, and G. R. Gao. TIDeFlow: The Time Iterated Dependency Flow Execution Model. In *Proceedings of the 1st Workshop on Data-Flow Execution Models for Extreme Scale Computing (DFM-11)*, pages 1–9, Galveston, TX, USA, October 10, 2011.

[16] T. White. *Hadoop: The Definitive Guide*. O'Reilly, 2012.

[17] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica. Spark: Cluster Computing with Working Sets. In *Proceedings of the 2nd USENIX Workshop on Hot Topics in Cloud Computing (HotCloud-10)*, pages 10–16, Boston, MA, USA, June 22-25, 2010.

[18] S. Zuckerman, J. Suetterlein, R. Knauerhase, and G. R. Gao. Using a "Codelet" Program Execution Model for Exascale Machines: Position Paper. In *Proceedings of the 1st International Workshop on Adaptive Self-Tuning Computing Systems for the Exaflop Era (EXADAPT-11)*, pages 64–69, San Jose, CA, USA, June 5, 2011.